Practicals 5 : Perceptron

1 Framework

The aim of this last session is to introduce the basics of neural networks theory through the special case of the perceptron. We will see that there exist many kinds of perceptron and we will focus on some of them to handle supervised learning problems in practice.

As usual, to get the data and some graphical functions in the sequel, we load the script *tp5.R*,

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

2 Neuron concept

A **neuron** is a mathematical object that was first introduced, among other reasons, to model how the human brain works from the point of view of the cognitive sciences. Connecting several neurons, we give form to a **neural network**. We are not going here to speak about the parallel between the biological concept of a neuron and this mathematical approach. The interested reader will find a lot of additional informations on the internet.

2.1 Introduction

A neuron is the atomic computing unit of a neural network. From a very general outlook, it returns some **output information** from several **input data**. The inputs can be the outputs of other neurons in the framework of a network. More precisely, we denote by $x_1, \dots, x_n \in \mathbb{R}$ the inputs and, for each $i \in \{1, \dots, n\}$, we give a weight $w_i \in \mathbb{R}$ to x_i. Unlike what we have seen during the lecture, here, the weights w_i are real numbers that can be negative with a sum not necessarily equal to 1. We also introduce a weight $w_0 \in \mathbb{R}$, called **bias coefficient**, associated to a virtual input $x_0 = -1$ that will play a particular role in the sequel. Each neuron does not deal with all the x_i's but only with the weighted mean,

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

The goal of a neuron is to give an answer y in $[0, 1]$ from $\bar{x}$. To that purpose, we use a function $g : \mathbb{R} \to [0, 1]$, called **activation function**. When the output is close to 1, we say that the neuron is **active** and if the output is close to 0, we say that the neuron is **inactive**. Thus, the answer given by the neuron is defined as

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

Let us notice that if g is a linear function of the inputs, this model is equivalent to the linear regression. Of course, neural networks become more interesting when g is not linear. According to the definition, any function with values in $[0, 1]$ can be used as an activation function. Nevertheless, two particular functions are often used in practice :

– Heaviside step function,

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

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

– Sigmoid function,

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

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

These two functions are nondecreasing and the advantage of the sigmoid is to be differentiable on $\mathbb{R}$. These two functions switch from inactivity to an active state when $\bar{x}$ becomes larger than 0. Thus, the role of the bias coefficient is to set where this happens around w_0,

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

Using vectors, we can write $x = (x_1, \ldots, x_n)' \in \mathbb{R}^n$, $w = (w_1, \ldots, w_n)' \in \mathbb{R}^n$ and define $\bar{x}$ as the scalar product of these vectors,

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

The frontier between active and inactive states is then given by the set

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

If $w \neq 0$, the space $\mathcal{X}_w$ is a hyperplane that splits $\mathbb{R}^n$ into two parts according to $w \cdot x$ is greater or smaller than w_0. So, the principle of a neuron is to split the space of the variables into two parts : one where the neuron is active, the other where the neuron is inactive. More generally, a neural network leads to several splits of the space that we will use to produce some puzzle. On each piece of space, the network will take a particular state that we can associate, for example, to a given cluster in a supervised learning problem.

2.2 First examples

To illustrate the introduced ideas, we take the basic case of $n = 2$ binary variables (*i.e.* with values in $\{0, 1\}$) and the Heaviside step function as activation function. This case is very simple because the pair of input variables (x_1, x_2) can only take 4 distinct values which can be summarized in a **truth table** with the value taken by the output variable y in the last column,

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

Being given the values taken by y, the arising question is to find a neural network compatible with these data.

First, we are going to study a toy example where only one neuron is enough : the binary operator OR related to the following table,

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

Because there are only two variables, we can draw these data in the plane with distinct colors, red for 0 and blue for 1,

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

With the help of the function `abline`, find and plot a straight line that splits these points according to their colors. Doing a parallel with the previous section, deduce three real numbers w_0, w_1 and w_2 such that y is equal to 1 if and only if $w_1 x_1 + w_2 x_2 > w_0$. In particular, verify that $w_0 = 1/2$ and $w_1 = w_2 = 1$ do the job. The network (of one neuron) that we have just described if sometimes represented as in Figure 1.

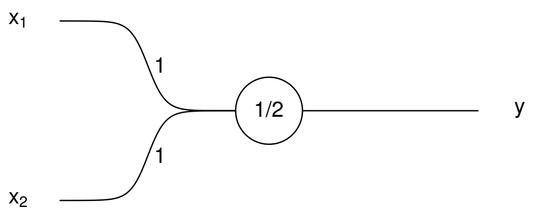

FIGURE 1 – Operator OR neuron

You can do the same exercise for the operator AND given by the following truth table,

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

which you can plot with the command

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

In particular, find some values w_0, w_1 and w_2 appropriated to this neuron.

Consider now the case of the operator XOR given by

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

and plotted with

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

What is the problem here ? This problem cannot be solved with only one neuron and we will be back to that later.

3 Single layer perceptron

3.1 Neural network

The first neural network that we consider is known as the single layer perceptron. The neurons are not really organized as in a network but can be seen as a set. As in Section 2.1, we deal with n input variables $x_1, \ldots, x_n \in \mathbb{R}$. The single layer perceptron is composed of p neurons, each one being connected to all the input variables. On the whole, this network has n inputs and p outputs and it can be represented as in Figure 2.

