1 Objectif

Découverte des librairies de Deep Learning Tensorflow / Keras pour R. Implémentation de perceptrons simples et multicouches.

Python et R sont les deux mamelles généreuses de la fertilité intellectuelle du data scientist. Parfois elles sont interchangeables, parfois elles se complètent. En tous les cas, elles nourrissent la pratique de la data science. Et, finalement, le choix entre ces fontaines de jouvence est avant tout affaire de goûts personnels, de circonstances, d'environnements de travail, de disponibilité des packages…

Ce tutoriel fait suite à un document récent (« Deep Learning avec Tensorflow et Keras (Python) », Avril 2018) consacré au deep learning via les librairies Tensorflow et Keras sous Python. Nous en reprenons les étapes point par point, mais sous R cette fois-ci. Nous verrons que la transposition est particulièrement simple.

2 Installation des librairies sous R

Keras a été développé initialement en Python pour le langage Python. Plusieurs portages pour R existent. Dans ce tutoriel, nous utiliserons le package « keras » développé par RStudio (https://keras.rstudio.com/). L'installation, à réaliser une seule fois, consiste à saisir dans la console la succession de commandes suivantes :

```
#installation du package à partir du dépôt CRAN
install.packages(''keras'')

#chargement de la librairie
library(keras)

#installation de l'environnement
install_keras()
```

Cette dernière instruction se charge d'installer à la fois le cœur de la librairie Keras, mais aussi le moteur (backend engine) Tensorflow, indispensable puisque c'est lui en réalité qui accomplit les calculs. Keras, rappelons-le, n'est qu'un front-end (facile à utiliser certes, d'où son intérêt) qui permet d'accéder aux fonctionnalités des librairies sous-jacentes, dont Tensorflow fait partie.

3 Perceptron simple et multicouche

3.1 Données

Nous avions utilisé ce jeu de données précédemment (RAK, 2013 ; section 2). Il s'agit d'un problème de discrimination binaire dans le plan. La frontière séparant les classes prend la forme d'une parabole (Figure 1) :

$$\text{Si } (0.1 * X_2 > {X_1}^2) \text{ Alors } (Y = \text{positif}) \text{ Sinon } (Y = \text{négatif})$$

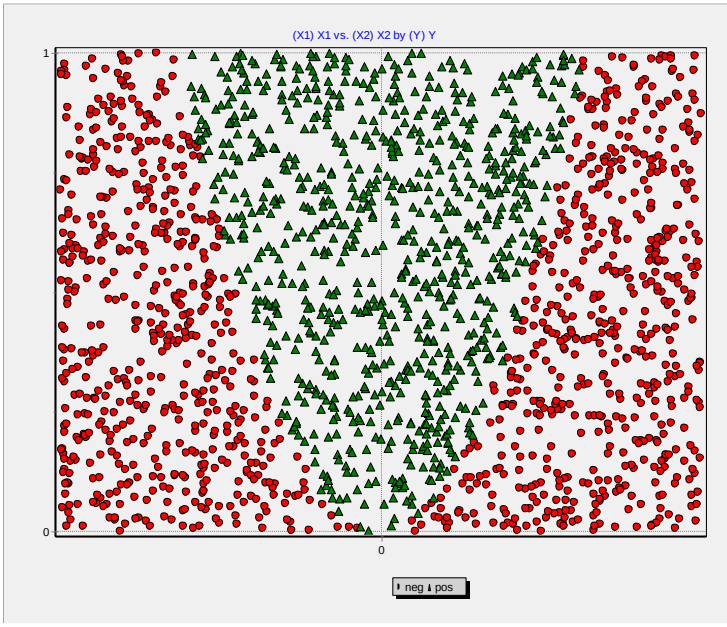

Figure 1 - Problème de discrimination binaire

3.2 Préparation des données

Importation des données. Nous importons le fichier « **artificial2d_data2.txt** » à l'aide de la fonction read_table() :

```
#modification du dossier par défaut
setwd("… votre répertoire …")

#chargement des données
D <- read.table("artificial2d_data2.txt",sep="\t",header=T,dec=".")

#vérification
print(str(D))
```

Nous avons un data frame avec 2000 observations et 3 variables.

```
'data.frame':       2000 obs. of  3 variables:
 $ X1: num  -0.355 0.464 0.001 0.427 -0.391 -0.425 0.467 0.295 -0.392 -0.405 ...
```

```
 $ X2: num  0.676 0.681 0.294 0.592 0.823 0.305 0.948 0.266 0.336 0.098 ...
 $ Y : Factor w/ 2 levels "neg","pos": 1 1 2 1 1 1 1 1 1 1 ...
```

Recodage de la variable cible. Notre variable Y prend ses valeurs dans {pos, neg}…

```
#fréquence absolue des classes
print(table(D$Y))
```

… avec respectivement 859 et 1141 observations.

```
neg  pos
1141  859
```

Nous devons la recoder en {1, 0} avant de pouvoir l'utiliser.

