Présentation de Keras

Kabirou Kanlanfeyi, Jordy Hounsinou

1. Présentation de Keras

Bibliothèque conçue pour faciliter l'implémentation des réseaux de neurones, Keras est open-source et a été écrite totalement en Python. Il est très utilisé pour des débuts avec les réseaux de neurones profonds compte tenu de sa facilité comparé à d'autres framework tels que TensorFlow, Théanos, Microsoft Cognitive ToolKit avec lesquels il nous permet d'interagir.

Initialement conçu pour une utilisation avec le langage Python, il est aussi accessible en installant tout simplement la bibliothèque *Keras* qui y est dédiée.

La commande est la suivante: *install.packages("keras")*

Il nous permet d'effectuer d'implémenter plusieurs architectures Deep Learning tels que les Réseaux de Neurones Convolutifs, les Réseaux de Neurones Récurrents et pleins d'autres.

Dans la partie suivante de notre documents, nous allons faire une illustration de ce framework en utilisant la base de données Fashion MNIST sur laquelle nous construirons un réseau de neurones simple avec plusieurs couches et d'autres paramètres.

Pour en savoir plus sur l'architecture d'un réseau de neurones, nous vous invitons à consulter les liens suivants:

https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels
https://medium.com/@davidroman0O/r%C3%A9seaux-neuraux-pour-les-nuls-a7f5f63b1c10

2. Implémentation de Keras sur la base de données Fashion MNIST

Nous tenons à préciser que le code ci-dessous est inspiré de la documentation officielle de Keras avec avec quelques modifications dans le but de notre documentation

Importation de la librairie et des données depuis le web

```
library(keras)

fashion_mnist <- dataset_fashion_mnist()
```

Nous allons créer deux matrices pour y stocker nos données déjà partitionnées

```
#Données d'entrainement
c(train_images, train_labels) %<-% fashion_mnist$train

#Données de test
c(test_images, test_labels) %<-% fashion_mnist$test
typeof(train_images)
```

```
## [1] "integer"
```

Prétraitement des données

```
library(tidyr)
library(ggplot2)

#Conversionn en DataFrame
image_1 <- as.data.frame(train_images[1, , ])
colnames(image_1) <- seq_len(ncol(image_1))
image_1$y <- seq_len(nrow(image_1))
image_1 <- gather(image_1, "x", "value", -y)
image_1$x <- as.integer(image_1$x)
```

Normalisation des données

Afin de normaliser les données, nous divisons nos valeurs par 255 afin d'éviter par exemple les valeurs abérantes. La valeur maximale étant 255 pour chacune des colonnes, nous faisons alors une division par 255 afin d'avoir des valeurs comprises entre 0 et 1

```
train_images <- train_images / 255
test_images <- test_images / 255
```

