Introduction aux réseaux de neurones

Guillaume Metzler

2021-2022

A propos des réseaux de neurones

Les réseaux de neurones sont des outils puissants que l'on peut retrouver dans différents domaines d'applications du Machine Learning comme la génération de données comme des images, la vision assistée par ordinateur, l'analyse de textures de matériaux, la détection d'anomalies dans des clichés médicaux ou encore dans l'analyse de textes ou de graphes, les chatbots, ... bref les applications sont multiples.

Les structures de réseaux de neurones sont égalements multiples et on adaptera cette structure en fonction de la nature de la tâche :

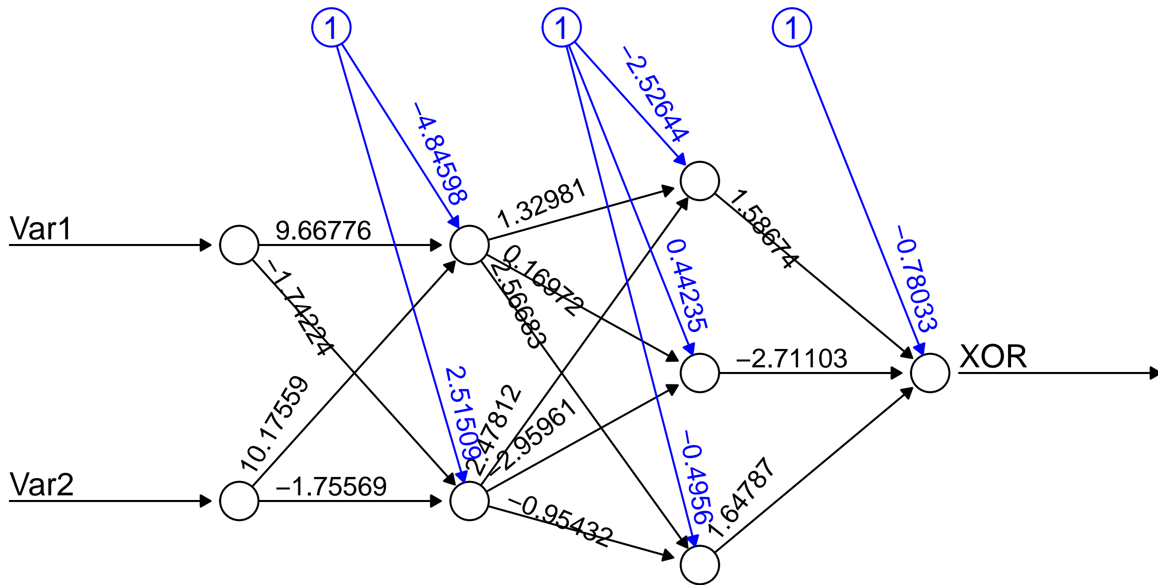
- des réseaux (deep) pour des tâches de classification (comme de la classification d'images quand les données sont peu complexes) ou encore de régression
- des réseaux convolutionnels pour de l'apprentissage de représentation servant à faire de la classification sémantique, de la vision assistée par ordinateur et donc de la reconnaissance de formes - de motifs. L'objectif sera d'extraire les variables importantes dans notre image (on va travailler avec la notion de gradient dans les images)
- les réseaux de neurones récurrents : utilisés notamment pour faire de la prédiction dans le temps, de l'analyse de textes ou encore dans des contextes économiques pour prévoir le cours d'une action au cours du temps (dans le futur). Il s'agit d'un type de réseau qui va stocker les informations du passé pour essayer de prédire l'avenir.
- les réseaux de neurones génératifs : permettre la synthèse d'images (Exemples sur ce site) par un système de deux modèles en oppostion. Un modèle génératif et un modèle de classification. Le premier a pour objectif de générer des images ayant pour but de tromper le modèle de classification afin de le rendre plus performant (il doit distinguer les images générées des vraies images). Cette performance du classifieur va aussi servir à améliorer les performances du générateur.

Vous retrouverez plus d'informations sur ces sujets dans les petites vidéos Youtube dont les liens sont données sur Moodle.

