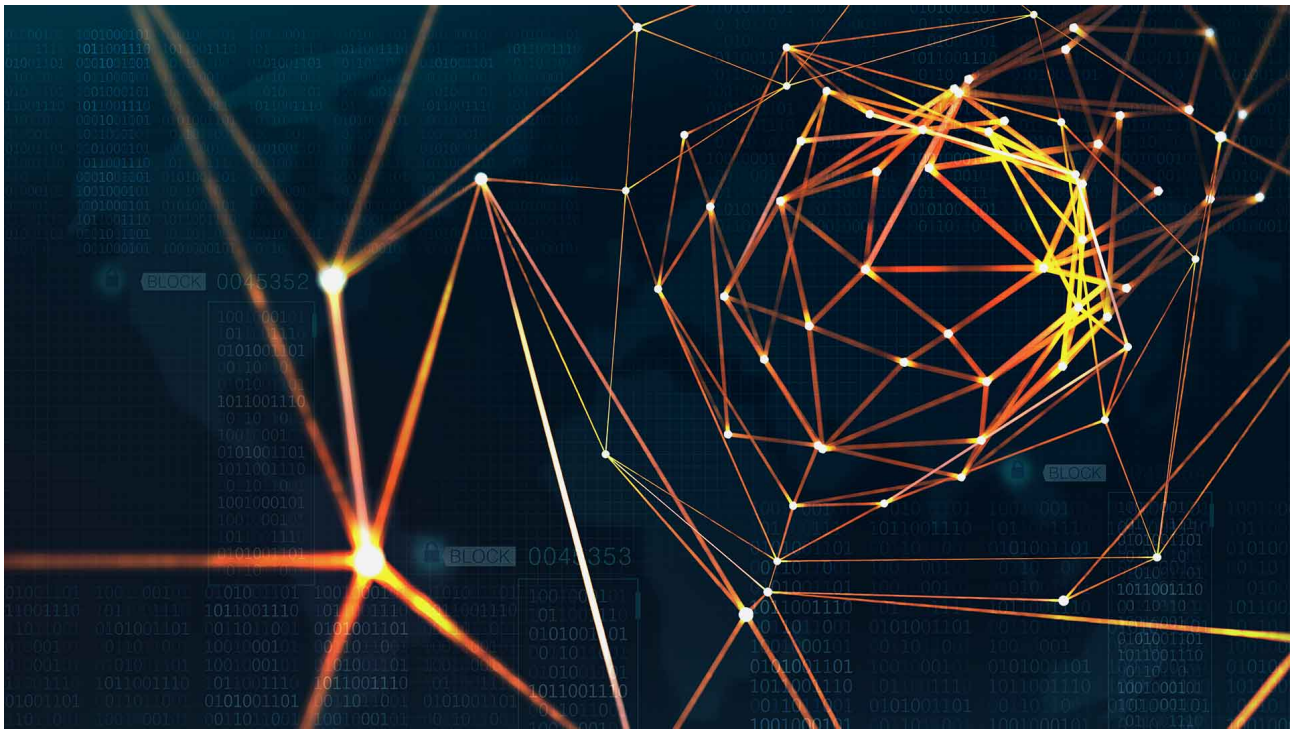
Réseaux de neurones



Élève :

Vincent Le Goualher
ISUP 3 ISDS
Alternant chez *Crédit Mutuel*

Professeure :

