



INSTITUT  
de la  
communication



# Introduction to Statistical Supervised Machine Learning TD - Mise en pratique Master 1 MIASHS (2022-2023)

Guillaume Metzler

Institut de Communication (ICOM)  
Université de Lyon, Université Lumière Lyon 2  
Laboratoire ERIC UR 3083, Lyon, France


[guillaume.metzler@univ-lyon2.fr](mailto:guillaume.metzler@univ-lyon2.fr)

## Abstract

L'objectif de cette présente fiche est de mettre en oeuvre le processus d'apprentissage en Machine Learning ainsi que les différents algorithmes qui sont présentés en cours.

On souhaite avant tout expliquer quelles sont les bonnes pratiques expérimentales à conduire : de la normalisation des données, choix d'une mesure de performance, à la mise en place de la cross-validation et de la comparaison des modèles sur différents jeux de données.

On souhaite également illustrer et comparer les différents algorithmes étudiés, que cela soit pour des tâches de régression ou encore des tâches de classification.

L'ensemble de ces expériences seront réalisées sous  et on s'attachera d'avantage à présenter les fonctions à employer pour réaliser les différents points. Cela n'exclut cependant de devoir programmer un minimum pour la manipulation des données, le calcul d'une mesure de performance ou encore la mise en place du protocole expérimental. En aucun cas on cherchera à recoder les algorithmes, sauf éventuellement le plus élémentaire.

# 1 Mise en place d'un protocole avec l'algorithme des plus proches voisins

## 1.1 Une implémentation brute

Nous commencerons par regarder comment implémenter son propre algorithme des plus proches voisins. Pour cela on va simuler notre propre jeu de données sur lequel nous allons travailler.

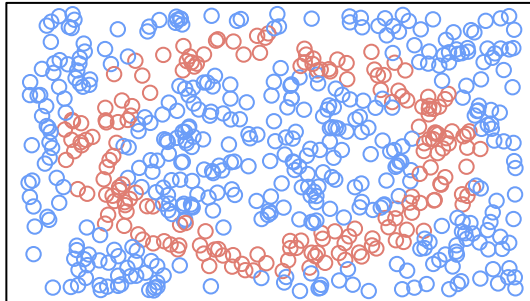
Les informations relatives aux individus seront stockées dans une matrice  $X$  et les labels relatifs dans un vecteur  $Y$ .

Commençons par générer un jeu de données

1. Générer un jeu de données  $X$  en 2D dans l'espace  $[-2, 2]^2$  selon une loi uniforme, on prendra  $n = 600$ .  
On fixera la graine à la valeur 3 en amont, à l'aide de la fonction `set.seed()`
2. Calculer la distance des points à l'origine et attribuer l'étiquette  $Y = 1$  si la distance à l'origine est comprise entre 1.5 et 3 et  $-1$  sinon
3. Représenter votre jeu de données, il devrait ressembler à la figure ci-dessous

```
# Représentation du jeu de données

plot(X, col = ifelse(Y == 1, "#DF7D72", "#6B9DF8"),
      xlab = "", ylab = "", xlim = c(-2,2), ylim = c(-2,2),
      xaxt="n", yaxt="n")
```



On va maintenant implémenter la fonction nous permettant de créer notre propre algorithme du plus proche voisin `my_knn`, il prendra au total quatre arguments :

- la valeur de  $k$  précisant le nombre de voisins
- votre jeu de données d'entraînement  $X_{\text{train}}$
- les étiquettes  $Y_{\text{train}}$  de vos exemples
- votre jeu de données test  $X_{\text{test}}$

Vous êtes libres sur la façon dont vous implémenter votre algorithme (boucle ou autres process) mais vous devrez faire figurer les étapes suivantes :

- Calculer la distance entre les objets de  $X_{\text{test}}$  aux objets  $X_{\text{train}}$ .
- Pour chaque exemple, on ordonnera les distances dans l'ordre croissant à l'aide de la fonction `sort`. On s'intéresse surtout aux indices des exemples pour récupérer les étiquettes associées !
- On attribue ensuite les étiquettes aux données tests en fonction des  $k$ -plus proches voisins.

4. Ecrire votre algorithme du plus proche voisin, on pourra compléter le code suivant

On va maintenant regarder les performances de l'algorithme, on va donc créer une fonction qui va prendre comme paramètres les données labels de nos données d'entraînement et les labels des données tests pour calculer le taux d'erreur.