Matériel

Pour apprendre ce qu'est un réseau de neurones et afin de pouvoir les manipuler vous aurez besoins de différentes notions de calcul matriciel mais aussi de calcul différentiel et un peu d'optimisation. Je présente rapidement ci-dessous ces différentes notions sans entrer dans les détails afin de vous expliquer l'apprentissage d'un réseau de neurones. Vous trouverez à la fin de ce TP un lien vous proposant une implémentation "from scratch" afin de manipuler ces notions et de mieux les appréhender. Nous aborderons uniquement les réseaux de neurones classiques (deep ou multi-couches)

Généralités sur la structure d'un réseau Regardons déjà la structure d'un réseau de neurones. Ces derniers peuvent se représenter de la façon suivante :



Error: 0.000146 Steps: 98

On aura pris ici un exemple d'un jeu de données en deux dimensions pour lequel on cherche à prédire l'étiquette de la donnée (0 ou 1).

Regardons de plus prêt cette structure, on remarque que l'on a un réseaux avec trois couches :

- une première couche qui se compose de deux neurones en entrée : ce qui correspond à la dimension de notre jeu de données
- une couche cachée (hidden layer) : ici composée de deux neurones aussi
- une couche de sortie de taille une : elle va retourner l'étiquette prédite par le modèle pour la donnée considérée

On remarque également un ensemble de valeurs pour les différentes connexions entre les différents neurones (on dit d'ailleurs que ce réseau est "fully connected" car, d'une couche à l'autre, les neurones sont tous liés entre eux), ce sont les poids des des liens, ou le poids des différents neurones pour la prédiction associée à un nouveau neurones. Ces poids sont des paramètres que l'on va apprendre à l'aide de nos données.

1. D'après-vous, que représentent les connexions bleues ?
2. Sur la configuration représentée en image, combien de paramètres doit-on apprendre ?
3. Imaginez que l'on dispose d'un jeu de données de taille d et d'un réseau de neurones multi-couches dont le nombre de neurones à chaque couche est égale à h_1, h_2, en sortie, on dispose uniquement d'une valeur à prédire. Combien de paramètres doit-on apprendre ? Essayez de généraliser cela à un réseau multi-couches dont le nombre de neurones à chaque couche est égale à $h_1, h_2, ..., h_H$.
4. A la lumière de la question précédente, qu'est-ce qui caractérise les réseaux de neurones comparés aux autres modèles en Machine Learning

Outre la structure générale, plusieurs choses doivent également être définies pour pouvoir définir parfaitement notre réseau :

- la fonction de loss que l'on va employer pour apprendre notre modèle. Par exemple, pour effectuer une tâche de régression, on utilisera la MSE (Mean Square Errror) :

$$\ell(y, \hat{y}) = \|y - \hat{y}\|_2^2.$$

Dans des tâches de classification, on emploiera plutôt ce que l'on appelle la "cross-entropy" (en lien direct avec la divergence de Kullback-Leibler, ou encore KL-divergence). A noter que cette dernière est utilisée quand la valeur prédite est une probabilité !