Annick Valibouze
Directrice adjointe de l'ISUP
Laboratoires LIP6 & LPSM

15/04/2020

Table des matières

1 Les réseaux de neurones artificiels

1.1 Introduction

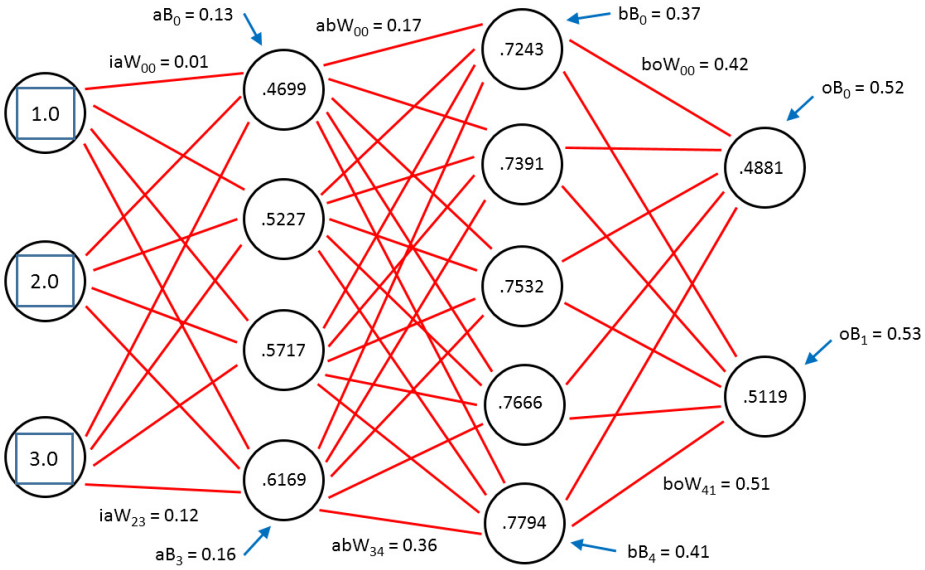
Les réseaux de neurones artificiels, schématiquement inspirés de la structure cellulaire du cerveau, sont des ensembles d'unités de calculs appelées neurones formels. Les neurones sont reliés entre eux et leur état évolue en fonction du temps. L'étude des réseaux de neurones est un champ de recherche à part entière de la statistique et d'une multitude d'autres domaines, avec des applications innombrables dans la médecine, la biologie, la cybersécurité, la finance, etc... Certaines prouesses techniques sont particulièrement médiatisées. Ce fut le cas en 2016 lorsque le réseau de neurones AlphaGo a battu Lee Sedol au jeu de go [1]. Trois ans après cette défaite, le joueur dix-huit fois champion du monde prend sa retraite et déclare : "même si je suis numéro un, il demeure une entité qui ne peut être vaincue." [2]

FIGURE 1 – Lee Sedol vs AlphaGo

1.2 Description mathématique

Un réseau de neurones peut-être vu comme un graphe orienté pondéré dans lequel chaque noeud est un neurone formel :

FIGURE 2 – Représentation d'un réseau de neurones



Notons N le nombre de neurones du réseau et $w_{i,j}$ le poids de l'arc du neurone i vers le neurone j, où $i, j \in [\![1; N]\!]$. Le neurone $j \in [\![1; N]\!]$ est caractérisé à l'instant discret t par :

- Un signal d'entrée $in_j(t)$
- Une activation $a_j(t)$
- Un *seuil* θ_j
- Une *fonction d'activation* f_j
- Un signal de sortie $out_j(t)$. Le plus souvent, $out_j(t) = a_j(t)$.

Notons $W_j := (w_{1,j}, ..., w_{N,j})$ l'ensemble des poids des arcs vers le neurone j (certains d'entre eux peuvent être nuls s'il n'y a pas d'arc), $A(t)$ le N-uplet de toutes les activations du réseau et ϕ_j une valeur réelle éventuellement nulle appelée *biais* du neurone. Le **signal d'entrée** du neurone j à l'instant t s'écrit $in_j(t) := in(W_j, A(t), \phi_j)$. La quantité $in_j(t)$ peut donc éventuellement dépendre de l'activation du neurone j lui-même, $a_j(t)$. Si c'est le cas, les activations initiales des neurones concernés doivent être correctement initialisées pour que le calcul du signal d'entrée soit possible. La fonction in est souvent booléenne, linéaire ou affine si le biais $\phi_j \neq 0$.

La *fonction d'activation* f_j et le *seuil* θ_j jouent un rôle déterminant puisqu'ils permettent de déterminer l'**activation** du neurone j à l'instant $t+1$ via la relation : $a_j(t+1) = f_j(a_j(t), in_j(t), \theta_j)$.

À noter qu'un *neurone d'entrée* n'a aucun neurone qui le précède : son signal de sortie est la valeur que l'on souhaite communiquer au réseau. À l'inverse, un *neurone de sortie* n'a pas de successeur, son signal de sortie est la valeur (ou l'une des valeurs) calculée par le système.

1.3 L'apprentissage

D'après ce qui précède, un réseau de neurones peut être décrit comme une application $y = f(x, \Theta)$ où x est l'entrée, y la sortie et Θ l'ensemble des paramètres du réseau de neurones. Le plus souvent, l'entrée x est un vecteur de $\mathbb{R}^d$ où $d \in \mathbb{N}$: chacune de ses composantes est l'activation d'un neurone d'entrée. La sortie y, éventuellement multidimensionnelle, est la plupart du temps à valeurs réelles dans le cas d'une régression ou à valeurs discrètes dans le cas d'une classification. En choisissant convenablement les fonctions d'activation de la couche de sortie, on peut imposer une contrainte sur la sortie y, comme par exemple $y \in [0; 1]$ dans le cas du calcul d'une probabilité.

Θ est l'ensemble des paramètres du réseau de neurones, à savoir l'ensemble des poids $\{w_{i,j} | i, j \in [\![1; N]\!]\}$ et éventuellement des biais $\{\phi_j, j \in [\![1; N]\!]\}$. L'efficacité du réseau de neurones dans la tâche qu'on désire le voir accomplir est quantifiée par une fonction dite d'erreur ou de perte que l'on note $E(\Theta)$.

