PROJET L3

But : une petite introduction au Deep Learning avec l'exemple d'un perceptron multicouche.

1. Introduction

Nous allons programmer avec numpy un petit perceptron multicouche dont le but est de faire de la classification à partir d'un jeu de données très simple. Voici un exemple de tel jeu de données (Fig. 1).

On dispose de dix points dans le carré $[0,1] \times [0,1]$. Ce sont les données qui vont servir pour l'apprentissage. Elles sont classées en deux classes : les bons sont les ronds verts, les méchants les triangles bleus (c'est comme ça la vie, il y a les bons et les méchants...). Le type de réseau que l'on considère ici, le perceptron multicouche, nécessite d'avoir de telles données annotées.

Le but du réseau est de nous fournir, à partir de cet ensemble de données annotées, une frontière de décision entre le bien et le mal... Une fois connue cette frontière, nous pourrons classer les nouvelles données apparaissant en bonnes ou méchantes... Voici la frontière que l'on obtient sur cet exemple (Fig. 1).

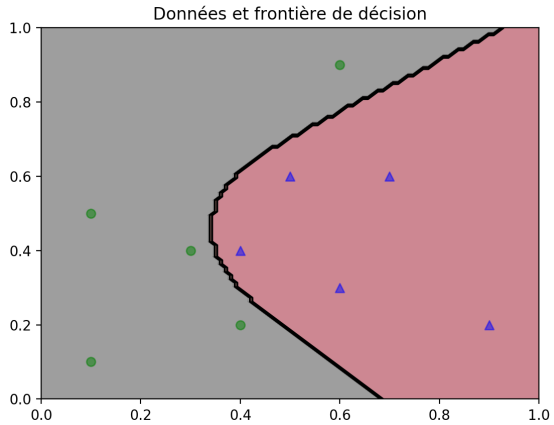

Figure 1. Les données et la frontière de décision

2. Les mathématiques

L'idée générale du perceptron est de considérer des neurones fabriqués à partir d'équations d'hyperplans et d'une fonction d'activation. Les hyperplans sont donnés par :

$$Wx + b = 0 \tag{1}$$

où x est un vecteur de $\mathbb{R}^2$, W est une matrice 2×2 et b est un vecteur de $\mathbb{R}^2$. Dans la terminologie W contient les poids, et b contient les biais.

L'activation du neurone se fait suivant la valeur de

$$\sigma(Wx + b) \tag{2}$$

où σ est la fonction d'activation considérée. Dans notre cas, nous utiliserons la sigmoide

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

L'un des avantages de la sigmoide est la forme très simple de l'équation différentielle qu'elle

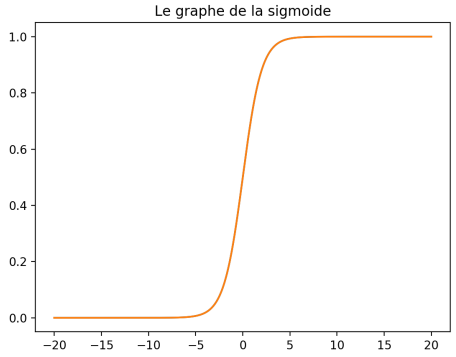

Figure 2. Le graphe de la sigmoide

satisfait. On vérifie facilement que

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{4}$$

Dans l'équation (2), les poids, W, et les biais, b, sont des paramètres que le réseau va se charger de trouver par optimisation.

Voici le petit, très petit, réseau que l'on va programmer. Il contient une couche d'entrée, une couche de sortie, et deux couches dites cachées.

Les données d'entrée sont des vecteurs x de $\mathbb{R}^2$. La sortie du layer 2 est donnée par

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2 \tag{5}$$

où $W^{[2]}$ est une matrice 2×2 et $b^{[2]}$ est un vecteur de $\mathbb{R}^2$. La sortie est donc un vecteur de $\mathbb{R}^2$.

Le layer 3 contient trois neurones. Sa sortie sera un vecteur de $\mathbb{R}^3$ donnée par

$$\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) \in \mathbb{R}^3 \tag{6}$$

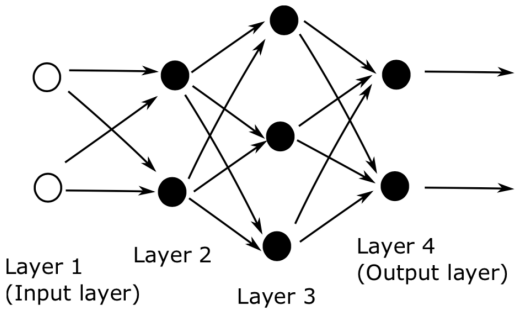

FIGURE 3. Le perceptron multicouche

où $W^{[3]}$ est une matrice 3×2 et $b^{[3]}$ est un vecteur de $\mathbb{R}^3$.

Pour le layer 4, on a pour sortie le vecteur de $\mathbb{R}^2$

$$\sigma\left(W^{[4]}\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) + b^{[4]}\right) \in \mathbb{R}^2 \tag{7}$$

où $W^{[4]}$ est une matrice 2×3 et $b^{[4]}$ est un vecteur de $\mathbb{R}^2$.

