Réseaux de neurones (Notes de cours)

Romain Tavenard

Chapitre 1

Préambule

Ce document est un ensemble de notes associées au module de classification supervisée pour la deuxième année de master de Statistiques de l'Université de Rennes 2, et plus particulièrement la partie sur les réseaux de neurones. Il est distribué librement (sous licence CC BY-NC-SA plus précisément) et se veut évolutif, n'hésitez donc pas à faire vos remarques à son auteur dont vous trouverez le contact sur sa page web.

1.1 Contenu du cours

Chapitre 2

Régression et perceptron

2.1 Retour sur la régression linéaire

Dans le cas d'une régression aux moindres carrés ordinaires, le modèle est le suivant :

$$y_i = \beta_0 + \sum_j \beta_j x_{i,j} + \epsilon_i,$$

où les ϵ_i sont i.i.d. gaussiens de moyenne nulle.

On note

$$\hat{y_i} = \beta_0 + \sum_j \beta_j x_{i,j}$$

et l'on souhaite estimer les paramètres β_j minimisant la quantité $L(\beta) = \frac{1}{2}\sum_i (y_i - \hat{y_i})^2$.

Vous connaissez probablement une forme explicite permettant d'exprimer les β_j en fonction des y_i et des $x_{i,j}$. Si vous ne connaissiez pas cette forme explicite (ce sera notre cas pour à peu près tous les réseaux de neurones que nous verrons dans la suite), un moyen d'estimer les β_j serait de minimiser $L(\beta)$ par descente de gradient. Pour cela, il nous faudra calculer les dérivées de L par rapport aux différents β_j :

$$\frac{\partial L}{\partial \beta_0} = -\sum_i (y_i - \hat{y_i}) \tag{2.1}$$

$$\forall j \geq 1, \frac{\partial L}{\partial \beta_j} = -\sum_i (y_i - \hat{y_i}) x_{i,j} \tag{2.2}$$

Travail personnel: Reprendre les formules précédentes pour le cas d'une régression logistique binaire (y_i peut prendre les valeurs 1 ou -1).

Dans la suite, on aura parfois recours à cet exemple de la régression aux moindres carrés ordinaires pour comprendre comment sont estimés les paramètres des réseaux de neurones. L'extension à des problèmes de classification (binaire ou non) revient alors à effectuer le même type de modifications que celles présentes dans ce travail personnel (changer l'expression de la vraisemblance, passer à la log-vraisemblance, recalculer les dérivées).

2.2 Le perceptron : notations et représentation

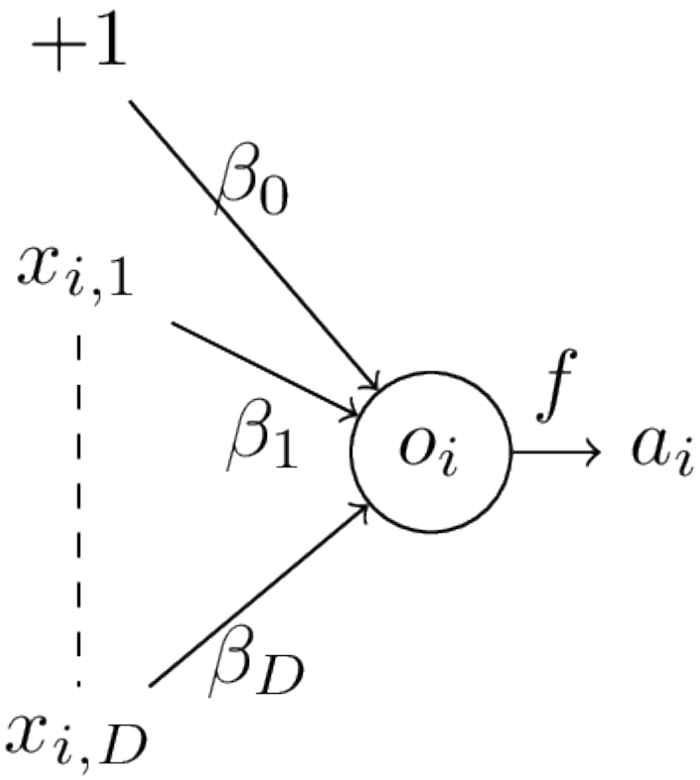
Avant de présenter ce qu'est un réseau de neurones, nous allons nous intéresser à un modèle dans lequel on n'a qu'un neurone : le perceptron. Dans ce modèle, on suppose :