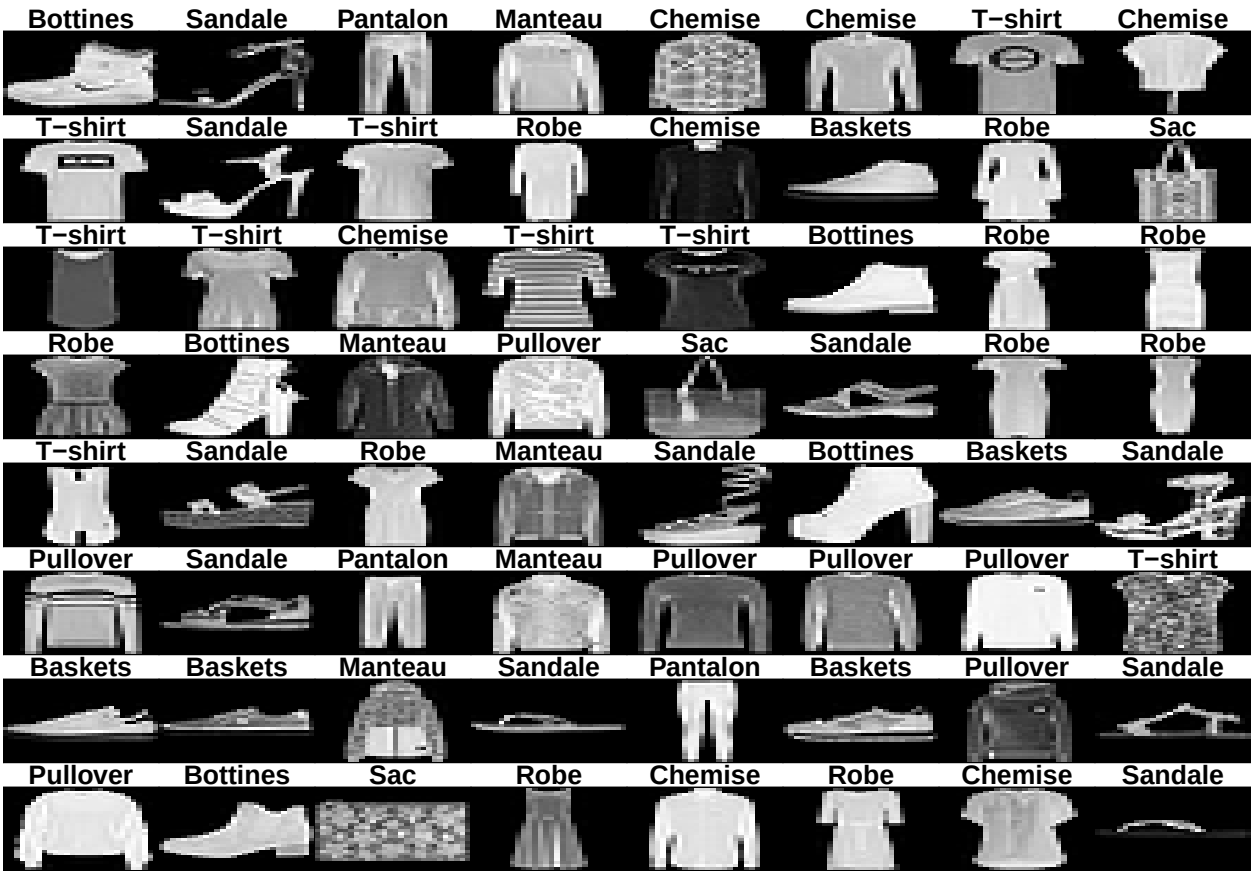
Nous allons dans cette partie, créer une liste de noms pour nos classes afin de mieux les identifier lors de l'affichage

```
noms_classes = c('T-shirt',
                 'Pantalon',
                 'Pullover',
                 'Robe',
                 'Manteau',
                 'Sandale',
```

```
                'Chemise',
                'Baskets',
                'Sac',
                'Bottines')
```

Nous afficherons ensuite la liste des 64 premières images des données d'entrainement

```
par(mfcol=c(8,8))
par(mar=c(0, 0, 1, 0), xaxs='i', yaxs='i')
for (i in 1:64) {
  img <- train_images[i, , ]
  img <- t(apply(img, 2, rev))
  image(1:28, 1:28, img, col = gray((0:255)/255), xaxt = 'n', yaxt = 'n',
        main = paste(noms_classes[train_labels[i] + 1]))
}
```



Création de notre réseau de neurones avec Keras

keras_model_sequential: Création d'un réseau de neurones vide

layer_flatten: très utilisé dans le traitement d'images, il s'agit de la première couche où l'on convertit notre matrice d'image en un vecteur.

layer_dense: cette fonction nous permet d'ajouter une couche qui prend en paramètre le nombre de neurones(paramètre *units*), la fonction d'activation. Pour les couches cachées la fonction d'activation *relu* est la plus utilisée. Pour la couche de sortie, *le nombre de neurones que l'on ajoute est à égal au nombre de classes* dans le cas d'une **classification** et la fonction d'activation utilisée dans ce cas est soit *Sigmoid* ou *Softmax*

layer_dropout: utilisé pour éviter les problèmes d'overfitting(surentrainement de données) pendants l'apprentissage, il permet d'ignorer certains neurones des neurones cachées choisis aléatoirement.

activation ou encore activation funcction: c'est une fonction mathétiue souvent non-linéaire qui est appliquée à un signal en sortie de neurone. Par exemple la fonction de **Heaviside** qui prend la valeur 0 pour toutes les valeurs négative et la valeurs 1 pour les valeurs positives.