On considère alors la fonction $F : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ définie par

$$F(x) = \sigma\left(W^{[4]}\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) + b^{[4]}\right) \tag{8}$$

Cette fonction dépend des paramètres suivants : les poids

$$w_{ij}^{[2]},\ w_{ij}^{[3]},\ w_{ij}^{[4]}$$

et les biais

$$b_i^{[2]},\ b_i^{[3]},\ b_i^{[4]}$$

La dimension de l'espace des paramètres est donc 23 (c'est très petit comparé aux usines à gaz actuelles pour lesquelles cette dimension peut être de l'ordre du million...).

Cette fonction est liée à notre problème de classification de la façon suivante. Notons x^i, $i = 1, 2..., 10$ les données. On pose

$$y(x^i) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{9}$$

si x^i est bon et

$$y(x^i) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{10}$$

si x^i est méchant. La fonction de coût que l'on considère (cost function ou encore loss) est donnée par

$$\text{Cost}\left(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}\right) = \frac{1}{10}\sum_{i=1}^{10}\frac{1}{2}\|y(x^i) - F(x^i)\|_2^2 \tag{11}$$

Le but est de minimiser cette fonction Cost qui est une fonction des paramètres du réseau. On trouve ainsi des poids et des biais que l'on ré-injecte dans F. La frontière de décision est alors donnée par $F_1(x) > F_2(x)$ avec $F(x) = (F_1(x), F_2(x))$.

3. La descente de gradient

Ce type de problème de minimisation fait appel à une descente de gradient : c'est le coeur du Deep Learning. Les fonctions coût à minimiser peuvent avoir un très grand nombre de minima locaux. L'étude de la convergence des descentes est un problème particulièrement étudié à l'heure actuelle. Les Figures 3 et 4 montrent une configuration de données pour lesquelles la descente de gradient a stagné pendant 3 millions d'itérations avant de décider à se remettre à descendre...

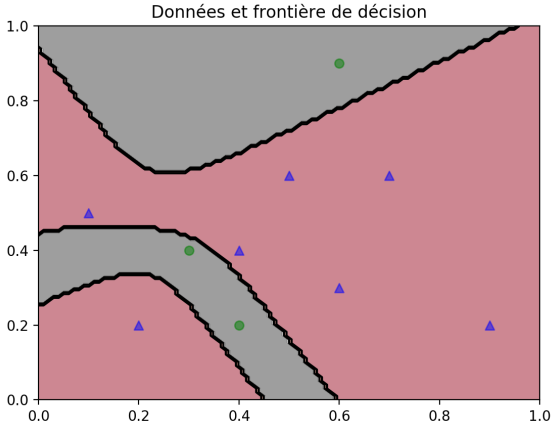

Figure 4. Une configuration particulière

On a

$$\text{Cost}(p + \Delta p) \simeq \text{Cost}(p) + \sum_{r=1}^{d}\frac{\partial \text{Cost}(p)}{\partial p_r}\Delta p_r \tag{12}$$

avec Cost : $\mathbb{R}^d \longrightarrow \mathbb{R}$, d est la dimension des paramètres, $p = (p_1, p_2, ..., p_d)$. Par définition

$$\langle \nabla \text{Cost}(p), v\rangle = d_p\text{Cost}(v) \tag{13}$$

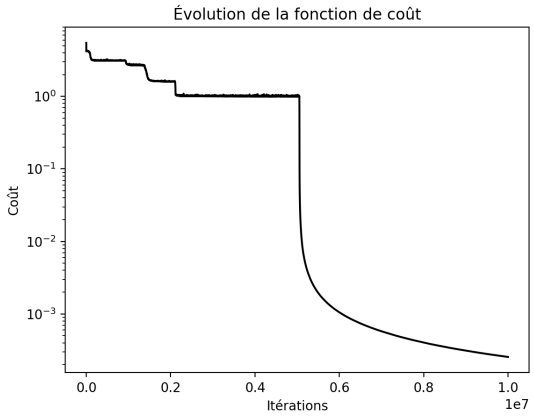

FIGURE 5. L'évolution de la fonction coût correspondante

pour tout vecteur v de $\mathbb{R}^d$. On peut considérer dans notre contexte (euclidien) que $\nabla \text{Cost}(p)$ est le vecteur de $\mathbb{R}^d$ de composantes

$$(\nabla \text{Cost}(p))_r = \frac{\partial \text{Cost}(p)}{\partial p_r} \tag{14}$$

L'équation (12) peut donc s'écrire

$$\text{Cost}(p + \Delta p) \simeq \text{Cost}(p) + \nabla \text{Cost}(p)^T \Delta p \tag{15}$$