$$y_i = f\left(\beta_0 + \sum_j \beta_j x_{i,j}\right) + \epsilon_i$$

et on note

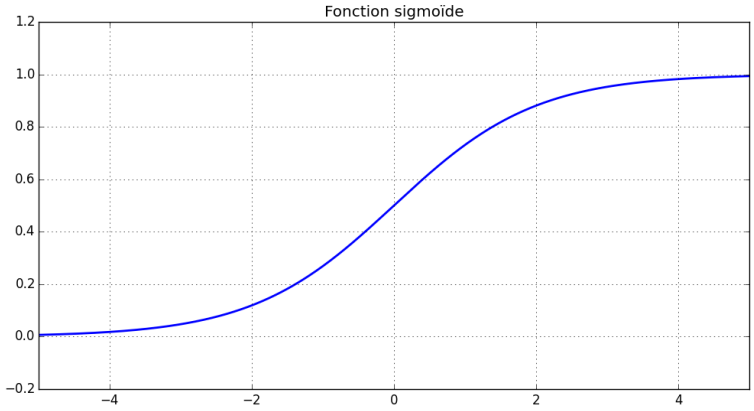
$$\begin{array}{rcl} o_i & = & \beta_0 + \sum_j \beta_j x_{i,j} \\ \hat{y_i} = a_i & = & f(o_i) = f\left(\beta_0 + \sum_j \beta_j x_{i,j}\right) \end{array}$$

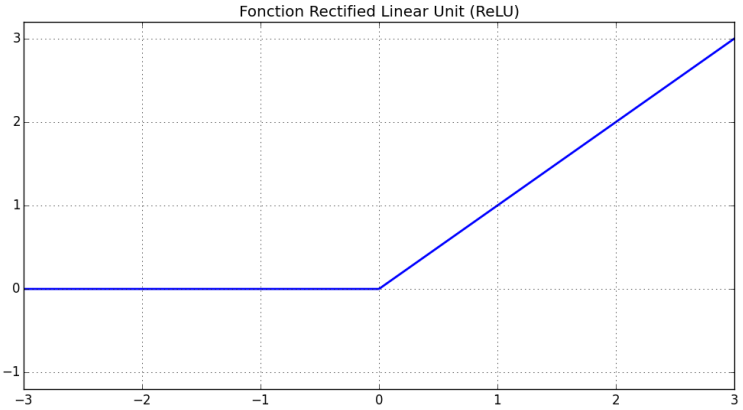
Les paramètres de ce modèle sont les β_j et la fonction f est appelée fonction d'activation.



Pour cette fonction d'activation, plusieurs choix sont possibles. On peut citer pour exemples les fonctions sigmoïde et *ReLU* (Rectified Linear Unit) :

$$\begin{aligned} f_1(o_i) &= \tfrac{1}{1+e^{-o_i}} \\ f_2(o_i) &= ReLU(o_i) = \begin{cases} o_i, & \text{si } o_i \geq 0 \\ 0, & \text{sinon} \end{cases} \end{aligned}$$





Puisque nous en aurons besoin par la suite, nous pouvons d'ores et déjà calculer la dérivée de ces fonctions :

$$\begin{aligned} \frac{\partial f_1}{\partial o_i} &= f_1(o_i) \cdot (1 - f_1(o_i)) \\ \frac{\partial f_2}{\partial o_i} &= \begin{cases} 1, & \text{si } o_i > 0 \\ 0, & \text{si } o_i < 0 \end{cases} \end{aligned}$$

Notez que f_2 n'est pas dérivable en 0. On choisit par convention de prolonger sa dérivée en 0 par la valeur 0. Il faut comprendre ici que ce calcul de dérivée nous servira à effectuer notre descente de gradient. En pratique, la probabilité que l'on ait à évaluer $\frac{\partial f_2}{\partial o_i}$ en 0 est nulle.