Nous créons la variable y à cet effet avec le très pratique ifelse() :

```
#recodage
y <- ifelse(D$Y=="pos",1,0)

#vérification
print(sum(y))
```

La somme des valeurs est égale à 859. Elle correspond au nombre d'observations positives (Y = 'pos') de notre jeu de données.

Subdivision en échantillons d'apprentissage et de test. Nous partitionnons les données en 1500 observations pour l'apprentissage, 500 pour le test.

```
#index désignant les individus en train
set.seed(100)
idTrain <- sample(1:nrow(D),1500,replace=F)

#X et y pour l'apprentissage
XTrain <- as.matrix(D[idTrain,1:2])
yTrain <- y[idTrain]

#X et y en test (indiçage négatif)
XTest <- as.matrix(D[-idTrain,1:2])
yTest <- y[-idTrain]
```

sample() permet d'obtenir par tirage aléatoire les indices idTrain des individus à inclure dans l'échantillon d'apprentissage ; l'indiçage négatif permet de désigner les observations « autres que 'train' » à inclure dans l'échantillon test.

A ce stade, nous sommes parés pour lancer le processus d'apprentissage supervisé.

3.3 Perceptron simple

Architecture du réseau. Nous chargeons la librairie « keras ». Nous créons une structure de réseau, vide pour l'instant, avec l'instruction keras_model_sequential()

```
#chargement de la librairie
library(keras)

#structure de réseau
modelSimple <- keras_model_sequential()
```

Avec l'instruction layer_dense(), nous ajoutons dans cette structure une couche « dense » faisant la jonction entre les neurones des couches d'entrée et de sortie du réseau.

```
#couche reliant l'entrée et la sortie
modelSimple %>%
  layer_dense(units=1,input_shape=c(2),activation="sigmoid")
```

Nous spécifions dans la couche : son nombre de neurones (units = 1, une seule sortie puisque la variable cible est binaire, codée 1/0), nombre de neurones de l'entrée (input_shape, nombre de neurones = nombre de variables prédictives), la fonction d'activation sigmoïde (activation).

Remarque : L'opérateur %>% est une pure coquetterie d'écriture qui devient de plus en plus en populaire dans les packages. Il signifie : l'élément à gauche de l'opérateur est le premier paramètre de l'instruction située à sa droite. Dans notre cas, nous aurions pu écrire l'instruction sous la forme suivante :

```
#autre écriture - ajout d'une couche dans le réseau
layer_dense(modelSimple,units=1,input_shape=c(2),activation="sigmoid")
```

Voyons la configuration de notre modèle à ce stade :

```
#configuration du modèle
print(get_config(modelSimple))
```

Plusieurs informations apparaissent…

```
[{'class_name': 'Dense', 'config': {'name': 'dense_3', 'trainable': True,
'batch_input_shape': (None, 2), 'dtype': 'float32', 'units': 1, 'activation':
'sigmoid', 'use_bias': True, 'kernel_initializer': {'class_name':
'VarianceScaling', 'config': {'scale': 1.0, 'mode': 'fan_avg', 'distribution':
'uniform', 'seed': None}}, 'bias_initializer': {'class_name': 'Zeros', 'config':
{}}, 'kernel_regularizer': None, 'bias_regularizer': None, 'activity_regularizer':
None, 'kernel_constraint': None, 'bias_constraint': None}}]
```

… nous retiendrons surtout que notre perceptron est pourvu du biais (ou intercept) avec l'option « use_bias : True » c.-à-d. un neurone qui prend systématiquement la valeur 1, ce qui évite à l'hyperplan séparateur de passer nécessairement par l'origine.

