TP : début de la classification d'images

ENSTA

2019/2020

Adresse mail: mohammed.sedki@universite-paris-saclay.fr Page web: masedki.github.io

---

1.Le jeu de données MNIST

Le jeu de données MNIST est hébergé sur le page web de Yann LeCun. Ce jeu de données sert de référence pour les compétitions en apprentissage où la donnée explicative est sous forme d'image. Commençons par une visualisation du jeu de données et plus précisément les 36 premières images. Pour cela nous allons installer le package keras qui installe à son tour tensorflow. Le bloc de code suivant permet cette visualisation

```
require(keras)
require(doParallel)
require(caret)

mnist   <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test  <- mnist$test$x
y_test  <- mnist$test$y
# visualize the digits
par(mfcol=c(6,6))
par(mar=c(0, 0, 3, 0), xaxs="i", yaxs="i")
for (idx in 1:36) {
  im <- x_train[idx,,]
  im <- t(apply(im, 2, rev))
  image(1:28, 1:28, im, col=gray((0:255)/255),
        xaxt="n", main=paste(y_train[idx]))
}
```

2. Réduction de dimension par ACP

Afin de pouvoir mettre en place des méthodes d'apprentissage, nous avons besoin d'aplatir les images en vecteurs. Cela revient à transformer une image de dimension 28×28 pixels en un vecteur de dimension 784.

1. Préparer le jeu de données en appliquant un aplatissement[1] ainsi qu'une suppression des pixels nuls pour l'ensemble des images du jeu de données. La fonction nearZeroVar du package caret permet de repérer les variables de variance nulle.
2. Le nombre de variables du jeu de données après aplatissement est très élevé. Proposer une procédure de réduction de dimension indépendante de la méthode d'apprentissage à utiliser. La fonction preProcess du package caret permet un ensemble de transformations de données. Afficher la dimension avant et après suppression et réduction de dimension et faire en sorte à ce que la procédure garde les mêmes dimensions pour les deux jeux de données train et test.

3. Forêt aléatoire et gradient boosting

3. À l'aide du package caret, entraîner un modèle de forêt aléatoire avec les paramètres de contrôle suivants :

```
control <- trainControl(method="cv", number=2)
rf.grid <-  expand.grid(mtry = 10*(1:7))
```

Réduire le temps de calcul en limitant le nombre d'arbre à 50 et le nombre d'observations par classe à 500 durant l'apprentissage.

4. À l'aide du package caret, entraîner un modèle de gradient boosting avec les paramètres de contrôle suivants :

```
control <- trainControl(method="cv", number=2)
boost.grid = expand.grid(eta = 1,
                         nrounds = c(700, 750, 800, 850),
                         max_depth = 2,
                         subsample = 1,
                         min_child_weight = 1.,
                         colsample_bytree = 0.5,
                         gamma = 0.)
```

4. Bonus : apprentissage par réseaux de neurones

Les RNA sont au cœur de l'apprentissage profond (*Deep Learning*). Ils sont polyvalents, puissants et extensibles, ce qui les rend parfaitement adaptés aux tâches d'apprentissage automatique extrêmement complexes, comme la classification de millions d'images (par exemple, Google Images), la reconnaissance vocale (par exemple, Apple Siri), la recommandation de vidéos auprès de centaines de millions d'utilisateurs (par exemple, YouTube) ou l'apprentissage nécessaire pour battre le champion du monde du jeu de go en analysant des millions de parties antérieures puis en jouant contre soi-même (AlphaGo de DeepMind).

Nous proposons ici une introduction aux Réseaux de Neurones Artificiels, en commençant par une description rapide des toute première architecture de RNA. Nous présenterons ensuite les *perceptrons multicouches* (PMC).

[1] La fonction *array_reshape* permet aplatir des matrices.