5. Ecrire une fonction qui permet de calculer le taux d'erreur de votre algorithme. Elle dépendra de deux paramètres : les vrais labels et les prédictions de l'algorithme.

Maintenant que votre procédure est écrite, nous allons maintenant tester notre procédure

6. générer un jeu de données tests, de taille  $n = 300$ , de la même façon que votre jeu d'entraînement
7. tester votre modèle pour différentes valeurs de  $k$ , par exemple pour toutes les valeurs de  $k$  allant de 1 à 17 et représentez graphiquement les résultats afin d'obtenir un graphe qui ressemble à celui présenté ci-dessous :

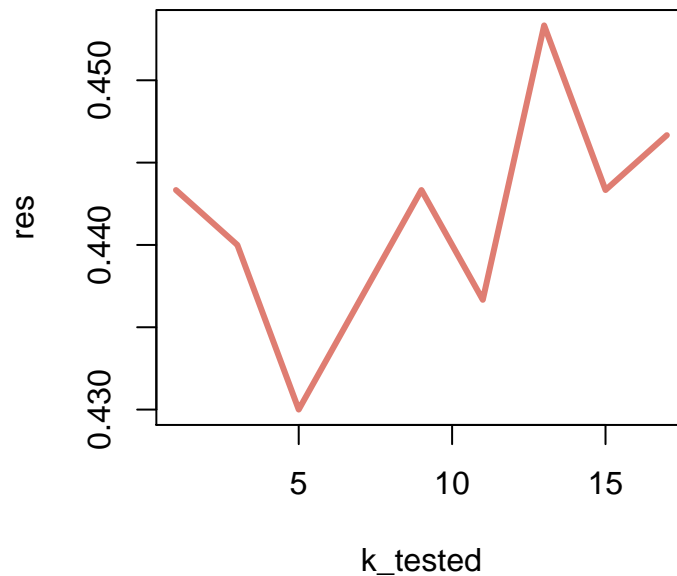
```
# Jeu de données tests


k_tested <- seq(1,17,by=2)
res <- rep(0,length(k_tested))


X_test <- cbind(x1 = runif(300,-2,2),x2 = runif(300,-2,2) )
Y_test <- ifelse( (1.5<dist)&(dist<3), 1, -1)

plot(k_tested, res, type='l', lwd= 3, col = "#DF7D72", main =
"Résultats en fonction de k")
```

## Résultats en fonction de k



On souhaite maintenant comparer notre implémentation avec la fonction `knn` de .

8. Comparez vos résultats avec la fonction  $k$ -NN de la librairie `class` de , en utilisant les instructions ci-dessous (on s'assurera de retrouver la même chose en terme de résultats)

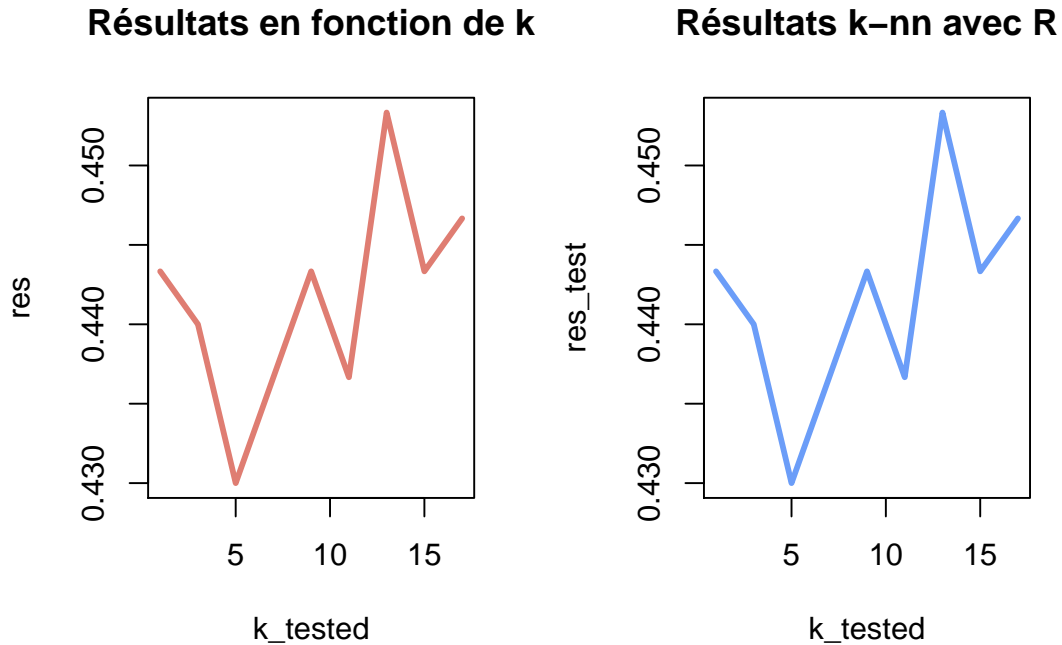
```
library(class)

res_test = NULL
i=1

for (j in seq(1,17,2)){
  pred <- knn(X,X_test,Y,k=j)
  res_test[i] = mean(pred != Y_test)
  i=i+1
}

par(mfrow=c(1,2))
plot(k_tested, res, type='l', lwd= 3, col = "#DF7D72",
```

```
main = "Résultats en fonction de k")
plot(k_tested, res_test, type = 'l', lwd= 3, col = "#6B9DF8",
main = "Résultats k-nn avec R")
```