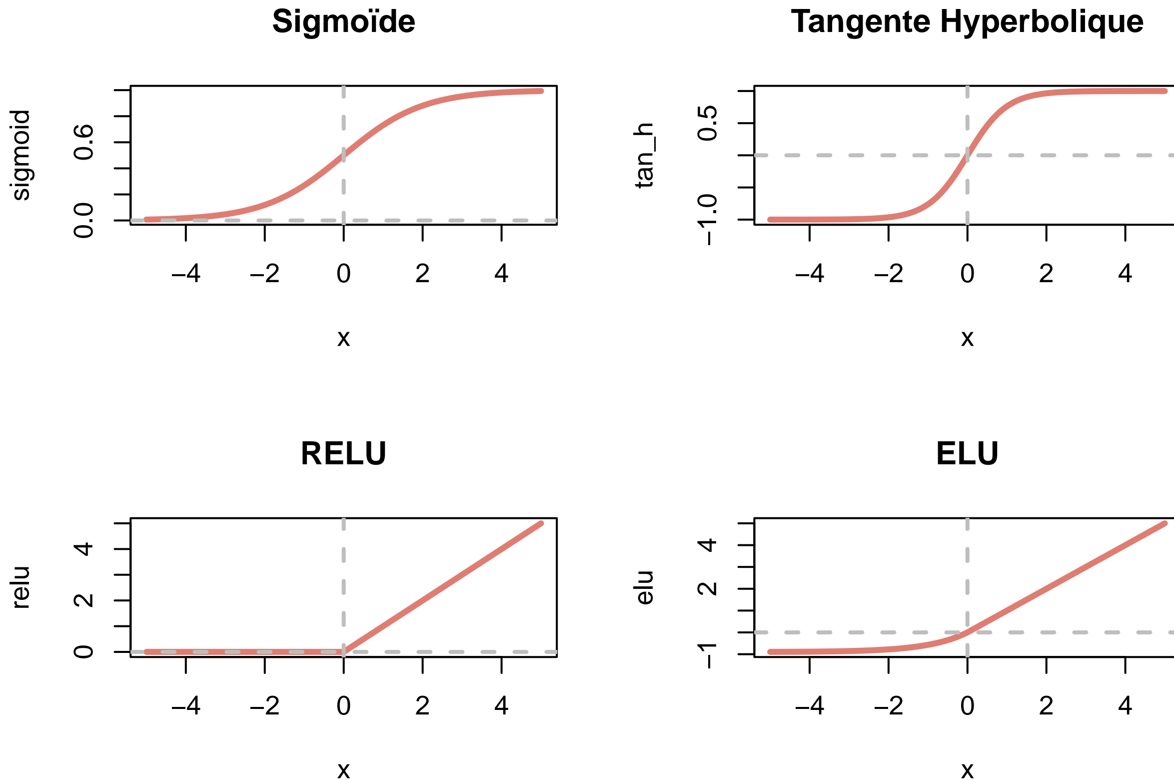
$$CE = -y_i \log(p_i) - (1 - y_i) \log(1 - p_i), \quad \text{dans le cas binaire}$$

ou encore, dans le cas multi-classe

$$CE = -\sum_{j=1}^{C} y_i^{(j)} log(p_{ij}),$$

où $y_i^{(j)} = 1$ si l'exemple i appartient à la classe j et 0 sinon. Enfin p_{ij} est la probabilité que l'exemple i appartienne à la classe j.

- un ensemble de fonctions d'activations pour permettre de déterminer les valeurs prises par les différents neurones d'une couche cachée. A ce propos, à quoi peuvent servir ces fonctions selon vous du point de vue de la "complexité des modèles" ? Elles permettent également d'attribuer des valeurs "faibles" voire de désactiver des neurones si les sorties des modèles ne sont pas satisfaisantes. Ci-dessous quelques fonctions d'activations couramment utilisées.



Chaque fonction va avoir sa propre mais aussi ses propres propriétés (convexité ou non, caractère différentiable ou non). Avant de regarder comment apprendre notre modèle, regardons comment sont calculés valeurs des différents neurones $h_{i,j}$ pour le neurone i de la couche j sur quelques exemples. On notera $\mathbf{x}$ notre input et et $W^{(j)}$ et $b^{(j)}$ la matrice des poids et le vecteur des biais associés à la couche j. Enfin on notera f la fonction d'activation choisie :

$$h_{1,1} = f(W_{1,1}^{(1)} x_1 + W_{1,2}^{(1)} x_2 + b_1^{(1)}) \quad \text{et} \quad h_{2,1} = f(W_{2,1}^{(1)} x_1 + W_{2,2}^{(1)} x_2 + b_2^{(1)}).$$

Regardons sur un neurone de la deuxième couche, le premier par exemple $(h_{1,2})$ avec la même fonction d'activation f :

