TP5 : Perceptron

1 Mise en place

Cette dernière séance de travaux pratiques a pour objectif d'introduire les bases de la théorie des réseaux de neurones et d'introduire le cas particulier du perceptron. Nous verrons qu'il y a plusieurs types de perceptron et nous mettrons certains d'entre eux en pratique pour résoudre des problèmes d'apprentissage supervisé.

Afin de faciliter la récupération des données et de disposer de certaines fonctions graphiques dans la suite de cette séance, nous commençons par charger le script *tp5.R*,

```
source("http://www.math.univ-toulouse.fr/~xgendre/ens/m2se/tp5.R")
```

2 Notion de neurone

Un **neurone** est un objet mathématique qui fut à l'origine introduit, entre autres choses, pour modéliser le fonctionnement du cerveau humain dans le cadre d'études de la cognition. En interconnectant plusieurs neurones, nous formons alors un **réseau de neurones**. Nous ne présenterons pas ici le parallèle existant entre la notion biologique de neurone et la version mathématique. Le lecteur intéressé trouvera de nombreuses précisions sur la question avec internet.

2.1 Introduction

Un neurone est donc l'unité élémentaire de calcul d'un réseau de neurones. Son principe général est de retourner **une information en sortie** à partir de **plusieurs informations en entrée**. L'information entrante peut être, par exemple, issue de l'information sortante d'autres neurones dans le cadre d'un réseau. Plus précisément, nous notons $x_1, \dots, x_n \in \mathbb{R}$ les informations entrantes et, pour chaque $i \in \{1, \dots, n\}$, nous associons un poids $w_i \in \mathbb{R}$ à x_i. Contrairement aux poids que nous avons utilisés en cours, il faut noter que les w_i sont ici des nombres réels potentiellement négatifs et dont la somme ne vaut pas nécessairement 1. Nous introduisons également un poids $w_0 \in \mathbb{R}$, appelé **coefficient de biais**, associé à une information virtuelle $x_0 = -1$ dont le rôle sera précisé ultérieurement. L'information traitée par le neurone n'est pas l'ensemble des x_i mais leur moyenne pondérée,

$$\bar{x} = \sum_{i=0}^{n} w_i x_i = \sum_{i=1}^{n} w_i x_i - w_0 \ .$$

Le rôle d'un neurone est de fournir une réponse y entre 0 et 1 à partir de $\bar{x}$. Nous utilisons pour cela une fonction $g : \mathbb{R} \to [0, 1]$, appelée **fonction d'activation**. Lorsque la réponse est proche de 1, nous dirons que le neurone est **actif** et pour une réponse proche de 0, le neurone sera dit **inactif**. Nous avons donc la sortie du neurone qui est définie par

$$y = g(\bar{x}) = g\left(\sum_{i=1}^{n} w_i x_i - w_0\right) \ .$$

Il faut noter que si g est une fonction linéaire des informations entrantes, ce modèle correspond à la régression linéaire. Bien entendu, les réseaux de neurones deviennent plus intéressants quand

g n'est pas une fonction linéaire. Selon la définition, toute fonction à valeurs dans $[0, 1]$ peut être utilisée comme fonction d'activation. Cependant, deux fonctions sont particulièrement prisées en pratique :

– la fonction de Heaviside,

$$\forall x \in \mathbb{R},\ g(x) = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{sinon.} \end{cases}$$

```
plot(Heaviside, xlim = c(-5, 5))
```

– la fonction sigmoïde,

$$\forall x \in \mathbb{R},\ g(x) = \frac{1}{1 + e^{-x}}\ .$$

```
plot(Sigmoide, xlim = c(-5, 5))
```

Ces deux fonctions sont croissantes et la sigmoïde présente l'avantage d'être dérivable sur $\mathbb{R}$. Ces deux fonctions "basculent" l'état inactif du neurone à l'état actif quand $\bar{x}$ dépasse 0. Le rôle du coefficient de biais est donc de déplacer ce changement d'état pour le situer autour de w_0,

$$\bar{x} = \sum_{i=1}^{n} w_i x_i - w_0 > 0 \Leftrightarrow \sum_{i=1}^{n} w_i x_i > w_0\ .$$