Cette première étape a permis de se familiariser avec la manipulation d'un ensemble d'apprentissage et de test avec une fonction implémentée soit même et une fonction prête à l'emploi sous . On va maintenant se concentrer uniquement sur l'usage de fonctions disponibles avec et aborder le problème d'un point de vue Machine Learning.

## 1.2 Un vrai apprentissage de modèles : validation croisée

Pour le moment, nous avons simplement tester différentes valeurs de  $k$ , mais dans la pratique, il faut que l'algorithme se serve de l'ensemble d'apprentissage pour qu'ils apprennent de lui-même quelle est la valeur de  $k$  la plus intéressante. Cela peut se faire à l'aide d'un ensemble de validation par validation croisée.

Pour cela, utilisera le package `caret` disponible sous ainsi que le jeu de données *Pima* disponible dans les ressources du TD

```
library(caret)

pima = read.csv("datasets/pima.csv")
head(pima)

##   X1  X2 X3 X4  X5   X6    X7 X8 y
## 1  6 148 72 35   0 33.6 0.627 50 1
## 2  1  85 66 29   0 26.6 0.351 31 0
## 3  8 183 64  0   0 23.3 0.672 32 1
## 4  1  89 66 23  94 28.1 0.167 21 0
## 5  0 137 40 35 168 43.1 2.288 33 1
## 6  5 116 74  0   0 25.6 0.201 30 0

pima$y <- as.factor(pima$y)
Var = c("X1", "X2", "X3", "X4", "X5", "X6", "X7", "X8")
```

Il y a donc un ensemble de 8 descripteurs.

9. Etudier le range de valeurs des différentes variables et, si nécessaire, normalisez les données par la méthode de votre choix.
10. A l'aide de la fonction `createDataPartition`, séparez votre jeu de données en deux : 75% des données pour l'entraînement et 25% pour le test
11. Etudier le fonctionnement de la fonction `trainControl` afin de définir une 10-fold cross-validation
12. A l'aide de la fonction `expand.grid` définir la grille de recherche des valeurs de  $k$  pour l'algorithme, on prendra les valeurs impaires de 1 à 19

```
# Partition du jeu de données

index_train <- createDataPartition(y=pima$y, p = 0.75, list=FALSE)
train <- pima[index_train,]
test <- pima[-index_train,]

# Préparation de la validation croisée

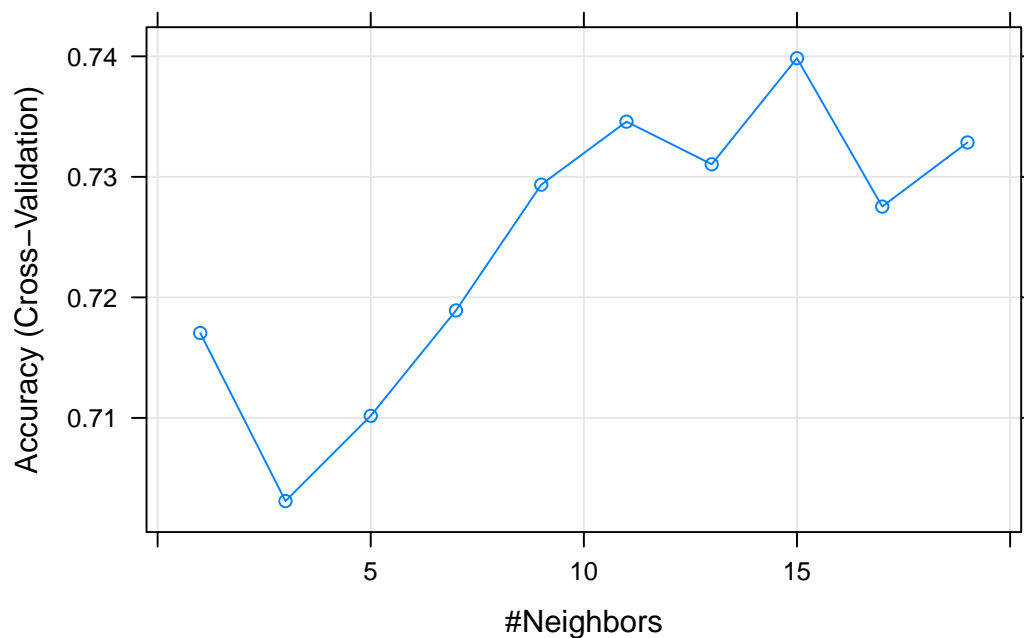
train_control <- trainControl(method="cv", number=10)
mygrid <- expand.grid(k=seq(1,19,2))
```