$$h_{1,2} = f(W_{1,1}^{(2)} h_{1,1} + W_{1,2}^{(2)} h_{1,2} + b_1^{(2)}).$$

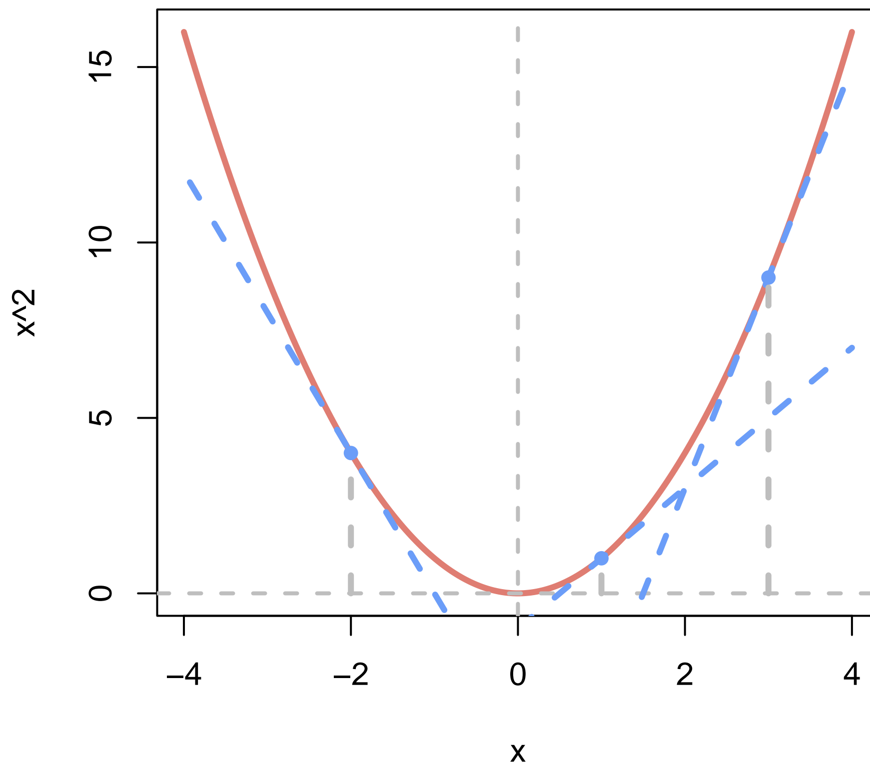
Or si on reprend les valeurs de $h_{1,1}$ et $h_{1,2}$ calculées précédemment, nous pourrions alors écrire :

$$h_{1,2} = f(W_{1,1}^{(2)} f(W_{1,1}^{(1)} x_1 + W_{1,2}^{(1)} x_2 + b_1^{(1)}) + W_{1,2}^{(2)} f(W_{2,1}^{(1)} x_1 + W_{2,2}^{(1)} x_2 + b_2^{(1)}) + b_1^{(2)}).$$

Et ainsi de suite si on rajoute d'autres couches à notre réseau.Passons maintenant à l'apprentissage de notre modèle.

Apprentissage du modèle Comme pour la plupart des algorithmes, nos réseaux de neurones dépendent de paramètres qui vont être mis à jour à chaque passage des données dans le modèle. Ce processus se fait par une rétropropagation du gradient (on détaillera cela dans la suite) qui va consister à dérivée des fonctions composées.

Regardons déjà ce qu'est la descente de gradient. Il s'agit d'un algorithme d'optimisation qui va chercher à atteindre le minimum (ou un maximum selon le contexte comme pour maximiser la vraisemblance) se basant sur la notion de gradient, i.e. en se servant des pentes de la fonction. C'est donc un algorithme de recherche solutions qui se sert des pentes de notre fonctions afin de se diriger vers le minimum ou maximum d'une fonction (vous aurez plus de détails en cours d'Optimisation). Mais regardons rapidement le cas de la fonction $x \mapsto x^2$.



On remarque, que si l'on suit la direction de la pente du gradient (en allant vers le bas), on sera capable d'approche la valeur minimale de la fonction. Par exemple, à droite de notre fonction, notre gradient $x \mapsto 2x$ est positif et ma valeur actuelle de x est éloigné de la solution optimale ($x = 0$) donc je si je vais dans le sens contraire du gradient, je vais me rapprocher de cette solution optimale (on peut faire un raisonnement analogue à gauche de la fonction, mon gradient prend des valeurs négatives donc en allant dans son sens

contraire, i.e. en augmentant la valeur de x, je me rapproche bien de la solution optimale). La solution optimale se caractérise (pour une fonction convexe) est la valeur de x pour laquelle le gradient de la fonction s'annule.