L'objet de l'*apprentissage* est la recherche d'un ensemble de paramètres Θ_0 tel que l'erreur $E(\Theta_0)$ soit la plus faible possible. Cette recherche s'effectue le plus souvent par un algorithme d'optimisation itératif :

1. Initialisation du paramètre Θ.
2. Calcul de l'erreur $E(\Theta)$ et des quantités relatives à la règle d'apprentissage.
3. Correction du paramètre Θ via la règle d'apprentissage.
4. Tant qu'un certain critère n'est pas satisfait, itération à l'étape 2.

Il existe de nombreuses règles d'apprentissage. L'une des plus connues est la méthode dite de rétropropagation du gradient qui cherche à minimiser le paramètre Θ par un algorithme de descente du gradient. D'autres règles s'appuient sur la méthode de Newton ainsi que ses

variantes [3] ou encore l'ajout ou la suppression de neurones, la modification des fonctions d'activation, etc... Les règles d'apprentissage sont souvent spécifiques à un paradigme donné. On présente maintenant les trois principaux paradigmes d'apprentissage.

1.3.1 L'apprentissage supervisé

Dans le cadre de l'apprentissage supervisé, on dispose d'un ensemble de couple $\{(x_i, y_{d_i})|i \in [\![1; n]\!]$ où $n \in \mathbb{N}$, x_i est la valeur de la variable explicative et y_{d_i} la valeur de la variable à expliquer réelle/désirée associée à la valeur x_i. Soit $\mathcal{C}$ un sous-ensemble de $[\![1; n]\!]$. On définit la fonction d'erreur $E(\Theta, \mathcal{C})$ du réseau de neurones en comparant le signal de sortie $y_i = f(x_i, \Theta)$ à la sortie réelle/désirée y_{d_i} pour tout $i \in \mathcal{C}$. Formellement, la fonction de perte du réseau de neurones s'écrit $E : (\Theta, \mathcal{C}) \to E(\{(f(x_i, \Theta), y_{d_i})|i \in \mathcal{C}\})$. Il existe de nombreuses fonctions d'erreur, dont l'une des plus courantes est l'erreur quadratique dans le cas d'un signal de sortie réel.

Ce procédé peut conduire à un phénomène de sur-apprentissage : l'erreur $E(\Theta, \mathcal{C})$ est très faible mais $E(\Theta, [\![1; n]\!] \setminus \mathcal{C})$ est très élevée. Afin de garantir de bonnes capacités de généralisation, l'apprentissage doit être régulièrement suspendu afin d'estimer et de contrôler $E(\Theta, [\![1; n]\!] \setminus \mathcal{C})$. On parle de validation. Le réseau de neurone devra aussi être évaluer en fin d'apprentissage sur un échantillon de test $\{(x'_i, y'_{d_i})|i \in [\![1; n']\!]\}$ différent de l'échantillon d'apprentissage.

1.3.2 L'apprentissage non-supervisé

Dans cette situation on ne dispose plus des *labels* $y_{d_i}, i \in [\![1; n]\!]$. Par conséquent, l'erreur s'écrit simplement $E : (\Theta, \mathcal{C}) \to E(\{x_i, y_i)|i \in \mathcal{C}\})$ où on rappelle que $y_i := f(x_i, \Theta)$ est la réponse du réseau au signal d'entrée x_i. Dans ce contexte, l'intérêt de l'apprentissage non-supervisé est la recherche d'une structure dans les données d'entrée : classes, anomalies ou relations entre les variables explicatives par exemple. Dans le cas d'un clustering par exemple, on peut définir l'erreur comme la somme des variances intra-classes. Il est souvent nécessaire, comme en apprentissage supervisé, de recourir à des méthodes de validation et de test.

Dans la section suivante, on présente un dernier type d'apprentissage.

2 L'apprentissage par renforcement

L'apprentissage par renforcement est le troisième principal paradigme d'apprentissage. L'objectif de l'apprentissage par renforcement est de permettre à un réseau de neurones, alors appelé *agent*, d'apprendre une tâche avec la plus grande autonomie possible : jeux de plateau/jeux vidéos (comme le jeu de go, évoqué dans l'introduction), voltige aérienne en hélicoptère, pilotage d'une centrale électrique, management d'un portefeuille, etc...