On cherche maintenant à représenter les résultats graphiquement.

13. Le modèle s'apprend à l'aide de la fonction `train` comme illustré ci-dessous. On peut ensuite regarder ce que donne le modèle à l'aide de la fonction `plot`. Quelle remarque pouvez-vous faire concernant la valeur de  $k$  ?

```
model <- train( y~ . ,data=train,
               trControl=train_control,
               method="knn",
               tuneGrid = mygrid
             )

plot(model)
```



On peut avoir le détail des résultats de la cross-validation à l'aide de la fonction `summary`. Par défaut, le modèle optimal retenu en phase de test sera celui maximisant l'accuracy en validation croisée.

14. Evaluer votre modèle sur votre jeu de données test en calculant l'accuracy à l'aide de la fonction `predict`
15. Donner la matrice de confusion



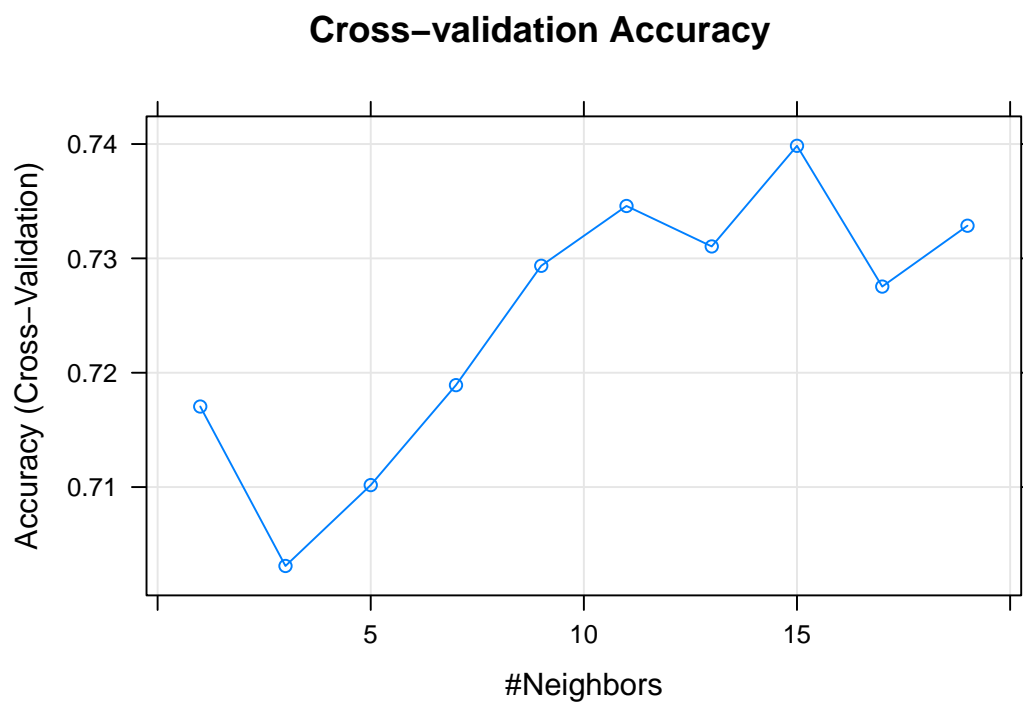
16. Comparez l'erreur en cross-validation et l'erreur en test pour les différentes valeurs de  $k$  de votre algorithme. Est-ce que la valeur retenue par le processus de cross-validation vous semble pertinente ?

```
# Vérification en test

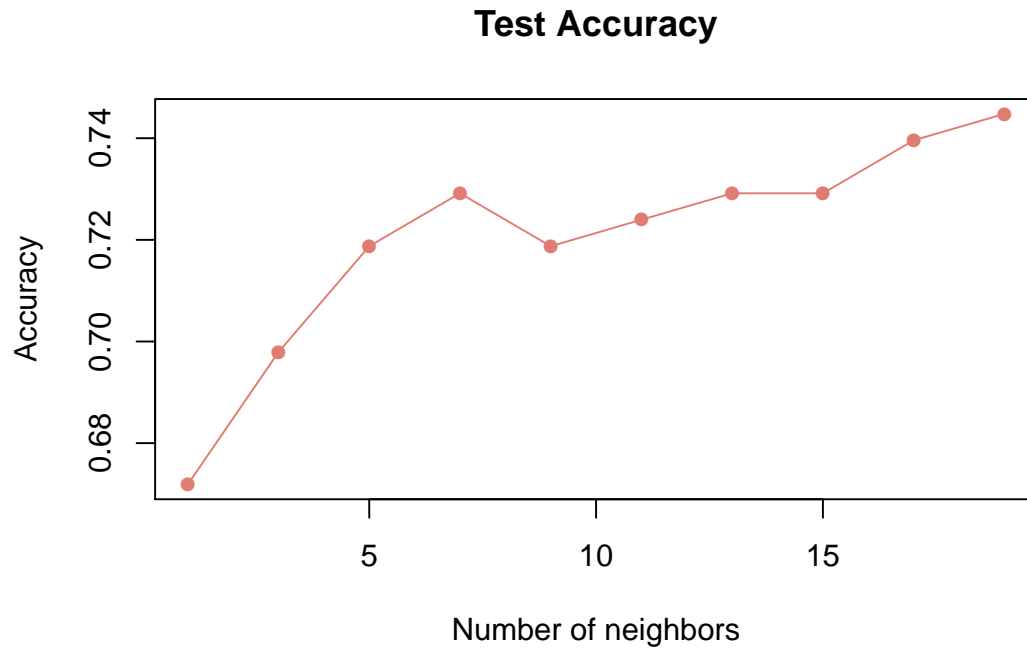
table_test = NULL
i=1

for (j in seq(1,19,2)){
  res <- knn(train[,-which(colnames(test)== "y")],
             test[,-which(colnames(test)== "y")],
             train[, "y"],
             k=j)
  table_test[i] = mean(res == test[, "y"])
  i=i+1
}

plot(model, main = "Cross-validation Accuracy")
```



```
plot(seq(1,19,2), table_test, xlab = "Number of neighbors",
     ylab="Accuracy", pch=16,