Plus formellement, notons J la fonction(nelle) que l'on souhaite minimiser, cette fonctionnelle dépend d'un paramètre w dont on va chercher la valeur pour minimiser $J(w)$. Alors notre algorithme de descente de gradient, qui consiste à "descendre le long du gradient de la fonction selon l'état courant" consiste à mettre à jour, itérativement, le paramètre w de la façon suivante :

$$w_{t+1} = w_t - \eta \nabla_w J(w_t),$$

où ∇ est un opérateur de dérivation et α est appelé "pas d'apprentissage" (ou "learning rate") et va définir à la distance à parcourir le long d'une droite pendant la descente. On va faire cette mise à jour jusqu'à ce que l'on atteigne la solution optimale, i.e. jusqu'à ce que $\|\nabla_w J(w)\| \simeq 0$.

C'est donc descente de gradient que l'on va utiliser pour mettre à jour nos paramètres de réseaux de neurones, mais d'un point de vue plus complexe, car on voit que certains paramètres dépendent directement du poids d'autres paramètres. Nous avons en effet vu que notre réseau de neurones se présentent finalement comme une succession de fonctions.

Nous avons en effet vu que notre réseau de neurones se présentent finalement comme une succession de fonctions. Ainsi les poids les plus en amonts du réseaux dépendent directement des poids en aval, i.e. pour mettre à jour les poids de ma première couche cachée (qui nécessite donc calculer la gradient par rapport à w_1) je vais devoir calculer le gradient par rapport à chacun des paramètres en aval de cette première couche ! On fait cela en utilisant la dérivation de fonctions composées

$$\frac{f \circ g)}{\partial w}(w) = \frac{\partial g(f(w))}{\partial f(w)} \times \frac{\partial f(w)}{\partial w}.$$

Je termine ici la partie explication sur l'apprentissage des réseaux de neurones. Dans la pratique ce sont bien évidemment les solveurs qui se chargeront vous faire cette mise à jour des poids du réseau.

Quelques exemples simples :

On se propose ici de regarder différents jeux de données afin de tester différentes configurations ainsi que différentes applications, on fera cela à l'aide de la library "neuralnet" de R.

Quelques exemples simple

Le percéptron simple Commençons par un premier exemple avec le perceptron, il s'agit du réseau de neurone le plus élementaire qui soit car celui-ci ne comporte pas de couches cachées. Il s'agit simplement d'un modèle linéaire avec une fonction d'activation f, i.e. notre prédiction aura la forme :

$$\hat(y) = f(w_1 x_1 + w_2 x_2 + ... + w_d x_d + b),$$

où les w_j et b sont les paramètres du modèle.

Pour ce premier exemple on va considérer le jeu de données suivant :

```
# Jeu de données
n_ech=400
x = matrix(rnorm(n_ech), n_ech/2, 2)
y = rep(c(-1, 1), c(n_ech/4, n_ech/4))
x[y == 1,] = x[y == 1,] + 5
```

1. Représentez le problème graphiquement

2. Mettre en place le perceptron simple avec une fonction d'activation de type sigmoïde (logistic) à l'aide de la librairie "neuralnet" et de la fonction "neuralnet" (on fixera "hidden=0"). Affichez les résultats sur les données d'entraînement en donnant la matrice de confusion. (On rappelle qu'un modèle logistique donne une probabilité, donc on utilisera le seuil de 0.5 pour déterminer si l'on appartient à la classe ± 1)
3. Que pouvez dire concernant le modèle ?

On considère maintenant le jeu de données suivant le 'XOR' :

```
# Mise en place du 'XOR'
X <- matrix(runif(800,-1,1),ncol=2)
y <- rep(-1,400)
y [((X[,1]>0) & (X[,2]>0) == 1)]=1
y [((X[,1]<0) & (X[,2]<0) == 1)]=1
```

1. Implémentez déjà le perceptron simple (option hidden = 0) et donnez la matrice de confusion. Que constatez vous et pourquoi ?
2. Modifier ensuite cette option (hidden) en ajoutant des neurones et le nombre couches afin d'améliorer les résultats. Essayez de trouver la meilleure configuration possible.