En utilisant une notation vectorielle, nous pouvons écrire $x = (x_1, \ldots, x_n)' \in \mathbb{R}^n$, $w = (w_1, \ldots, w_n)' \in \mathbb{R}^n$ et ainsi définir $\bar{x}$ avec le produit scalaire de ces vecteurs,

$$\bar{x} = w \cdot x - w_0\ .$$

La région où l'état du neurone change est donc définie par

$$\mathcal{X}_w = \{x \in \mathbb{R}^n \text{ tels que } w \cdot x = w_0\}\ .$$

Si $w \neq 0$, l'espace $\mathcal{X}_w$ est un hyperplan qui sépare $\mathbb{R}^n$ en deux parties selon que $w \cdot x$ est inférieur ou supérieur à w_0. Ainsi, le principe d'un neurone est donc de séparer l'espace des variables en deux parties : l'une dans laquelle le neurone est actif, l'autre dans laquelle il est inactif. Plus généralement, un réseau de neurones va mener à plusieurs séparations de l'espace des variables qui nous permettront de le "découper". Sur chaque partie de l'espace ainsi découpé, le réseau prendra un certain état que nous pourrons, par exemple, associer à une classe donnée dans un cadre d'apprentissage supervisé.

2.2 Premiers exemples

Pour illustrer les notions introduites précédemment, nous allons considérer le cas de $n = 2$ variables binaires (*i.e.* à valeurs dans $\{0, 1\}$) en prenant la fonction de Heaviside comme fonction d'activation. Ce cas est particulièrement simple car le couple (x_1, x_2) des variables d'entrée ne peut alors prendre que 4 valeurs qui sont résumées dans une **table de vérité** avec la valeur prise par la variable de sortie y dans la dernière colonne,

x_1	x_2	y
0	0	y_{00}
0	1	y_{01}
1	0	y_{10}
1	1	y_{11}

Étant données les valeurs prises par y, la question est donc de trouver un réseau de neurones compatible avec ces "données".

Dans un premier temps, nous allons étudier une situation où un seul neurone suffit. Il s'agit de la **disjonction logique** (ou porte OR) associée à la table de vérité suivante,

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Puisqu'il n'y a que deux variables, nous pouvons représenter ces données dans le plan en mettant un point rouge si la sortie vaut 0 et bleu si la sortie vaut 1,

```
plot(x1b, x2b, col = yOR)
```

A l'aide de la fonction `abline`, trouvez et tracez une droite qui sépare les points de couleurs différentes. En faisant le parallèle avec la section précédente, déduisez trois nombres réels w_0, w_1 et w_2 tels que y soit égal à 1 si et seulement si $w_1x_1 + w_2x_2 > w_0$. En particulier, vérifiez que $w_0 = 1/2$ et $w_1 = w_2 = 1$ est une solution à ce problème. Le réseau (de un neurone) que nous venons de décrire est parfois représenté comme sur la Figure 1.

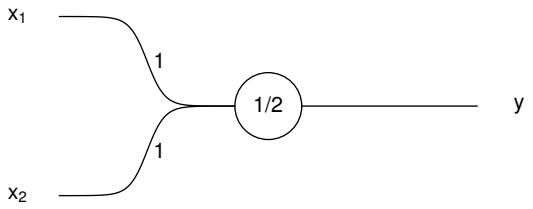

FIGURE 1 – Neurone de la porte OR

Faites de même pour le cas de la **conjonction logique** (ou porte AND),

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

que vous pouvez représenter avec la commande

```
plot(x1b, x2b, col = yAND)
```

En particulier, trouvez des valeurs de w_0, w_1 et w_2 convenables et représenter le neurone obtenu.

Considérez maintenant le cas de la porte OR exclusive (ou porte XOR) donnée par la table

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

et représentée par

```
plot(x1b, x2b, col = yXOR)
```

Quel est le problème dans ce cas ? Ce problème ne peut pas être résolu à l'aide d'un unique neurone et nous y reviendrons à la fin de cette séance.