     col = "#DF7D72", type='o', main = "Test Accuracy" )
```



Nous venons donc de réaliser un processus complet d'apprentissage - validation - test.

Idéalement, lorsque l'on cherche à évaluer les performances d'un algorithme, on doit effectuer ce processus plusieurs fois et moyenner les résultats obtenus et calculer l'écart-type. Cela permet d'avoir une idée plus précise des performances mais aussi d'avoir une idée de la variabilité des résultats.

## 2 Classification

### 2.1 La Régression Logistique

### 2.2 Les Séparateurs à Vaste Marge

Le code ci-dessous vous permettra de générer des graphiques vous permettant de visualiser les résultats de votre algorithmes.

```

# On fixe la graine, cela vous permettra d'avoir des résultats identiques
set.seed(10111)

# librairie
library(e1071)

# Grille pour graphe
make.grid = function(x, n = 200) {
  grange = apply(x, 2, range)
  x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
  x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
  expand.grid(X1 = x1, X2 = x2)
}

#Jeu de données
n_ech = 100
x = cbind(rnorm(100), rnorm(100))
y = (2*rbinom(100,1,0.5)-1)
x[y == 1,] = x[y == 1,] + 4

# Génération de la grille adaptée à la taille des données
xgrid = make.grid(x)

```

On va maintenant s'attaquer à l'apprentissage du modèle. Pour cela regarder comment fonctionne la fonction `svm` et apprendrait un modèle de SVM linéaire qui va chercher à prédire  $y$  en fonction de  $x$  avec un paramètre  $C$  (cost) que vous fixerez à 1 dans un premier temps et répondez aux questions suivantes :