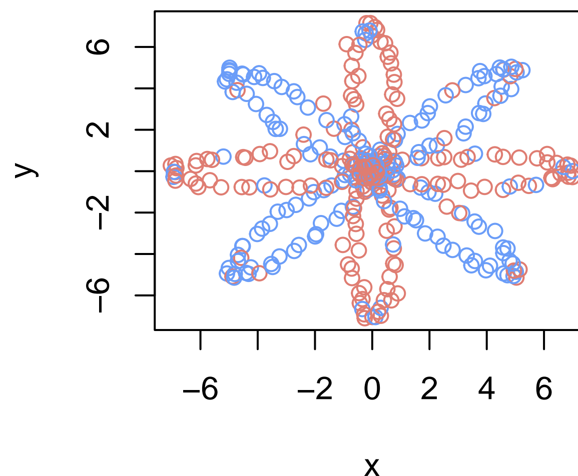
Un autre jeu de données :

En reprenant le code étudié dans le précédent TP, essayez de représenter les frontières de décisions d'un réseaux de neurones appris sur le jeux de données suivant. Il s'agit de générer des données selon une fleur et d'attribuer des étiquettes en fonction du pétale sur lequel on se trouve.

```
#Jeu de données
set.seed(69)

# Matrice de rotation d'angle 45*
rot = rbind(c(cos(pi/4), -sin(pi/4)),
            c(sin(pi/4), cos(pi/4)))

# Paramètre servant à définir l'ellipse
t = seq (0,2*pi,length=50)

# Equation paramétrique d'une ellipse
x1 = 3.5*cos(t)  # x = x_c + a*cos(t)
x2 = 0.7*sin(t)  # x = y_c + b*cos(t)
x1 = x1 - max(x1)

# On perturbe un peu l'ellipse avec un bruit gaussien
x_s = cbind(x1+rnorm(50,0,0.1),x2+rnorm(50,0,0.1))

X = NULL
Y = NULL

# Construction des différents pétales
for (i in 1:8){

  y = rep(0,50)
  s = ifelse (i%%2==0,  7 , 43 )
  y[sample(50,s)]=1
  x_s <- t(rot%*%t(x_s))
```

```
  X <- rbind(X,x_s+rnorm(100,0,0.1))
  Y <- c(Y,y)
}
```

Vous obtiendrez ainsi un jeu de données ayant la forme suivante, vous êtes libres de modifier les paramètres à votre guise afin de travailler avec des problèmes plus ou moins complexes.\



Essayer de tester différentes combinaisons de réseaux de neurones afin d'obtenir des résultats performants à l'aide de la librairie "neuralnet" de R à l'aide des indications suivantes :

1. séparer le jeu de données de façon à garder 20% des données pour la phase de test
2. entraîner plusieurs modèle en faisant varier le nombre de couches et le nombres de neurones dans chaque couche
3. étudier l'influence de ces deux paramètres (taille et profondeur) sur les résultats observés
4. option : représenter les zones de décisions en utilisant le code indiqué dans le précédent TP

Jeu de données MNIST (images)

Ce jeu de données est certainement le plus connu en Machine Learning pour les amateurs de réseaux de neurones, c'est un jeu de données très classique sur les lequel les algorithmes actuels peuvent obtenir des performances extraordinaires.

Pour ce faire, vous commencerez par installer "tensorflow" et téléchargerez les données MNIST

```
# Vous aurez peut-etre d'installer le package devtools
#devtools::install_github("rstudio/keras")
library(keras)
#install_tensorflow()
```

On commencera ensuite par étudier le jeu de données "MNIST"

```
mnist<-dataset_mnist()
```

1. Séparez votre jeu de données en un ensemble d'entraînement et de test à l'aide des informations fournies dans le jeu de données (on remarquera que le découpage est déjà effectué)

2. Etudiez le jeu de données : nombre d'exemples, dimension, label, valeurs des variables
3. Que représente ces données selon vous ?
4. Utilisez la fonction "image" pour ploter une des images

Après cette première étude, on va maintenant regarder comment implémenter son propre réseau de neurones à l'aide de cette librairie "Keras" et de "Tensorflow". La syntaxe n'étant pas évidente, en voici un exemple ci-dessous afin que vous puissiez le modifier et créer un modèle qui va permettre de correctement classer vos données.

```
# Séparation des données en train/test
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y


# On transforme nos données sous forme de vecteurs (784 = 28 x 28)
dim(x_train) <- c(nrow(x_train), 784)
dim(x_test) <- c(nrow(x_test), 784)
# On va ensuite rescaler nos données dans [0,1] pour travailler avec des niveaux de gris
x_train <- x_train / 255
x_test <- x_test / 255
# Enfin, on indique bien à R que les nombres 0 à 9 sont des classes et non des entiers
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

On peut maintenant regarder comment construire un modèle séquentiel.On va définir chaque couche les unes après les autres en choisissant à chaque fois le nombre de neurones dans la couche en question, la fonction d'activation à utiliser.

Pour le premier neurone on précisera également la dimension des données en entrée. Enfin pour le dernier neurone, on précisera aussi la dimension de la sortie (cette dimension est égale au nombre de classes à prédire, ici 10).

Une autre fonction "layer_dropout" peut également être utilisée. Elle a pour effet d'annuler l'effet d'un neurone dans la prise de décision lors du passage d'une donnée. On indique donc la probabilité qu'un neurone soit désactivé lors du passage d'une donnée. Cela permet d'éviter le sur apprentissage.