3 Perceptron mono-couche

3.1 Réseau de neurones

Le premier réseau de neurones que nous allons voir est le perceptron mono-couche. Les neurones ne sont pas, à proprement parlé, en réseau mais ils sont considérés comme un ensemble. Comme dans la Section 2.1, nous considérons n variables d'entrée $x_1, \dots, x_n \in \mathbb{R}$. Le perceptron mono-couche est composé de p neurones, chacun étant connecté à toutes les variables d'entrée. Globalement, ce réseau dispose donc de n entrées et de p sorties et il peut être représenté comme sur la Figure 2.

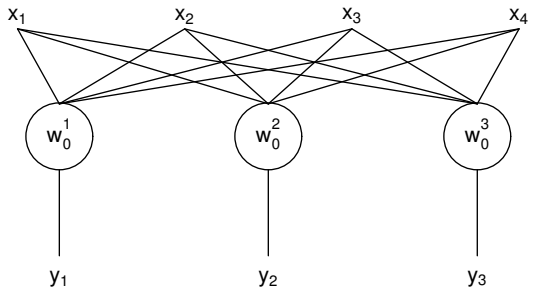

FIGURE 2 – Perceptron mono-couche à $n = 4$ entrées et $p = 3$ sorties

Pour les notations, nous écrirons $x = (x_1, \dots, x_n)' \in \mathbb{R}^n$ pour le vecteur des entrées, $y = (y_1, \dots, y_p)' \in [0,1]^p$ pour celui des sorties et, pour tout $j \in \{1, \dots, p\}$, $w_0^j \in \mathbb{R}$ est le coefficient de biais du $j^{\text{ème}}$ neurone. De plus, pour tout $i \in \{1, \dots, n\}$ et tout $j \in \{1, \dots, p\}$, $w_i^j \in \mathbb{R}$ est le poids associé à x_i par le $j^{\text{ème}}$ neurone. Nous pouvons ainsi définir la matrice $W = (w_i^j)_{i,j}$ des poids de taille $n \times p$.

3.2 Apprentissage supervisé

Le réseau de neurones du perceptron mono-couche permet de construire une procédure de classification en p classes en considérant chaque neurone comme un "indicateur" d'une classe. La classe affectée à une série d'entrées est celle associée au neurone retournant la sortie maximale. Construire une telle procédure revient à choisir des coefficients de biais $w_0^1, \dots, w_0^p \in \mathbb{R}$ et une matrice W des poids. Comme pour les autres procédures d'apprentissage supervisé, nous allons construire celle-ci à partir d'un jeu de données étiquetées en considérant des poids et des coefficients de biais qui rendent un certain critère d'erreur (moindres carrés, entropie, ...) minimal.

Nous ne développerons pas ici les algorithmes d'optimisation utilisés en pratique pour trouver W et les w_0^j. Les approches les courantes sont basées sur des méthodes de descente de gradient.

Parmi elles, l'algorithme de Widrow-Hoff (dit *règle delta* ou *back propagation*) est une des plus populaires. Le lecteur intéressé trouvera des explications, des preuves et des implémentations dans les références de fin.

Pour mettre en pratique le perceptron mono-couche dans le cadre de cette séance, nous allons utiliser le paquet `neuralnet` (si ce dernier n'est pas installé sur votre poste, utilisez la commande `install.packages("neuralnet")`). Ce paquet implémente, entre autres, la méthode de Widrow-Hoff et sa principale fonction est `neuralnet`,

```
library(neuralnet)

## Loading required package: grid
## Loading required package: MASS

help(neuralnet)
```

Pour mettre en pratique cette procédure, nous allons considérer les données de fertilité de 248 femmes issues du travail de Trichopoulos *et al.* (1976). Ces données sont accessibles dans R sous le nom de `infert`. Pour obtenir plus de détails, consultez l'aide,

```
help(infert)
```

La fonction `neuralnet` fonctionne de manière similaire à `lm`. Les trois arguments principaux à fournir sont :

`formula` ce paramètre nous permet d'indiquer quelle est la variable à expliquer et quelles sont les variables explicatives. Sa syntaxe est la même que pour la fonction `lm`. Nous n'entrons pas ici dans les détails des objets de type *formula* de R et nous nous contenterons d'utiliser la variable `infert_form` définie comme suit,

```
infert_form <- case ~ age + parity + induced + spontaneous
```

`data` il s'agit du jeu de données contenant les variables annoncées dans la formule. Pour nous, il s'agit simplement de `infert`.