1. Faites varier  $C$  (cost) dans l'intervalle  $[0, 10]$  par exemple.
2. Qu'observez-vous ? Quelle est l'influence de ce paramètre ?
3. Vous avez observé que certains points sont encadrés, que peut-on dire la position de ces points par rapport au séparateur (ligne noire) ? Que représentent ces points ?

```

# Apprentissage du modèle et évaluation des labels

dat = data.frame(x, y = as.factor(y))
svmfit = svm(y~., data = dat, cost = 10, scale = FALSE, kernel = "linear")
print(svmfit)
plot(svmfit, dat)

```

```

ygrid = predict(svmfit, xgrid)

# Extraction des paramètres

beta = drop(t(svmfit$coefs)%*%x[svmfit$index,])
beta0 = svmfit$rho

# Représentation graphique

plot(xgrid, col = ifelse( (as.numeric(ygrid)-1)==1
, "#DF7D72", "#6B9DF8"), pch = 20, cex = .2,
     main = "Un SVM linéaire")
points(x, col = ifelse( y==1 , "#DF7D72", "#6B9DF8"), pch = 19)
points(x[svmfit$index,], pch = 5, cex = 2)
abline(beta0 / beta[2], -beta[1] / beta[2])
abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)

```

Ce cas là est presque trop simple dans le sens où nos données sont parfaitement séparables. Reprendre le bout de code permettant de générer les données de façon à ce que le jeu de données ne soit pas linéairement séparable et refaite quelques tests sur la valeur du paramètre  $C$  (cost) en affichant aussi la matrice de contingence des résultats de classification.

En repartant du code précédant et en modifiant le jeu de données comme indiqué ci-dessous, que pouvez vous dire concernant ce type de modèle ?

```

# Génération des données
n_ech = 400
x = matrix(rnorm(n_ech), n_ech/2, 2)
y = ifelse(x[,1]^2+x[,2]^2<1, 1, -1)

```

En fait les modèles linéaires ne sont que très rarement utiles en pratiques lorsqu'ils sont employés seuls (en les combinant on peut créer des modèles non linéaires puissants ! Mais cela sort du cadre de ce TP).

Une autre approche consiste à utiliser des méthodes à noyaux, cela permet de potentiellement projeter ses données dans un espace de dimension infinie dans lequel un séparateur linéaire est capable de correctement séparer les données.

1. Regardez les options de la fonction "svm" afin de créer un modèle gaussien.

2. Combien d'hyper-paramètres devons-nous renseigner pour ce type de modèle ? Essayez d'expliquer l'influence de ces paramètres.
3. Jouer avec le paramètre spécifique au noyau gaussien pour observer son effet.

On pourra également comparer les différences de résultats sur le jeu de données *cats* disponible dans la librairie **MASS**.

On se propose maintenant d'apprendre le meilleur modèle non linéaire gaussien qui permet de maximiser les performances en classification et les comparer à un potentiel meilleur modèle linéaire. Pour cela, on utilisera un jeu de données disponible dans les ressources disponible dans les ressources de ce TD. On se passera cette fois-ci d'une représentation graphique, sauf si vous souhaitez en faire une.

On rappelle les différentes étapes :

1. Commencer par séparer votre jeu de données en un ensemble train/test avec 75%/25% des données.
2. Effectuer une 4-CV pour apprendre le meilleur SVM linéaire en tunant les hyper-paramètres associés
3. Faire de même avec un SVM non linéaire
4. Tester les deux modèles sur votre jeu de données test et comparer les résultats.

## 2.3 Les Arbres de Décision

# 3 Régression

## 3.1 La Régression Linéaire

## 3.2 Les Arbres de Décision

# 4 Autour 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.

## 4.1 Généralités

Les structures de réseaux de neurones sont également 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<sup>1</sup> par un système de deux modèles en opposition. 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.

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).

Regardons déjà la structure d'un réseau de neurones avec le code suivant :

```
library(neuralnet)
XOR <- c(0,1,1,0)
xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
```

<sup>1</sup>On peut en avoir un exemple au lien suivant : [Exemples sur ce site](#)

```
net.xor <- neuralnet( XOR~Var1+Var2, xor.data, hidden=c(2,3), rep=5)
plot(net.xor, rep="best")
```

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 près cette structure en trois niveaux :

- 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 liens, ou le poids des différents neurones pour la prédiction associée à un nouveau neurone. 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  $h_1, h_2, \dots, h_H$ .
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 Error) :

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

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 !