2.2.1 Optimisation en pratique

Considérons une fois encore le problème de régression aux moindres carrés ordinaires avec ce modèle. On cherche donc à minimiser

$$L(\beta) = \frac{1}{2}\sum_i (y_i - \hat{y_i})^2 = \frac{1}{2}\sum_i \underbrace{(y_i - a_i)^2}_{L_i(\beta)}$$

et l'on n'a, cette fois, plus d'expression analytique pour les β_j optimaux. On va donc chercher à effectuer une descente de gradient.

Pour cela, on doit calculer $\frac{\partial L_i}{\partial \beta_j}$ pour tout i et pour $j = 0 \dots D$ où D est la dimension de l'espace dans lequel vivent nos x_i.

Avant toute chose, rappelons la règle de dérivation en chaîne qui nous sera utilie par la suite :

Si $z = f(y)$ et $y = g(x)$, alors on a :

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Revenons maintenant à nos $\frac{\partial L_i}{\partial \beta_j}$. On sépare le cas $j = 0$ et on obtient :

$$\begin{array}{rrcl} & \frac{\partial L_i}{\partial \beta_0} & = & \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial o_i} \cdot \frac{\partial o_i}{\partial \beta_0} \\ & & = & (y_i - a_i) \cdot \frac{\partial a_i}{\partial o_i} \cdot 1 \\ \forall j > 0, & \frac{\partial L_i}{\partial \beta_j} & = & \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial o_i} \cdot \frac{\partial o_i}{\partial \beta_j} \\ & & = & (y_i - a_i) \cdot \frac{\partial a_i}{\partial o_i} \cdot x_{i,j} \end{array}$$

où $\frac{\partial a_i}{\partial o_i}$ vaut $\frac{\partial f_1}{\partial o_i}$ ou $\frac{\partial f_2}{\partial o_i}$ selon la fonction d'activation choisie. Par exemple, dans le cas de la fonction d'activation sigmoïde, on aura :

$$\frac{\partial a_i}{\partial o_i} = a_i \cdot (1 - a_i).$$

En pratique, pour effectuer notre descente de gradient, on effectuera les opérations suivantes pour chaque exemple d'apprentissage (x_i, y_i) :

1. Calculer a_i puis l'erreur $\epsilon_i = y_i - a_i$ (passe *forward*) ;
2. Utiliser ces quantités pour calculer les $\frac{\partial L_i}{\partial \beta_j}$ (rétropropagation, ou *backpropagation*);