`hidden` le rôle de ce paramètre sera expliqué dans la prochaine section. Pour le perceptron mono-couche, il suffit de le mettre à `0`.

Un premier essai consiste donc à exécuter la commande suivante,

```
mono <- neuralnet(infert_form, data = infert, hidden = 0)
```

Il y a plusieurs façon d'avoir des détails sur le perceptron obtenu. Entrez, par exemple, les commandes suivantes,

```
mono
mono$result.matrix
```

Avec l'aide sur les objets définis par `neuralnet`, expliquez ce qu'est la valeurs `Steps` ? Qu'est-ce qu'indique l'erreur ? Où pouvons-nous lire les poids ? *et cætera* ...

Bien entendu, il est également possible de visualiser le perceptron,

```
plot(mono)
```

Expliquez le graphique obtenu. De plus, si vous disposez de nouvelles données entrantes, vous pouvez appliquer le réseau de neurones obtenu à l'aide de la fonction `compute`,

```
help(compute)
```

4 Perceptron multicouche

Il est possible de généraliser le perceptron en empilant plusieurs perceptrons mono-couches. De cette façon, les sorties d'une couche sont les entrées de la suivante. Ce réseau de neurone est, bien entendu, plus compliqué mais s'utilise de manière similaire à un perceptron mono-couche. L'intérêt principal est d'être capable d'approcher des comportements moins linéaires et d'obtenir ainsi des erreurs plus faibles sur les données d'entraînement (forte adéquation aux données) au prix d'une complexité plus grande (voir dernier chapitre du cours).

Dans le cas d'un perceptron à deux couches ayant n entrées et p sorties, il nous faut donc choisir le nombre de neurones à mettre dans la couche intermédiaire. En pratique, il vaut toujours mieux avoir trop de neurones cachées que pas assez si nous souhaitons capter les phénomènes non-linéaires. Cependant, si nous utilisons un trop grand nombre de neurones intermédiaires, nous risquons le phénomène d'*over fitting* sur nos données d'apprentissage. Il est possible d'utiliser la validation croisée pour trouver une solution à ce problème pratique. Le paramètre de `neuralnet` qui permet d'indiquer le nombre de neurones cachés est `hidden`. Pour construire le réseau à deux couches avec 2 neurones cachés, entrez la commande suivante,

```
multi <- neuralnet(infert_form, data = infert, hidden = 2)
```

Comparez l'erreur avec l'exemple mono-couche. Représentez graphiquement ce réseau de neurone. Faites varier les paramètres (en particulier, `algorithm`).

Pour terminer, nous proposons de revenir sur le problème de la porte XOR vu à la Section 2.2. Nous créons un simple jeu de données `xor_data` et un premier perceptron par les commandes suivantes,

```
x1 <- c(0, 0, 1, 1)
x2 <- c(0, 1, 0, 1)
y <- c(0, 1, 1, 0)
xor_data <- matrix(c(x1, x2, y), ncol = 3)
colnames(xor_data) <- c("x1", "x2", "y")
perceptron <- neuralnet(y ~ x1 + x2, data = xor_data)
```

Faites varier le paramètre `hidden` et étudiez les perceptrons obtenus en terme d'erreur. Répétez l'opération (voir le paramètre `rep` de `neuralnet`) pour décrire une "bonne" structure du réseau pour cette porte XOR. Introduisez une nouvelle variable d'entrée `x3 <- c(0,0,0,1)`. Que représente `x3` ? Étudiez à nouveau les perceptrons obtenus. Trouvez un réseau de neurones simple pour la porte XOR.

Références

[1] J. Friedman, T. Hastie and R. Tibshirani, *The Elements of Statistical Learning : Data Mining, Inference and Prediction*, 2009

[2] Ben Kröse and Patrick van der Smagt, *An introduction to Neural Network*, Eighth edition, 1996

[3] Alp Mestan, *Introduction aux Réseaux de Neurones Artificiels Feed Forward*, `http://alp.developpez.com/tutoriels/intelligence-artificielle/reseaux-de-neurones`, 2008

[4] Frauke Günther and Stefan Fritsch, *neuralnet : Training of Neural Networks*, 2010