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

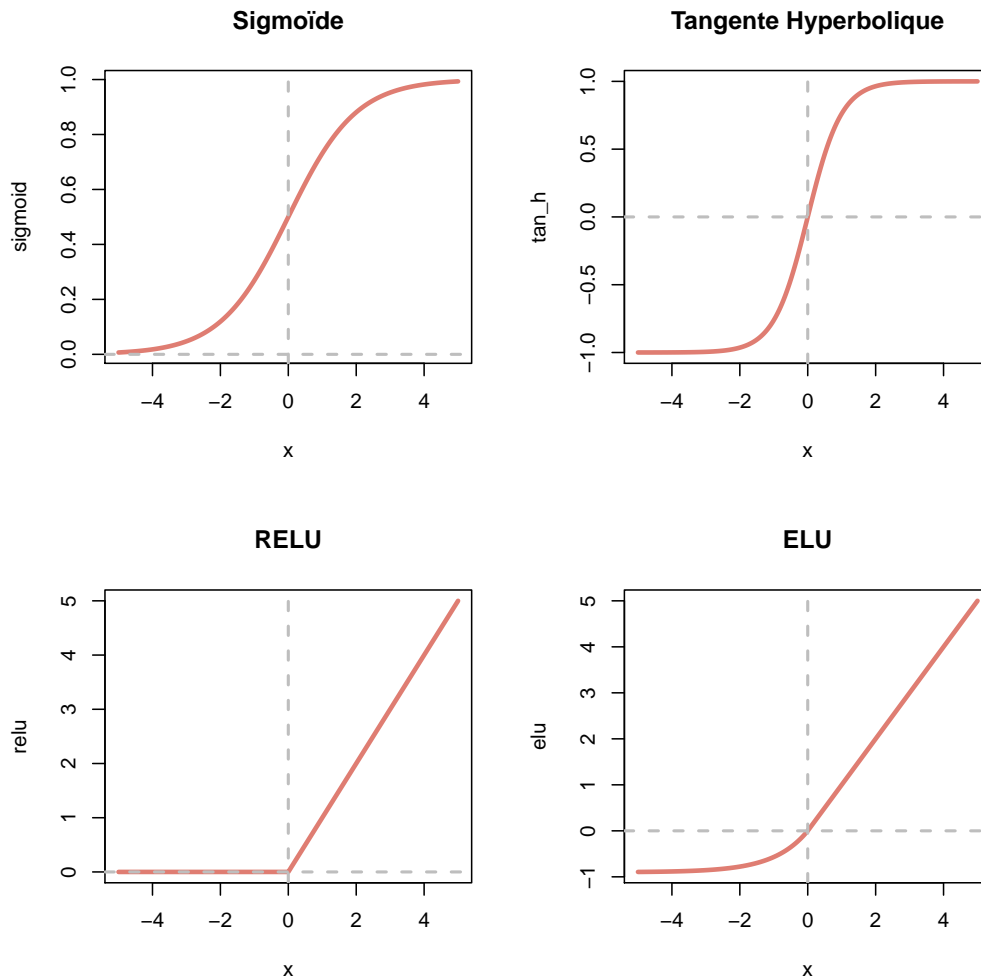
ou encore, dans le cas multi-classe

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

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  $W^{(j)}$  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 :

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

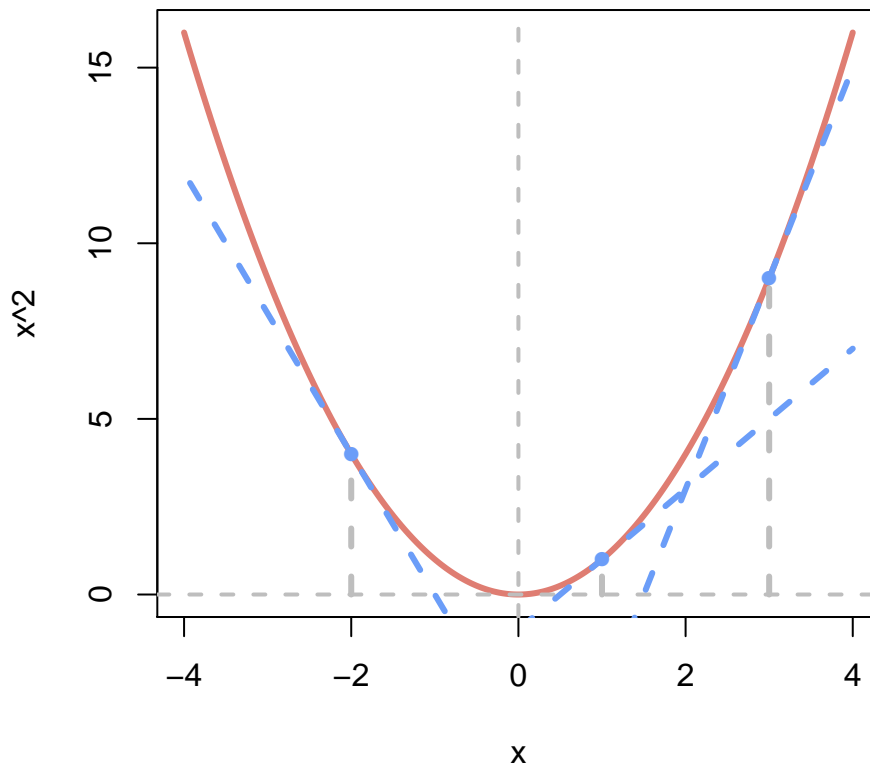
Et ainsi de suite si on rajoute d'autres couches à notre réseau. Passons maintenant à l'apprentissage de notre modèle.