L'instruction summary() propose une autre lecture de l'architecture du réseau :

```
#autre vision de la configuration
print(summary(modelSimple))
```

Il y a 3 paramètres (coefficients, poids synaptiques) à estimer :

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

En effet, visuellement, notre réseau ressemble à ceci (Figure 2) :

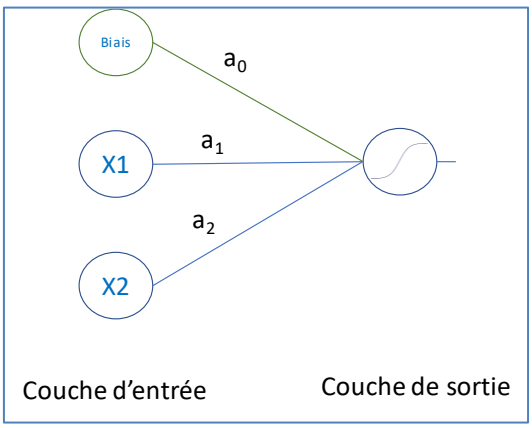

Figure 2 - Architecture de notre perceptron simple

En entrée du neurone de la couche de sortie, nous avons la combinaison linéaire :

$$d(X) = a_0 + a_1 X_1 + a_2 X_2$$

Avec l'application de la fonction d'activation sigmoïde, nous avons en sortie du neurone de la couche de sortie

$$g(d) = \frac{1}{1 + e^{-d}}$$

g(d) est une estimation de la probabilité conditionnelle P(Y = pos / X_1, X_2), déterminante dans les problèmes de classement.

Algorithme d'apprentissage. L'étape suivante consiste à « compiler » les caractéristiques de l'algorithme d'apprentissage : la fonction de perte à optimiser (loss) est l'entropie croisée binaire, elle correspond à la log-vraisemblance d'un échantillon où la probabilité conditionnelle d'appartenance aux classes est modélisée à l'aide de la loi binomiale (voir R.R., « Pratique de la régression logistique », section 1.4) ; Adam est l'algorithme d'optimisation (optimizer), elle est une alternative efficace à la descente du gradient stochastique ; la métrique (metrics) utilisée pour mesurer la qualité du modèle est le taux de reconnaissance ou taux de succès (accuracy en anglais, à ne pas confondre avec la précision qui est un terme technique dont la formule est différente).

```
#algorithme d'apprentissage
modelSimple %>% compile(
  loss="binary_crossentropy",
  optimizer="adam",
  metrics="accuracy"
)
```

Estimation des paramètres du réseau sur l'échantillon d'apprentissage. A l'aide de la commande fit(), nous pouvons lancer l'estimation des poids synaptiques (coefficients) du réseau à partir des données étiquetées.

```
#processus d'apprentissage
modelSimple %>% fit(
  x=XTrain,
  y=yTrain,
  epochs=150,
  batch_size=10
)
```

epochs est le nombre maximum d'itérations ; batch_size correspond aux nombre d'observations que l'on fait passer avant de remettre à jour les poids synaptiques.

L'évolution de l'apprentissage est affichée dans la console R. En ce qui me concerne, voici les valeurs finales de loss = 0.6429 et accuracy = 0.6220. La similitude des valeurs est une coïncidence pour notre exemple.

De plus, R affiche dans sa sortie graphique les courbes d'évolutions de la fonction de perte et du taux de reconnaissance (Figure 3).

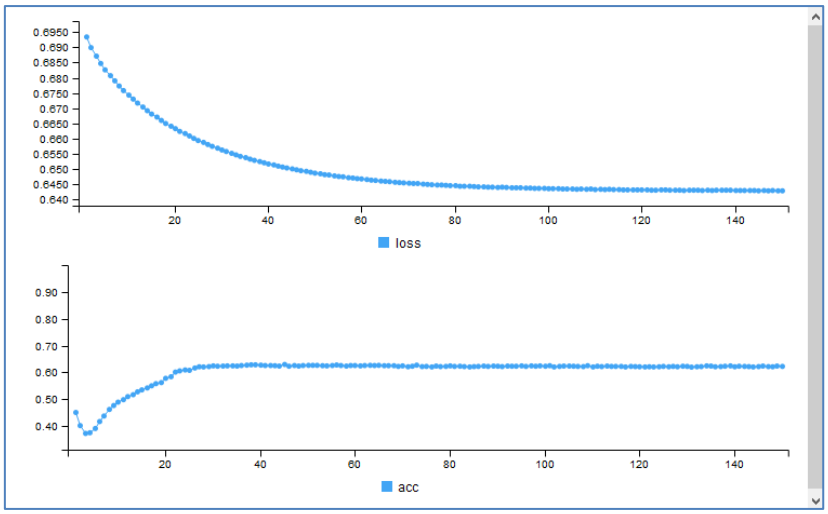

Figure 3 - Evolution de la perte et de l'accuracy durant l'apprentissage

Une fois l'apprentissage finalisée, nous pouvons afficher les poids estimés :

```
#poids synaptiques estimés
get_weights(modelSimple)
```