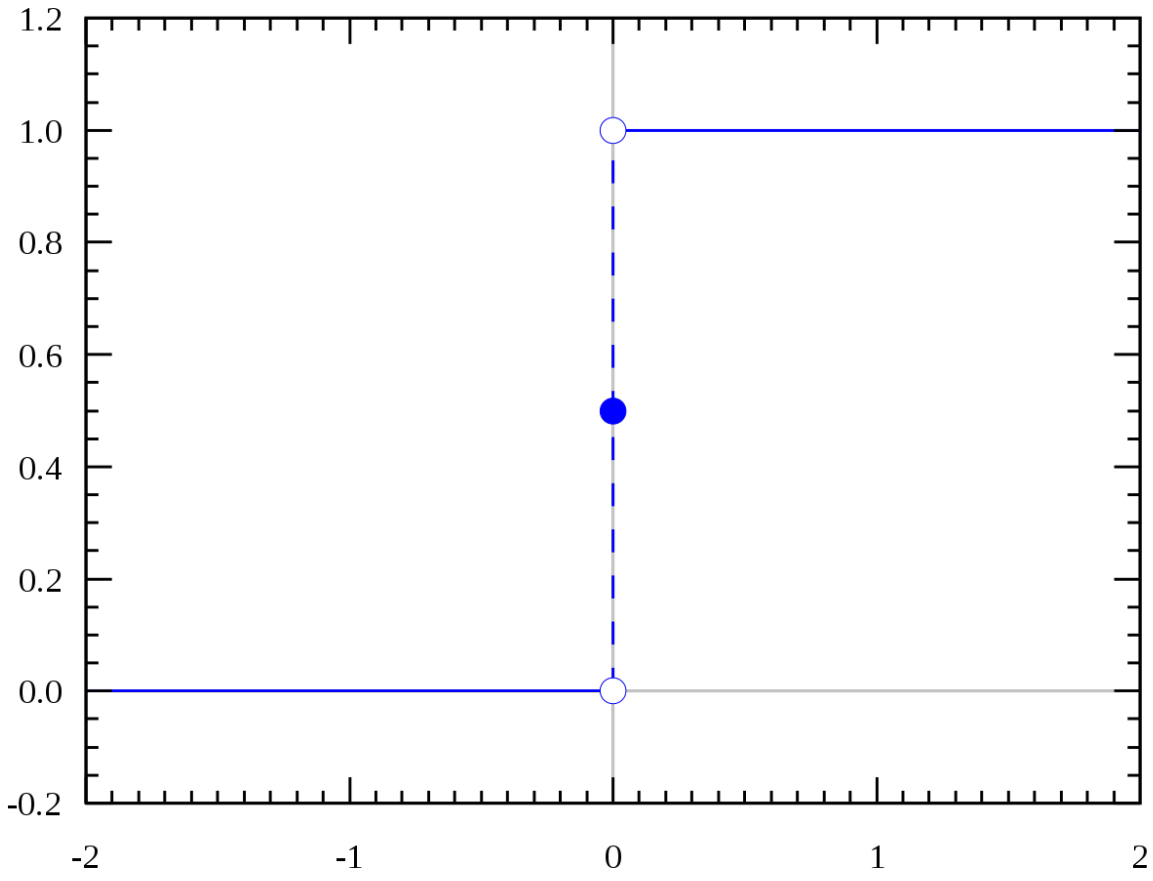


Figure 1: Fonction de Heaviside.

```r
model <- keras_model_sequential()
model %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.1)%>%
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dropout(rate = 0.1)%>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dropout(rate = 0.1)%>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## flatten (Flatten)                   (None, 784)                     0
## ________________________________________________________________________________
## dense (Dense)                       (None, 128)                     100480
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 128)                     0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 64)                      8256
## ________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 64)                      0
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 32)                      2080
## ________________________________________________________________________________
## dropout_2 (Dropout)                 (None, 32)                      0
## ________________________________________________________________________________
## dense_3 (Dense)                     (None, 10)                      330
## ================================================================================
## Total params: 111,146
## Trainable params: 111,146
## Non-trainable params: 0
## ________________________________________________________________________________
```