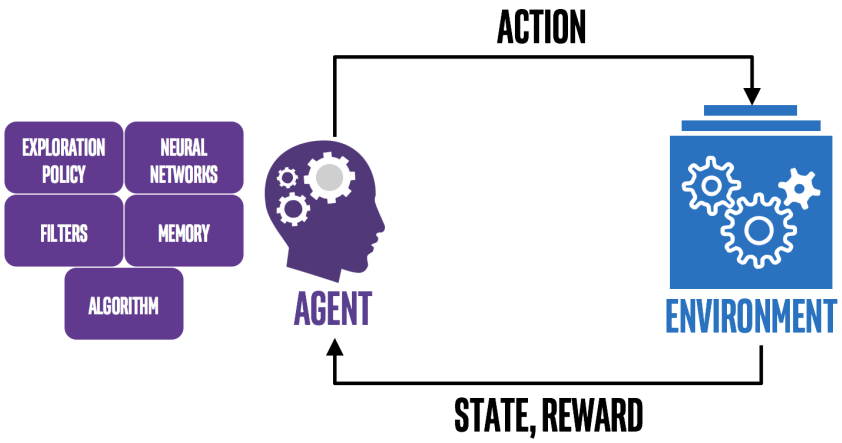
2.1 Différences avec les autres paradigmes d'apprentissage

Plaçons-nous dans la perspective d'une classification pour comprendre pourquoi l'apprentissage par renforcement n'est ni supervisé ni non-supervisé. En apprentissage supervisé, le système connaît la classe d'un individu x_i (durant la phase d'apprentissage dont il est question ici) et le nombre de classe. En apprentissage non-supervisé, le système ne connaît ni la classe d'un individu x_i, ni le nombre de classes. En apprentissage par renforcement, l'agent sait seulement si un individu est bien classé ou pas, sans connaître le nombre de classes.

Les principales différences avec les paradigmes supervisé/non-supervisé sont les suivantes :

— L'apprentissage ne se fait plus par une fonction de perte $E(\Theta)$, mais par un couple *retour := (observation, récompense)* qui dépend de paramètres extérieurs à l'agent.
— Le retour n'est pas instantané ; le temps fait partie du signal d'entrée.
— Les actions de l'agent modifient les données qu'il reçoit par la suite.

FIGURE 3 – L'apprentissage par renforcement



2.2 La notion de récompense

La récompense notée R_t est une variable aléatoire réelle, positive ou négative, qui dépend du temps. C'est une évaluation des performances de l'agent à l'instant t. Le rôle de l'agent est donc de maximiser les récompenses cumulées. Ce principe fondamental de l'apprentissage par renforcement est basé sur l'hypothèse suivante : tout but peut être décrit comme la maximisation de l'espérance des récompenses cumulées. Dans le cas de la voltige aérienne en hélicoptère [4] par exemple, une récompense positive est la réalisation de la figure acrobatique désirée, alors que le crash de l'hélicoptère entraîne une récompense négative et très importante en valeur absolue.

Un des problèmes les plus complexes qui se posent dans le cadre de l'apprentissage par renforcement est le fait que les actions de l'agent peuvent avoir des conséquences éloignées dans le temps. Le fait de sacrifier un gain immédiat peut entraîner l'obtention d'une récompense importante à plus long terme.

2.3 Modélisation du problème

À chaque étape t, l'agent :

— Exécute l'action A_t
— Reçoit l'observation O_t
— Reçoit la récompense R_t

Alors que l'environnement :

— Reçoit l'action A_t
— Émet l'observation O_{t+1}
— Émet la récompense R_{t+1}

L'historique H_t à l'instant t, sur lequel se base l'agent pour choisir son action A_{t+1}, est la séquence : $O_1, R_1, A_1, ..., A_{t-1}, O_t, R_t$. L'enjeu d'un apprentissage par renforcement est de modéliser H_{t+1} afin de déterminer à tout instant t la quantité $\mathbb{E}[R_{t+1}|H_t, A_t]$.

3 Pratique des réseaux de neurones - Présentation du jeu de données

Afin de mettre en pratique les réseaux de neurones, un jeu de données [6] a été choisi sur le site de l'UCI [7], bien connu par la communauté de l'apprentissage automatique. Les données comptent 21 263 individus, 81 variables explicatives et 1 variable à expliquer.

Les individus sont des des molécules décrites par leur formule chimique. Les variables explicatives sont toutes quantitatives, positives et se décomposent en 8 groupes relatifs aux caractéristiques atomiques suivantes :

— Masse atomique (masse des protons et des neutrons)
— Première énergie d'ionisation (énergie nécessaire pour arracher un électron de valence)
— Rayon atomique
— Densité
— Affinité électronique (énergie nécessaire pour ajouter à l'atome neutre)
— Température de fusion
— Conductivité thermique
— Valence (nombre de liaisons formées par l'atome à l'état naturel)

De chacun de ces 8 groupes sont dérivées 10 variables explicatives, par agrégation des valeurs relatives à chaque atome d'une molécule donnée :

— Moyenne simple
— Moyenne simple pondérée
— Moyenne géométrique
— Moyenne géométrique pondérée
— Entropie
— Entropie pondérée
— Étendue
— Étendue pondérée
— Écart-type
— Écart-type pondéré

On obtient ainsi $8 \times 10 = 80$ variables explicatives, parfois très corrélées. La dernière variables explicative est le nombre d'atomes qui composent la molécule.