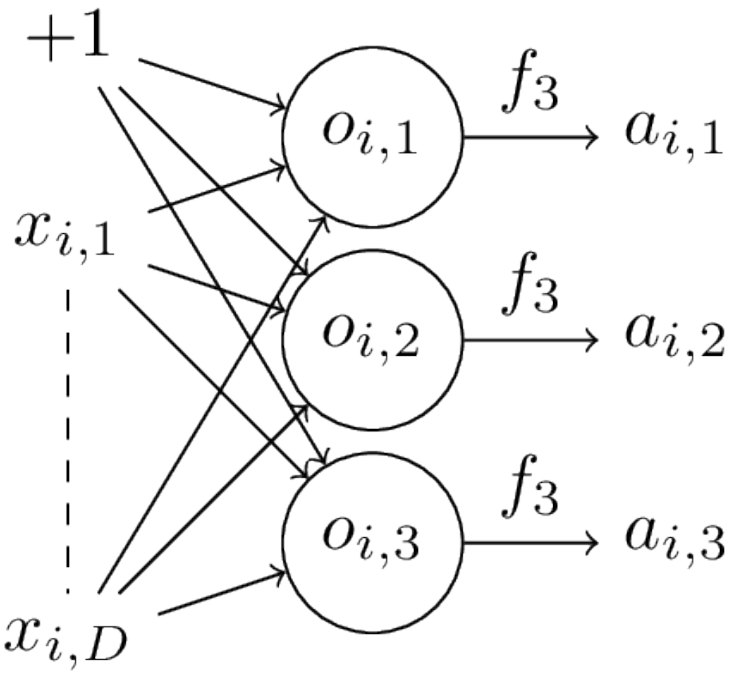
Finalement, on calculera $\frac{\partial L}{\partial \beta_j}$ comme la somme des $\frac{\partial L_i}{\partial \beta_j}$ et on mettra à jour les β_j selon ce qui est indiqué par l'algorithme de descente de gradient choisi.

On remarque ici que cette descente de gradient sera très facilement parallélisable. En effet, les $\frac{\partial L_i}{\partial \beta}$ peuvent être calculés de manière indépendante (et notamment en parallèle) et l'algorithme complet vu comme un exemple de *Map-Reduce*.

2.2.2 Cas de la classification multi-classes

Lorsque le problème auquel on est confronté est un problème de classification binaire, le modèle de perceptron présenté plus haut peut convenir, où la valeur de sortie est interprétée comme la probabilité (prédite par le modèle) d'être en présence d'une instance de la classe `1`. Pour cela, il faut bien entendu utiliser une fonction d'activation dont l'image est $[0, 1]$.

En revanche, lorsque l'on a affaire à un problème de classification multi-classes (avec un nombre de classes $K > 2$), on devra utiliser plusieurs perceptrons (un pour chaque classe à prédire) tel que celui-ci :



Il faudra alors avoir recours à une fonction d'activation un peu particulière qui garantisse que la somme des probabilités prédites pour chacune des classes soit égale à 1. Pour cela, on utilise généralement la fonction $softmax$ définie comme :

$$\forall 1 \leq k \leq K,\ f_3(o_{i,k}) = softmax(o_{i,k}) = \frac{e^{o_{i,k}}}{\sum_{k'=1}^{K} e^{o_{i,k'}}}$$

On peut aisément montrer que f_3 :

- a pour image $]0, 1[$;
- est dérivable en tout point de $\mathbb{R}$;
- vérifie la propriété $\forall i, \sum_k f_3(o_{i,k}) = 1$.

On vient ici de construire un réseau de neurones à une couche. Les trois perceptrons (ou neurones) utilisés ici sont connectés aux mêmes entrées, on dit donc qu'ils appartiennent à la même couche.

Dans la suite de ce cours, on s'intéressera aux réseaux de neurones multi-couches.

Chapitre 3

Perceptron multi-couches

http://cs231n.github.io/neural-networks-1/

3.1 Cas d'un réseau de neurones à une couche cachée

3.1.1 Calcul de gradient

3.1.2 Visualisation de frontières apprises

Exemple intéressant : http://cs231n.github.io/neural-networks-case-study/

3.2 Cas multi-couches

3.2.1 Calcul de gradient

http://cs231n.github.io/optimization-2/

3.2.2 Visualisation

Comprendre qu'on apprend une représentation (eg PCA améliorée) suivie d'un classifieur linéaire (logistic regression)

- https://rajarsheem.wordpress.com/2017/05/04/neural-networks-dynamics/
- http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

3.3 Quelques considérations pratiques

3.3.1 Initialisation des poids du réseau

3.3.2 Régularisation

http://cs231n.github.io/neural-networks-2/

3.3.2.1 Régularisations L1/L2

3.3.2.2 Utilisation du *dropout*