Phase de Compilation:

Une fois notre réseau de neurones crée avec tous les paramètres, il faut le compiler en lui précisant plusieurs paramètres très importants tels que:

optimizer: Les optimiseurs mettent à jour les paramètres de poids pour minimiser la fonction de perte

metrics: la mesure de performance du modèle de notre modèle (précision, rappel, Score F1) que l'on peut aussi calculer via la matrice de confusion.

```
model %>% compile(
  optimizer = 'adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)
```

Après la phase de compilation, nous arrivons à celle de l'apprentissage des données avec la fonction *fit()* qui prend principalement en paramètre:

les données d'entrainement: *train_images* dans notre cas **le nombre d'epochs**: nombre d'époques (itérations) pendant lequel nos données seront entrainées

```
model %>% fit(train_images, train_labels, epochs = 200, verbose = 2)
```

Test sur de nouvelles obesrvations

```
predictions <- model %>% predict_classes(test_images)
#Affichage des 50 premières données prédites
predictions[1:50]
```

```
##  [1] 9 2 1 1 6 1 4 6 5 7 4 5 5 3 4 1 2 2 8 0 2 5 7 5 1 4 6 0 9 3 8 8 3 3 8 0 7 5
## [39] 7 9 0 1 6 7 6 7 2 1 2 6
```

Matrice de confusion

Nous convertissons les résultats en facteur avec la fonction *factor()*

```
library(caret)
```

```
## Loading required package: lattice
```

```
test_labels2 = factor(test_labels)
predictions2 = factor(predictions)
matrice_Conf = confusionMatrix(predictions2,test_labels2)
matrice_Conf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 838   3  21  38   0   0 124   0   3   0
##          1   0 980   2   8   1   0   2   0   0   0
##          2  13   2 822  10  81   0  68   0   7   0
##          3  21   7   9 872  21   1  27   0   3   0
```

```
##          4   4   5  72  38 811   0  46   0   3   0
##          5   2   0   0   0   0 972   0  19   1   8
##          6 117   3  73  29  85   1 727   0   8   1
##          7   0   0   0   1   0  15   0 955   3  24
##          8   4   0   1   4   1   0   6   0 970   0
##          9   1   0   0   0   0  11   0  26   2 967
##
## Overall Statistics
##
##                Accuracy : 0.8914
##                  95% CI : (0.8851, 0.8974)
##     No Information Rate : 0.1
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8793
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.8380   0.9800   0.8220   0.8720   0.8110   0.9720
## Specificity            0.9790   0.9986   0.9799   0.9901   0.9813   0.9967
## Pos Pred Value         0.8160   0.9869   0.8195   0.9074   0.8284   0.9701
## Neg Pred Value         0.9819   0.9978   0.9802   0.9858   0.9790   0.9969
## Prevalence             0.1000   0.1000   0.1000   0.1000   0.1000   0.1000
## Detection Rate         0.0838   0.0980   0.0822   0.0872   0.0811   0.0972
## Detection Prevalence   0.1027   0.0993   0.1003   0.0961   0.0979   0.1002
## Balanced Accuracy      0.9085   0.9893   0.9009   0.9311   0.8962   0.9843
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.7270   0.9550   0.9700   0.9670
## Specificity            0.9648   0.9952   0.9982   0.9956
## Pos Pred Value         0.6964   0.9569   0.9838   0.9603
## Neg Pred Value         0.9695   0.9950   0.9967   0.9963
## Prevalence             0.1000   0.1000   0.1000   0.1000
## Detection Rate         0.0727   0.0955   0.0970   0.0967
## Detection Prevalence   0.1044   0.0998   0.0986   0.1007
## Balanced Accuracy      0.8459   0.9751   0.9841   0.9813
```

Références

https://fr.wikipedia.org/wiki/Keras

https://fr.wikipedia.org/wiki/Fonction_de_Heaviside

https://blog.rstudio.com/2017/09/05/keras-for-r/

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3