Nous obtenons :

```
[[1]]
     [,1]
[1,] 0.2562927
[2,] 1.9467665

[[2]]
[1] -1.283107
```

Plus sibyllin, c'est difficile. Nous avons une liste composée d'une matrice et d'un vecteur. L'enjeu est de pouvoir les replacer dans notre structure (Figure 2) : **[[1]]** est une matrice (2, 1), il s'agit des poids reliant les 2 neurones de la couche d'entrée à la sortie avec 1 seul neurone ; **[[2]]** correspond au biais (Figure 4).

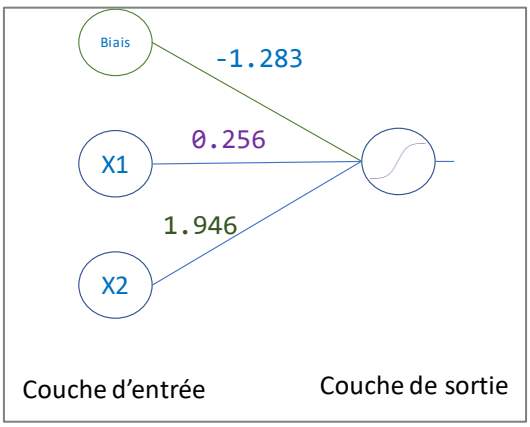

Figure 4 - Perceptron simple - Poids synaptiques estimés

3.4 Evaluation du modèle

Prédiction et confrontation. L'approche usuelle d'évaluation consiste à réaliser la prédiction sur l'échantillon test, puis à la confronter avec les valeurs observées de la variable cible. Voici le code pour la première étape :

```
#prédiction sur l'échantillon test
predSimple <- modelSimple %>% predict_classes(XTest)
print(table(predSimple))
```

Nous avons 335 prédictions « neg » et 165 « pos ».

```
predSimple
  0   1
335 165
```

Remarque : Il est possible d'obtenir les scores d'appartenance, pour le scoring ou encore pour l'élaboration de la courbe ROC, avec la commande predict_proba().

Nous utilisons l'instruction confusionMatrix() de la librairie « **caret** » (qu'il faut installer au préalable si nécessaire – voir « Machine Learning avec caret », avril 2018) pour calculer la matrice de confusion :

```
#matrice de confusion
library(caret)
print(caret::confusionMatrix(data=factor(predSimple),reference=factor(yTest),positive="1"))
```

Le taux de reconnaissance est de **0.678** ; « caret » fournit des informations supplémentaires qui permettent de mieux juger le comportement du modèle (sensibilité, précision, etc.).

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 229 106
         1  55 110

               Accuracy : 0.678
                 95% CI : (0.6351, 0.7188)
    No Information Rate : 0.568
    P-Value [Acc > NIR] : 3.022e-07

                  Kappa : 0.3248
 Mcnemar's Test P-Value : 8.129e-05

            Sensitivity : 0.5093
            Specificity : 0.8063
         Pos Pred Value : 0.6667
```

```
         Neg Pred Value : 0.6836
             Prevalence : 0.4320
         Detection Rate : 0.2200
   Detection Prevalence : 0.3300
      Balanced Accuracy : 0.6578

       'Positive' Class : 1
```

Fonction d'évaluation. L'autre solution consiste à utiliser l'outil evaluate() de la librairie Keras.

```
#fonction d'évaluation de Keras
res <- modelSimple %>% evaluate(
  x = XTest,
  y = yTest
)

#affichage
print(res)
```

Il fournit la perte (loss) et le taux de reconnaissance.

```
$loss
[1] 0.6205502

$acc
[1] 0.678
```

3.5 Perceptron multicouche

Dans cette section, nous passons à un perceptron multicouche. Nous créons toujours une structure Sequential, dans lequel nous ajoutons successivement deux objets Dense : le premier fait la jonction entre la couche d'entrée (d'où l'option input_dim indiquant le nombre de variables prédictives) et la couche cachée qui comporte (units = 3) neurones ; le second entre cette couche cachée et la sortie à un seul neurone (units = 1). Nous avons une fonction d'activation sigmoïde dans les deux cas.