La variable à expliquer est appelée **température critique**. Elle est également quantitative et positive. Il s'agit de la température, exprimée en degrés kelvin, en-dessous de laquelle une molécule devient superconductrice. La supraconductivité est un phénomène caractérisé par l'absence de résistance électrique, ce qui permet de transporter l'électricité sans perte d'énergie par effet Joule. Elle a été découverte en 1911 par le physicien néerlandais Heike Kamerlingh Onnes. À l'heure actuelle, aucun modèle physique n'a pu établir avec précision le lien entre la température critique d'une molécule et sa formule chimique. L'enjeu est donc ici d'entraîner des réseaux de neurones avec le jeu de données dont on dispose afin de prédire la température critique d'une molécule avec la meilleure précision possible.

Ce jeu de données ainsi que le problème associé ont fait l'objet d'une publication scientifique [8] dans lequel on trouvera davantage d'informations. L'auteur a proposé une solution basée sur l'algorithme XGBoost (eXtreme Gradient Boosting) [9] dont l'erreur *RMSE* (*root-mean-square error*) sur échantillon de test s'élève à 9,5 °K.

4 Le perceptron multicouche avec *R*

4.1 Préparation des données

Tout d'abord, nous procédons à la normalisation des données, afin de supprimer l'influence du choix arbitraire des unités dans lesquelles sont exprimées les grandeurs physiques. Pour cela nous séparons d'abord le jeu de données en un échantillon d'apprentissage et un échantillon de test. La normalisation des deux échantillons est ensuite effectuée en utilisant uniquement la moyenne et l'écart-type de l'échantillon d'apprentissage. De cette manière, aucune information provenant de l'échantillon de test n'est utilisée lors de la construction du modèle [10].

```
# On fixe la graine du generateur aleatoire
# pour assurer la reproductibilite
set.seed(42)

# Import du jeu de donnees
don <- read_csv( file = "~data/data.csv")

# Constitution des echantillons d'apprentissage (75% des donnees)
# et de test (25% des donnees)
index_test <- sample( nrow(don), nrow(don)/4 )
don_train <- don %>% slice( -index_test )
don_test <- don %>% slice( index_test )

# Separation variables explicatives / variable de sortie
don_train_x <- don_train %>% select(-critical_temp)
don_train_y <- don_train %>% pull(critical_temp)
don_test_x <- don_test %>% select(-critical_temp)
don_test_y <- don_test %>% pull(critical_temp)

# Calcul des moyennes / ecart-types des variables explicatives
# sur l'echantillon d'apprentissage
means <- apply( X = don_train_x, MARGIN = 2, FUN = mean )
stdvs <- apply( X = don_train_x, MARGIN = 2, FUN = sd )

# Centrage / reduction des donnees
don_train_x_scaled <- don_train_x %>%
  sweep( MARGIN = 2, STATS = means, FUN = "-" ) %>%
  sweep( MARGIN = 2, STATS = stdvs, FUN = "/" ) %>%
  as.data.frame()

don_test_x_scaled <- don_test_x %>%
  sweep( MARGIN = 2, STATS = means, FUN = "-" ) %>%
  sweep( MARGIN = 2, STATS = stdvs, FUN = "/" ) %>%
  as.data.frame()
```

On prend soin de sauvegarder les données ainsi séparées et normalisées au format .csv afin de pouvoir les utiliser directement avec d'autres interfaces.