```
# Mise en place du modèle (de sa structure)
model <- keras_model_sequential() #
model %>%
  layer_dense(units = 4, activation = "relu", input_shape = c(784)) %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 3, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")

# Pour avoir quelques informations sur votre modèle
summary(model)
```

Passons maintenant au processus d'apprentissage en définissant la fonction à optimiser, l'optimiseur (i.e. l'algorithme de descente de gradient à employer) ainsi que la métrique d'évaluation (le taux de bonne classification par exemple).

```r
# Contexte de l'apprentissage
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = adam(),
  metrics = c("accuracy")
)
```

On peut maintenant passer à l'entraînement du modèle à l'aide de la fonction "fit". On précise les données d'entraînement, le nombre d'epochs correspond au nombre de fois où le modèle voit le jeu de données complet (ici 30 fois), batch size sert à définir la taille des groupes de données qui passe itérativement dans le modèle pendant l'apprentissage. Enfin validation split indique le pourcentage de données utilisé pour évaluer le modèle après chaque epoch (ici 20%).

```r
# entraînement du modèle
history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

Finalement, vous pouvez regarder l'évolution des performances du modèle au cours de l'apprentissage avec la fonction "plot" et afficher les performances de votre modèle sur l'ensemble test a posteriori

```r
# Analyse du modèle
plot(history)
model %>% evaluate(x_test, y_test,verbose = 0)
```

Vous pouvez maintenant regardez comment faire de même avec des réseaux convolutionnels avec les jeux de données "CIFAR10" ou encore "IMDB"

```r
cifar10<-dataset_cifar10()
imdb<-dataset_imdb()
```

Pour cela vous pouvez vous inspirez du code ci-dessous, pour le jeu de données CIFAR10 et de ce qui précède pour entraîner un modèle de classification

```r
model <- keras_model_sequential() %>%
layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu", input_shape = c(32,32,3)) %>%
layer_max_pooling_2d(pool_size = c(2,2)) %>%
layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%
layer_max_pooling_2d(pool_size = c(2,2)) %>%
layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%
layer_flatten()
```

Les réseaux convolutionnels consistent à réduire la dimension des images afin de réduire le nombre de paramètres à apprendre, cela se fait à l'aide d'un masque de convolution de taille fixe ($d \times d$) qui va parcourir notre image afin de génére de nouvelles images à partir de cette convolution mais dont la taille sera réduite. On combine cela à des opérations de sampling –> pooling pour réduire encore la dimension des données.

- Convolution (layer conv 2d) : filters : nombre d'images à générer (= nombre de masques), kernel size : taille du masque –> apprentissage du masque
- Pooling (layer max pooling) : filters : nombre d'images à générer (= nombre de masques), kernel size : taille du masque –> ici on retient la plus grand valeur observée pour chaque position du masque sur l'image.
- Layer flatten va ensuite vous permettre de passer d'un format 'tensoriel' (généralisation des matrices) à un format vectoriel de vos données, vous pourrez ainsi ajouter des coiches classiques denses comme sur

l'exemple pour le jeu de données MNIST, le reste est inchangé.

Regardez maintenant les différentes options qui s'offrent à vous afin de créer le modèle le plus performant sur les différnts jeux de données.

Pour finir

Les personnes qui souhaitent voir comment coder son propre réseaux de neurones sous R "from scratch" peuvent consulter le site suivant : il vous proposera de construit pas à pas les différentes étapes de votre réseau de neurones avec une seule couche cachée :

Tutoriel : Réseaux de Neurones