Pour faire diminuer le plus rapidement la fonction Cost, on choisit Δp de sorte que

$$\nabla \text{Cost}(p)^T \Delta p$$

soit le plus négatif possible. D'après Cauchy - Schwarz, il faut choisir

$$\Delta p = -\eta \nabla \text{Cost}(p)$$

où η est un pas appelé le Learning Rate. La mise à jour est donnée par

$$p \longleftarrow p - \eta \nabla \text{Cost}(p) \tag{16}$$

Dans notre cas, comme dans la plupart des cas, notre fonction coût est une somme. On a donc

$$\nabla \text{Cost}(p) = \frac{1}{N} \sum_{i=1}^{N} \nabla \text{Cost}_i(p) \tag{17}$$

avec N le nombre de données (ici 10) et

$$\text{Cost}_i = \frac{1}{2} \|y(x^i) - F(x^i)\|_2^2 \tag{18}$$

En général, le nombre N est très grand ce qui pose des problèmes pour le calcul du gradient complet. Pou pallier cette difficulté, on utilise un gradient dit stochastique en expliquant comment sélectionner un nombre restreint de fonctions Cost_i et ce de façon à

couvrir au mieux les données. Ici, par soucis de simplicité, on va sélectionner une seule fonction coût Cost_i en tirant, pour chaque mise à jour, un indice i de façon aléatoire et uniforme dans l'ensemble $\{1, 2, ..., N = 10\}$:

$$p \longleftarrow p - \eta \nabla \text{Cost}_i(p)$$

4. La backprop

On commence par introduire quelques notations plus génériques pour les calculs. On note $x \in \mathbb{R}^{n_1}$ les données d'entrée (pour nous $n_1 = 2$). On note également $a_j^{[l]}$ la sortie du j-ème neurone du layer l. On a donc

$$a^{[1]} = x \in \mathbb{R}^{n_1} \tag{19}$$

$$a^{[l]} = \sigma\left(W^{[l]} a^{[l-1]} + b^{[l]}\right) \in \mathbb{R}^{n_l} \tag{20}$$

pour $l = 2, 3, ..., L$ (pour nous $L = 4$ et $n_4 = 2$). On aura aussi besoin de l'entrée du j-ème neurone du layer l (avant activation)

$$z_j^{[l]} = \left(W^{[l]} a^{[l-1]} + b^{[l]}\right)_j \tag{21}$$

Notre problème se résume donc à calculer le gradient de la fonction

$$C = \frac{1}{2} \|y - a^{[L]}\|_2^2 \tag{22}$$

vue comme fonction des poids et des biais (qui interviennent dans la sortie $a^{[L]}$ du réseau).

La propagation de l'information dans le réseau est donnée par

$$a^{[l]} = \sigma\left(z^{[l]}\right) \tag{23}$$

pour $l = 2, 3, ..., L$.

On note $\delta^{[l]} \in \mathbb{R}^{n_l}$ le vecteur de composantes

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \tag{24}$$

pour $j = 1, 2, ..., n_l$ et $l = 2, 3, ..., L$.

Pour faire les calculs on peut utiliser le produit de Hadamard : si x et y sont deux vecteurs de $\mathbb{R}^n$, le produit de Hadamard $x \circ y$ de x et y est le vecteur de $\mathbb{R}^n$ de composantes $(x \circ y)_i = x_i y_i$, pour tout $i = 1, 2, ..., n$. Tout est dans le résultat suivant.

Proposition - On a

$$\delta^{[L]} = \sigma'\left(z^{[L]}\right) \circ \left(a^{[L]} - y\right) \tag{25}$$

$$\delta^{[l]} = \sigma'\left(z^{[l]}\right) \circ \left(W^{[l+1]}\right)^T \delta^{[l+1]} \tag{26}$$

pour $2 \leq l \leq L-1$;

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \tag{27}$$

pour $2 \leq l \leq L$;

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \tag{28}$$

pour $2 \leq l \leq L$, en notant $w_{jk}^{[l]}$ les coefficients de la matrice $W^{[l]}$.

Preuve - Exercice !!

On peut encore condenser un peu l'écriture en introduisant la matrice $D^{[l]}$ de taille $n_l \times n_l$ diagonale dont les termes diagonaux sont

$$D_{ii}^{[l]} = \sigma'\left(z_i^{[l]}\right)$$

On a alors

$$\delta^{[L]} = D^{[L]}\left(a^{[L]} - y\right)$$

et

$$\delta^{[l]} = D^{[l]}\left(W^{[l+1]}\right)^T \delta^{[l+1]}$$

L'écriture finale est donc

$$\delta^{[l]} = D^{[l]}\left(W^{[l+1]}\right)^T D^{[l+1]}\left(W^{[l+2]}\right)^T \cdots D^{[L-1]}\left(W^{[L]}\right)^T D^{[L]}\left(a^{[L]} - y\right) \tag{29}$$

Il ne faut pas oublier que le calcul de la dérivée de la sigmoide est trivial. L'equation (29) est l'équation de la *Back-propagation*.

Le perceptron muticouche est l'exemple type d'un réseau *Forward/Backward*. La passe Foraward permet d'évaluer les fonctions d'activation, les poids et la sortie $a^{[L]}$. La passe Backward permet de calculer l'erreur et la mise à jour. Ce double mécanisme doit bien apparaitre dans le programme.