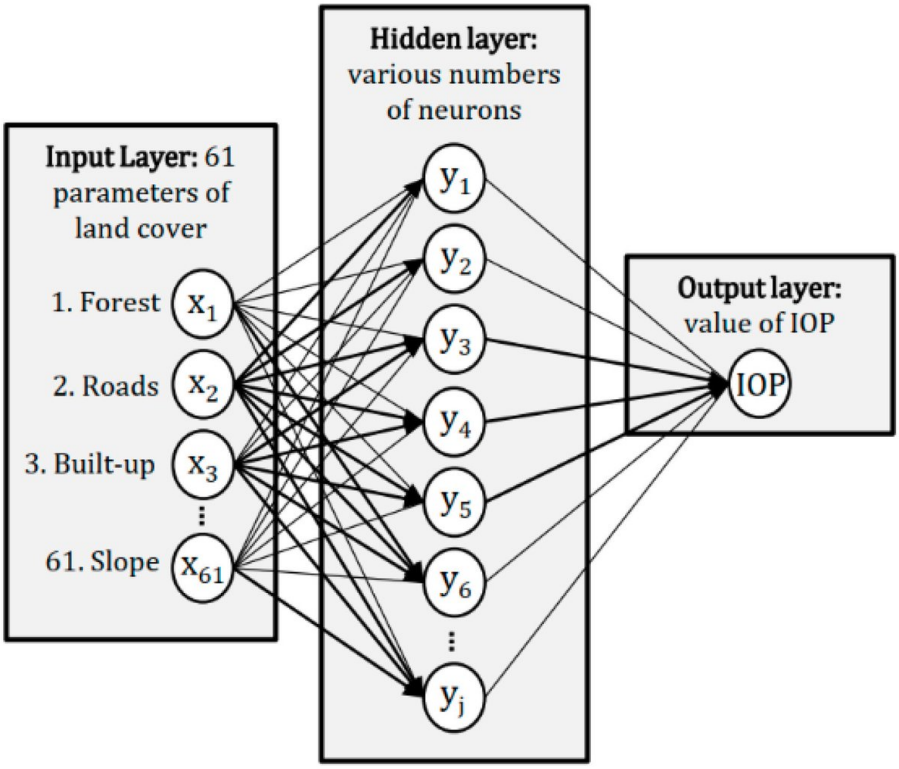
4.2 Parallélisation des calculs

Les réseaux de neurones s'appuient grandement sur l'algèbre linéaire, dont les calculs sont connus pour se prêter à la parallélisation [11]. Par conséquent, le package *doParallel* sera utilisé afin d'exploiter au mieux les quatre coeurs du processeur qui est utilisé dans ce projet.

4.3 Package *nnet*

Ce package *R* implémente le perceptron multicouche à une seule couche cachée.

FIGURE 4 – PMC à une couche cachée



Les paramètres du réseau de neurones qui sont contrôlés sont les suivants :

- *maxit* : nombre maximal d'époques, fixé à 1 000 (défaut : 100)
- *MaxNWts* : nombre maximal de poids, fixé à 10 000 (défaut : 1 000)
- *linout* : pour une variable à expliquer réelle, fixé à TRUE
- *decay* : pas d'apprentissage
- *size* : nombre de neurones dans la couche cachée

Dans le but de déterminer des valeurs adaptées au problème pour les paramètres *decay* et *size*, on recourt au package *caret*. À la manière du package *e1071*, il permet de construire en une seule ligne de commande plusieurs modèles selon une grille de paramètres fournie par l'utilisateur. L'erreur RMSE associée à chaque modèle est évaluée par validation croisée répétée afin de déterminer les paramètres qui la minimisent.

```
set.seed(42)

# Methode de controle pour estimer l'erreur d'un modele,
# et selectionner le meilleur modele
# On utilise la methode "Repeated k-fold validation",
# ici 5-fold validation repetee 3 fois
nn_control <- trainControl(method = "repeatedcv",
                           number = 5,
```

```
                            repeats = 3,
                            allowParallel = TRUE,
                            verboseIter = TRUE,
                            returnData = FALSE
                            )

# Grille des parametres que l'on souhaite tester
# Un modele pour chaque couple (decay, size) sera entraine
nn_grid <- expand.grid(.decay = c(1e-5, 1e-4, 1e-3, 1e-2, 0.1, 1, 10),
                       .size = c(5, 15, 30, 50, 75, 100)
                      )

# Parallelisation des calculs sur 4 coeurs
cluster <- makePSOCKcluster(4)
registerDoParallel(cluster)

# Entrainement des modeles
# Selection du meilleur modele du point de vue RMSE
nn_model <- train(x = don_train_x_scaled,
                  y = don_train_y,
                  method = "nnet",
                  maxit = 1000,
                  MaxNWts = 10000,
                  linout = TRUE,
                  trace = FALSE,
                  verbose = FALSE,
                  metric = "RMSE",
                  trControl = nn_control,
                  tuneGrid = nn_grid
                  )

# Sauvegarde du modele
saveRDS(nn_model, "~/model/nn_model.rds")

# Fin de la parallelisation
stopCluster(cluster)
```