## 4.2 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 rétro-propagation du gradient (on détaillera cela dans la suite) qui va consister à dériver 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. Mais regardons rapidement le cas de la fonction  $x \mapsto x^2$ .

## Square function with gradients



On remarque, que si l'on suit la direction de la pente du gradient (en allant vers le bas), on sera capable d'approcher 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 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  $\nabla$  est un opérateur de dérivation et  $\alpha$  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 amont 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.

### 4.3 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 .

**Le perceptron simple** Commençons par un premier exemple avec le perceptron, il s'agit du réseau de neurone le plus élémentaire 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 :

$$\hat{y} = f(w_1x_1 + w_2x_2 + \dots + w_dx_d + b),$$

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. Représenter 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"). Afficher 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  $\pm 1$
3. Que pouvez dire concernant le modèle ?

```
# Implémentation du perceptron simple et représentation graphique

library(neuralnet)

data <- data.frame(x,y)
net <- neuralnet( y~X1+X2, data, hidden=0, act.fct = "logistic" , learningrate = 0.2, algo
plot(net, rep="best")

# Matrice de confusion

y_hat <- predict(net, data[,c(1,2)])
table(data$y, 2*(y_hat > 0.5)-1)
```

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

“r 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.

## 4.4 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){

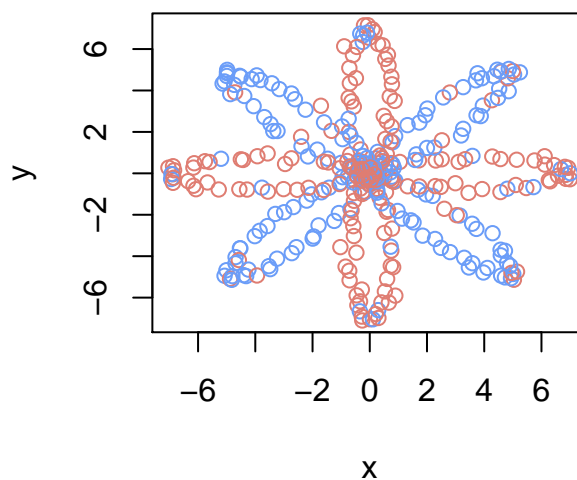
  y = rep(0,50)
  s = ifelse(i%%2==0, 7, 43)
  y[sample(50,s)]=1
  x_s <- t(rot%*%t(x_s))

  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.

### Flower dataset



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

## 4.5 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 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. Séparer 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. Etudier le jeu de données : nombre d'exemples, dimension, label, valeurs des variables
3. Que représente ces données selon vous ?
4. Utiliser la fonction `image` pour obtenir une représentation des images

```
x_train <- mnist$train$x
y_train <- mnist$train$y

# visualize the digits
par(mfcol=c(4,4))
par(mar=c(0, 0, 3, 0), xaxs='i', yaxs='i')
for (idx in 1:16) {
  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]))
}
```



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).

```

# 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%).

```

# 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

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

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

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

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


```
model <- keras_model_sequential() %>%
layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu", input_shape = c(32,
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 générer de nouvelles images à partir de cette convolution mais dont la taille sera réduite. On combine cela à des opérations de sampling et de pooling pour réduire encore la dimension des données ainsi que pour améliorer les capacités du modèle à généraliser.

- *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 grande 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 couches 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 différents jeux de données.

## 4.6 Pour finir

Les personnes qui souhaitent voir comment coder son propre réseaux de neurones sous  *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 from Scratch](#)