```
#structure pour perceptron multicouche
modelMc <- keras_model_sequential()

#couches du perceptron
modelMc %>%
  layer_dense(units=3,input_shape=c(2),activation="sigmoid") %>%
  layer_dense(units=1,activation="sigmoid")
```

Notre perceptron prend cette forme (Figure 5) :

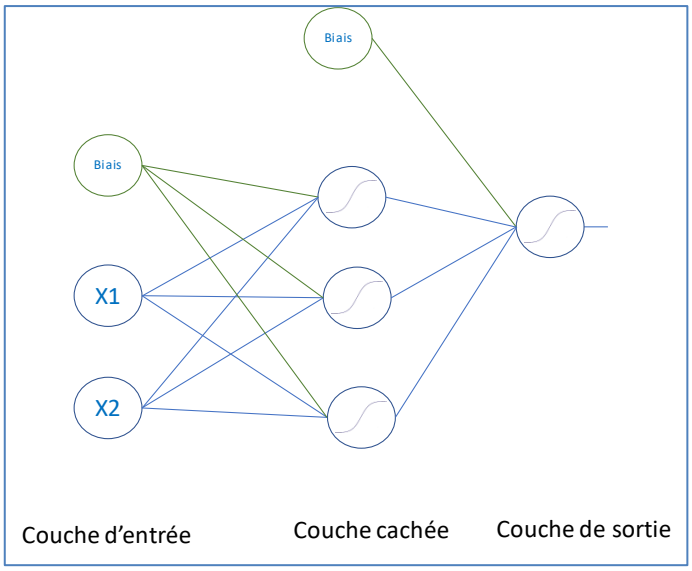

Figure 5 - Perceptron multicouche - Structure

La commande summary() nous le confirme…

```
#résumés
print(summary(modelMc))
```

…avec

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 3)                 9
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 4
=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________
```

Il y a 9 poids à estimer entre la couche d'entrée et la couche cachée [(2 x 3) pour les neurones + (1 x 3) pour le biais] ; et 4 entre la couche cachée et la couche de sortie [(3 x 1) pour les neurones + (1 x 1) pour le biais].

Les étapes suivantes sont usuelles.

```
#algorithme d'apprentissage
modelMc %>% compile(
  loss="binary_crossentropy",
  optimizer="adam",
  metrics="accuracy"
)
```

```
#processus d'apprentissage
modelMc %>% fit(
  x=XTrain,
  y=yTrain,
  epochs=150,
  batch_size=10
)

#poids synaptiques estimés
get_weights(modelMc)
```

L'évolution de la perte et du taux de reconnaissance sont autrement plus favorables avec ces nouveaux paramétrages.

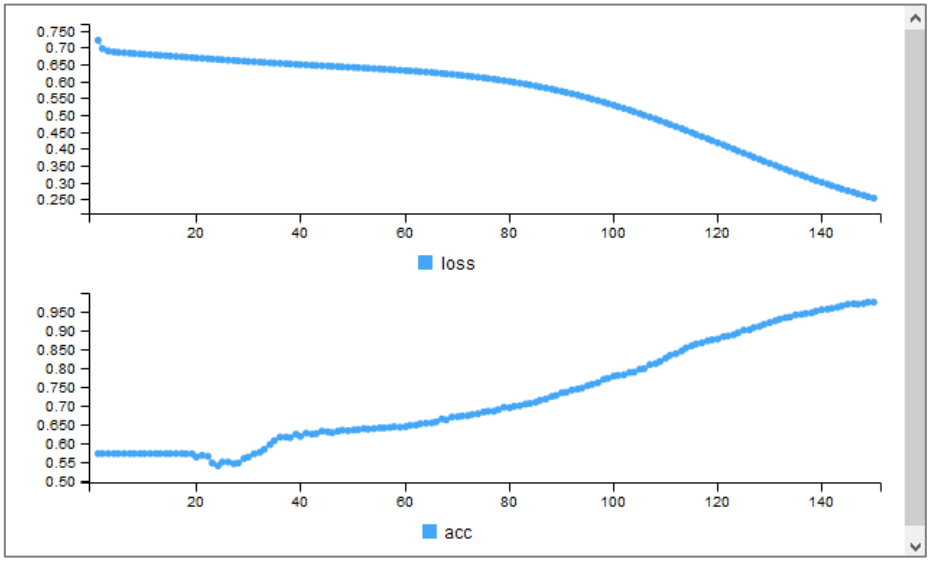

Figure 6 - Perceptron multicouche - Evolution de la perte et du taux de reconnaissance