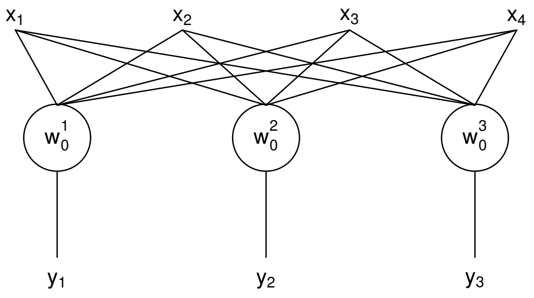

FIGURE 2 – Single layer perceptron with $n = 4$ inputsand $p = 3$ outputs

We denote the input vector by $x = (x_1, \ldots, x_n)' \in \mathbb{R}^n$, the output vector by $y = (y_1, \ldots, y_p)' \in [0,1]^p$ and, for any $j \in \{1, \ldots, p\}$, $w_0^j \in \mathbb{R}$ is the bias coefficient of the j-th neuron. Moreover, for any $i \in \{1, \ldots, n\}$ and any $j \in \{1, \ldots, p\}$, $w_i^j \in \mathbb{R}$ is the weight associated to x_i by the j-th neuron. Then, we define the $n \times p$ weight matrix $W = (w_i^j)_{i,j}$.

3.2 Supervised learning

The neural network of the single layer perceptron allows us to build a classification procedure into p clusters by considering each neuron as a score. The cluster returned for some input vector is the one associated to the neuron giving the maximal output. Building such a procedure amounts to compute the bias coefficients $w_0^1, \ldots, w_0^p \in \mathbb{R}$ and a weight matrix W. As we did for the other supervised learning methods, we are going to handle this one from a labeled dataset by considering the weights and the bias coefficients that lead to some minimal criterion (least squares, entropy, ...).

We do not develop here the optimization algorithms used in practice for getting W and the w_0^j's. Classical approaches are based on gradient descent techniques. Among them, the algorithm of Widrow-Hoff (*a.k.a. delta rule* or *back propagation*) is one of the most popular. The interested reader will find explanations, proofs and implementations in the references given at the end of this document.

To put into practice the single layer perceptron, we use the package `neuralnet` (if needed, install it with `install.packages("neuralnet")`). This package implements, among other things, the Widrow-Hoffmethod and its main function is `neuralnet`,

```
library(neuralnet)

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

help(neuralnet)
```

Let us apply that to a fertility dataset about 248 women that comes from the work of Trichopoulos *et al.* (1976). These data are reachable in the R software with `infert`. To get more details, read the fine manual,

```
help(infert)
```

The function `neuralnet` is similar to `lm`. The three main parameters to give are :

`formula` to indicate what is the variable to explain and what are the explanatory variables. Its syntax is the same as in the function `lm` and additinal informations can be found in the manual pages about R object *formula*. We simply deal with the variable `infert_form` defined by

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

`data` the dataset that contains the variables used in the formula. Here, this is `infert`.

`hidden` this parameter will be explained in the next section and, for the single layer perceptron, it suffices to set it to `0`.

Let's go for a first try with the following command,

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

There are various ways for getting details about the obtained perceptron. For example,

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

With the help about objects defined in `neuralnet`, explain what does represent the value `Steps` ? What is the error measurement ? Where can we get the weights ? *et cætera* . . .

Of course, we can get a picture of this perceptron,

```
plot(mono)
```

Explain what you see on this graph. Moreover, if you have at your disposal some new inputs, you can apply the neural network to them with the function `compute`,

```
help(compute)
```

4 Multilayer perceptron

It is possible to generalize the perceptron by piling up several single layer perceptrons. In such a procedure, the outputs of one layer are the inputs of the next one. This neural network is, of course, more complicated but can be used in the same way as the single layer perceptron. The main benefit is the ability to approximate nonlinear behaviours and to get smaller errors on the training dataset (better adequation to the data) at the cost of an higher complexity (does it remind you something ?).

For a two layers perceptron with n inputs and p outputs, we have to choose how many neurons we put on the transitional layer. In practice, this is often better to have too much hidden neurons than not enough if we plan to capture nonlinear phenomena. However, if we use too much neurons, we take the risk of an overfitting behaviour on our training dataset. In practice, we can tackle this problem through cross validation. The parameter of `neuralnet` that sets the number of hidden neurons is `hidden`. To compute the two layers network with 2 hidden neurons,

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

Compare the error with the one obtained in the single layer case. Plot this neural network and modify the parameters to understand their roles (especially `algorithm`).

To conclude, we propose to go back to the toy problem of the binary operator XOR seen in Section 2.2. We create a simple dataset `xor_data` and a first perceptron with what follows,

```r
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)
```

Vary the parameter `hidden` and study the perceptrons that you get from the point of view of the smallest error. Repeat the computation (see the parameter `rep` in `neuralnet`) to describe a "good" network structure for XOR. Introduce a new input variable `x3 <- c(0,0,0,1)`. What does represent `x3` ? Compute again the perceptrons and find a neural network for the operator 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