Après environ une semaine de calcul, on appelle l'objet **nn_model** généré par la fonction **caret : :train()** :

```
> nn_model
Neural Network

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 3 times)
Summary of sample sizes: 12760, 12759, 12758, 12758, 12757, 12758, ...
Resampling results across tuning parameters:

  decay  size  RMSE      Rsquared   MAE
  1e-05    5   15.79579  0.7885414  11.027712
  1e-05   15   14.35797  0.8254625   9.825861
  1e-05   30   12.74194  0.8633286   8.565660
  1e-05   50   12.40057  0.8719705   8.162427
```

```
  1e-05   75   13.15038  0.8586011   8.351040
  1e-05  100   13.96290  0.8435332   8.683300
  1e-04    5   15.66137  0.7918441  10.846563
  1e-04   15   13.95057  0.8353748   9.522246
  1e-04   30   12.69400  0.8644975   8.447316
  1e-04   50   12.42646  0.8712986   8.133703
  1e-04   75   13.24571  0.8567670   8.419104
  1e-04  100   13.70104  0.8492301   8.543949
  1e-03    5   15.70431  0.7902644  10.901087
  1e-03   15   14.20744  0.8290784   9.658067
  1e-03   30   12.76828  0.8625183   8.513579
  1e-03   50   12.41871  0.8716674   8.106322
  1e-03   75   13.11666  0.8594194   8.394045
  1e-03  100   13.82398  0.8465442   8.616601
  1e-02    5   14.69466  0.8171692   9.970757
  1e-02   15   13.98463  0.8346373   9.550958
  1e-02   30   12.75389  0.8629944   8.527336
  1e-02   50   12.37803  0.8721640   8.072023
  1e-02   75   13.02108  0.8613004   8.215411
  1e-02  100   13.78220  0.8468775   8.558684
  1e-01    5   14.04808  0.8329538   9.552760
  1e-01   15   13.49144  0.8458663   9.134810
  1e-01   30   12.66509  0.8649104   8.437007
  1e-01   50   12.22231  0.8752987   7.972757
  1e-01   75   12.72878  0.8666858   8.042945
  1e-01  100   13.68678  0.8487455   8.457393
  1e+00    5   13.54296  0.8448299   9.137652
  1e+00   15   12.95842  0.8581729   8.744582
  1e+00   30   12.47244  0.8689497   8.365980
  1e+00   50   12.25228  0.8745492   7.973375
  1e+00   75   12.67962  0.8669216   8.008915
  1e+00  100   13.35097  0.8548600   8.216051
  1e+01    5   13.34483  0.8491759   8.983514
  1e+01   15   12.72701  0.8628583   8.540179
  1e+01   30   12.16661  0.8748381   8.129884
  1e+01   50   11.91736  0.8801590   7.806046
  1e+01   75   11.79010  0.8829364   7.583449
  1e+01  100   11.66709  0.8856981   7.361772

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were size = 100 and decay = 10.
```

Pour chaque couple $(size, decay)$, on observe l'erreur RMSE, l'erreur MAE (mean absolute error) ainsi que le coefficient de détermination R^2 estimés par validation croisée répétée. Ce sont les valeurs les plus extrêmes de la grille de paramètres testée qui obtiennent la plus faible erreur RMSE, à savoir $size = 100$ et $decay = 10$. L'erreur RMSE est estimée à 11,67 °K. Évaluons cette erreur sur l'échantillon de test :

```
> don_pred_test <- predict( nn_model, don_test_x_scaled )
> don_pred_train <- predict( nn_model, don_train_x_scaled )
> metrics_test <- postResample( pred = don_pred_test,
                                obs = don_test_y ) %>%
t() %>%
as_tibble()
```

```
> metrics_train <- postResample( pred = don_pred_train,
                                 obs = don_train_y ) %>%
t() %>%
as_tibble()
> metrics_train
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  7.78    0.949  4.71
> metrics_test
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  11.0    0.895  7.02
```

On synthétise les différentes erreurs RMSE obtenues lors des trois étapes apprentissage/validation/test dans le tableau ci-dessous :

PMC à 1 couche cachée - nnet	Erreur RMSE
Échantillon d'apprentissage	7,78 °K
Estimation par validation croisée répétée	11,67 °K
Échantillon de test	11,0 °K

Le grand nombre d'individus (plus de 20 000) permet une bonne estimation de l'erreur de test dès la phase de validation. Fait suffisamment rare pour être souligné, l'erreur de validation se situe même légèrement au-dessus de l'erreur de test. Les performances obtenues par ce premier modèle sont satisfaisantes au regard de ce qui a été accompli dans la publication [8]. On rappelle que l'erreur RMSE obtenue par l'algorithme XGBoost est 9,5 °K sur échantillon de test.

4.4 Comparaison avec d'autres méthodes

4.4.1 Régression PLS [12]

Les variables explicatives du présent jeu de données sont pour certaines d'entre elles extrêmement corrélées [8], si bien que les hypothèses d'un modèle linéaire ne sont pas clairement vérifiées. Une sélection de variables en amont est possible, mais étant donné le grand nombre de composantes, on choisit d'effectuer une régression Partial Least Squares, adaptée à ce type de problème. Cet algorithme génère des variables artificielles qui sont combinaisons linéaires des variables explicatives, et orthogonales entre elles i.e. totalement décorrélées. Les nouvelles variables sont classées par corrélation décroissante avec la variable à expliquer.