4.1. Le perceptron

Le perceptron, inventé en 1959 par F. Rosenblatt, est l'architecture de RNA la plus simple. Il se fonde sur un neurone artificiel appelé unité linéaire à seuil (LTU, *Linear Threshold Unit*). Les entrées correspondent aux variables explicatives et la sortie correspond à la prédiction de la variable à expliquer. Par analogie aux neurones biologiques, la LTU est un modèle qui se caractérise par une fonction dite *d'activation* qu'on notera g et les poids des entrées. La sortie d'un neurone artificiel est donnée par

$$h(x_1, \ldots, x_p) = g\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = g(w_0 + \mathbf{w}'\mathbf{x}).$$

La fonction d'activation applique une transformation d'une combinaison linéaire des entrées pondérées par des poids $w_0, w_1, \ldots, w_p$ où le poids w_0 est appelé le biais du neurone. Le choix de la fonction d'activation g se fait selon la nature de la variable à expliquer y (binaire ou continue). Souvent on fait appel à une fonction d'activation correspondant à l'identité dans le cas d'un problème de régression ou la fonction d'activation softmax quand il s'agit d'un problème de classification[2]. Entraîner une LTU revient à choisir les valeurs des poids $w_0, w_1, \ldots, w_p$ appropriées par rétro propagation élémentaire du gradient[3].

4.2. Le perceptron multicouches

Les performances d'apprentissage d'un perceptron composé d'une seule LTU sont souvent médiocres. Il est souvent possible d'améliorer les performances des perceptrons en empilant plusieurs perceptrons. Le RNA résultant est appelé *perceptron multicouche* (PMC). Un PMC est constitué d'une couche d'entrée (qui ne fait rien, si ce n'est distribuer les entrées aux neurones de la couche suivante), d'une ou plusieurs couches successives de LTU appelées *couches cachées* et d'une dernière couche de LTU appelée *couche de sortie*. Une couche est un ensemble de LTU n'ayant pas de connexion entre elles. Selon les auteurs, la couche d'entrée qui n'introduit aucune modification n'est pas comptabilisée. Une ou plusieurs couches cachées participent au transfert. Dans un perceptron multicouche, une LTU d'une couche cachée est connectée en entrée à chacune des LTU de la couche précédente et en sortie à chaque LTU de la couche suivante. Lorsque un perceptron multicouche possède deux couches cachées ou plus, on parle de réseaux de neurones profond (RNP) au lieu de RNA (en anglais, on parle de *Deep Neural Network (DNN)*).

4.3. Implémentation d'un RNA avec une seule couche cachée avec keras

Reprenons le jeu de données après aplatissement mais sans réduction de dimension. Appliquons une normalisation en divisant les intensités des pixels par 255 (intensité maximale d'un pixel).

```
require(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

[2] Il est fortement recommandé de consulter le cours de Philippe Besse disponible ici pour plus de détails.
[3] Le cours de Philippe Besse détaille l'algorithme de rétro propagation élémentaire du gradient.

```
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
# reshape
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
# rescale
x_train <- x_train / 255
x_test <- x_test / 255
```

5. À l'aide du package keras, entraîner un RNA avec une couche cachée comme suit :
   a. Le nombre de neurones (LTU) de la couche cachée est de 784 avec la fonction d'activation relu[4]. De combien de LTU a-t-on besoin à la dernière couche ? préciser sa fonction d'activation.
   b. Utiliser la fonction de perte ainsi que la métrique appropriés pour le contexte de la classification. Faire appel à l'algorithme d'optimisation *adam*. Utiliser des mini-lots de données de taille 200 et 10 cycles d'apprentissage. Inclure dans la procédure le jeu de données test comme jeu de données de validation[5]. Préciser la meilleure performance obtenue en terme de taux de bon classement sur les 10 cycles d'apprentissage.

[4] Il faut absolument consulter l'aide du package keras pour comprendre le principe de fonctionnement et de construction des réseaux de neurones avec keras.

[5] La lecture de ce document: *Optimization algorithms* permet de se familiariser avec le vocabulaire.