Les poids sont en adéquation avec la structure :

```
> get_weights(modelMc)
[[1]]
          [,1]      [,2]      [,3]
[1,]  9.143174 -6.088501 11.914215
[2,] -1.953827 -1.981644  2.461681

[[2]]
[1] -1.1278218 -0.6482403  1.4128209

[[3]]
          [,1]
[1,] -6.335299
[2,] -3.202064
[3,]  3.867735

[[4]]
[1] -0.6016442
```

Nous avons **[[1]]** une première matrice 2 lignes (parce que 2 neurones dans la couche d'entrée) et 3 colonnes (parce que 3 neurones dans la couche intermédiaire) ; **[[2]]** un vecteur avec 3 valeurs (lien entre le biais de la couche d'entrée et les 3 neurones de la couche cachée) ; **[[3]]** une matrice avec 3 lignes (3 neurones de la couche cachée) et 1 colonne (1 neurone de la couche de sortie) ; **[[4]]** et enfin un vecteur avec une valeur (lien entre le biais et la couche de sortie) (Figure 7).

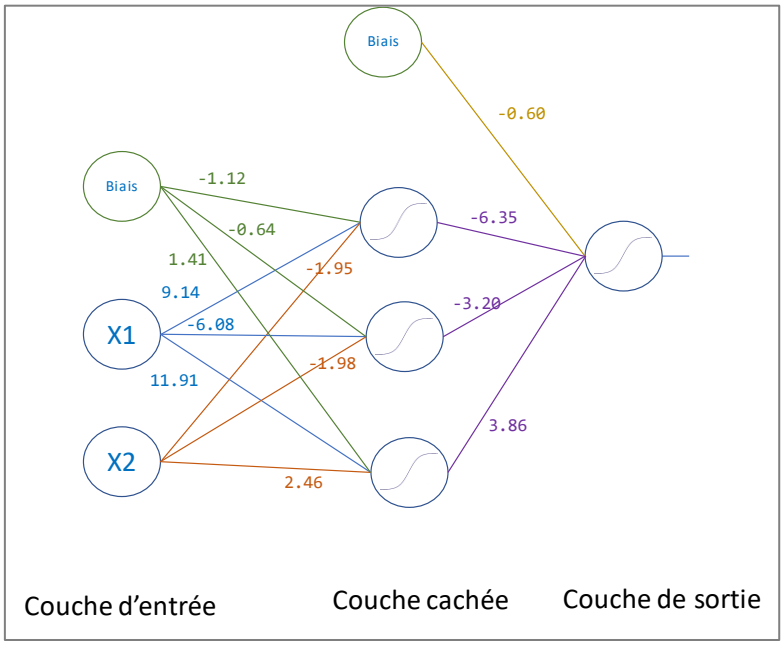

Figure 7 - Perceptron multicouche - Poids synaptiques estimés

Les performances en test…

```
#fonction d'évaluation de Keras
res <- modelMc %>% evaluate(
  x = XTest,
  y = yTest
)

#affichage
print(res)
```

… correspondent à un taux de reconnaissance de **97.4%**.

4 Conclusion

Je suis dans une phase où je m'intéresse de plus en plus activement aux méthodes de deep learning. La question des outils se pose forcément puisqu'un cours doit s'appuyer

sur des exemples illustratifs, compréhensibles, faciles à mettre en œuvre, reproductibles. Les librairies Tensorflow et Keras semblent incontournables au regard de la littérature accessible en ligne. Après avoir exploré leur usage sous Python, nous avons étudié ce qu'il en était sous R dans ce tutoriel. On se rend compte que (1) leur utilisation (l'accès aux méthodes de Tensorflow via Keras devrait-on dire) est relativement simple, (2) les modes opératoires sont très similaires sous R et Python, passer d'un langage à l'autre n'est absolument pas un problème.

5 Références

- Keras : The Python Deep Learning Library (https://keras.io/).
- R Interface to Keras - https://keras.rstudio.com/
- CRAN : https://cran.r-project.org/web/packages/keras/index.html
- Tensorflow : An Open-Source Machine Learning Framework for Everyone (https://www.tensorflow.org/).
- (RAK, 2013) Ricco Rakotomalala, « Paramétrer le perceptron multicouche », avril 2013.
- Tutoriel Tanagra, « Deep Learning avec Tensorflow et Keras (Python) », avril 2018.
- Karlijn Willems, « keras : Deep Learning in R », June 2017.