À nouveau, on procède par validation croisée pour sélectionner le meilleur modèle. Dans le cas de la régression PLS, le paramètre qui varie est le nombre de composantes retenues.

```
set.seed(42)

# Parametrage de la methode de validation
caret_control <- trainControl(method = "repeatedcv",
                              number = 10,
                              repeats = 5,
                              verboseIter = TRUE,
                              returnData = FALSE
                              )
```

```
# Entrainement des modeles
# Selection du meilleur modele
pls_model <- train(x = don_train_x_scaled,
                   y = don_train_y,
                   method = "pls",
                   tuneLength = 81,
                   metric = "RMSE",
                   trControl = caret_control
                   )
```

```
> pls_model
Partial Least Squares

15948 samples
   81 predictor

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 5 times)
Summary of sample sizes: 14353, 14353, 14354, 14354, 14352, 14353, ...
Resampling results across tuning parameters:

  ncomp  RMSE      Rsquared   MAE
   1     24.66837  0.4848975  20.14509
   2     22.28450  0.5796308  17.54102
   3     21.33706  0.6146576  16.83523
   4     20.68178  0.6379660  16.34981
   5     20.09587  0.6581981  15.78638
   6     19.59956  0.6748642  15.38482
   7     19.23952  0.6866953  15.04843
   8     19.04821  0.6928955  14.81369
   9     18.86514  0.6987793  14.67248
  10     18.76069  0.7020905  14.49190
  ...
  ...
  ...
  65     17.56240  0.7388956  13.30875
  66     17.56167  0.7389174  13.30951
  67     17.56114  0.7389319  13.31440
  68     17.56072  0.7389443  13.31664
  69     17.56055  0.7389487  13.31490
  70     17.56057  0.7389483  13.31494
  71     17.56058  0.7389482  13.31552
  72     17.56048  0.7389515  13.31501
  73     17.56057  0.7389490  13.31521
  74     17.56054  0.7389500  13.31520
  75     17.56057  0.7389492  13.31527
  76     17.56058  0.7389489  13.31528
  77     17.56052  0.7389504  13.31507
  78     17.56052  0.7389505  13.31497
  79     17.56051  0.7389507  13.31486
  80     17.56050  0.7389511  13.31471

RMSE was used to select the optimal model using the smallest value.
```

```
The final value used for the model was ncomp = 72.

don_pred_test <- predict( pls_model, don_test_x_scaled )
don_pred_train <- predict( pls_model, don_train_x_scaled )

> postResample( pred = don_pred_train,
                obs = don_train_y ) %>%
t() %>%
as_tibble()
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  17.5    0.741  13.3

> postResample( pred = don_pred_test,
                obs = don_test_y ) %>%
t() %>%
as_tibble()
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  17.8    0.724  13.4
```

On donne ci-dessous l'erreur RMSE pour les phases d'apprentissage/validation/test pour la régression PLS :

Régression PLS	Erreur RMSE
Échantillon d'apprentissage	17,5 °K
Estimation par validation croisée répétée	17,56 °K
Échantillon de test	17,8 °K

À nouveau l'erreur est très stable d'une phase à l'autre. Cela est probablement dû à la taille du jeu de données, et au processus de sélection des meilleurs paramètres possibles en vue d'une généralisation. L'avantage de cette méthode est la rapidité de sa mise en oeuvre : la durée du calcul est de quelques minutes sans parallélisation, à comparer à la recherche du réseau de neurones précédent qui a duré une semaine. Mais les performances de la régression PLS sont significativement en dessous de la référence [8] et du perceptron multicouche à une couche cachée du package *nnet*. Étant donné l'absence de modèle physique qui perdure depuis la découverte de la supraconductivité, on peut raisonnablement penser qu'il existe des liens non-linéaires entre la température critique d'une molécule et ses autres caractéristiques.

4.4.2 Forêts aléatoires [13]

L'algorithme des forêts aléatoires est choisi car c'est une méthode d'agrégation d'arbres décisionnels, similaire au modèle XGBoost qui a été mis en oeuvre dans la publication de référence [8]. C'est une méthode dite de *bagging*. D'abord, un grand nombre d'arbres de décision sont construits. Afin d'introduire de la variabilité dans ce processus, chaque arbre de la forêt est construit sur un sous-échantillon bootstrappé de l'échantillon d'apprentissage, et les variables utilisées sont un sous-ensemble aléatoire des variables explicatives de cardinal fixe noté *mtry*. Enfin, pour une entrée donnée la valeur de sortie de la forêt aléatoire est la moyenne des valeurs de sortie de tous les arbres décisionnels (ou le vote majoritaire dans le cas d'une classification).

Quant à l'algorithme XGBoost, il fonctionne comme son nom l'indique par boosting d'arbres de

décision : les arbres sont construits de façon séquentielle et non en parallèle comme dans le cas des forêts aléatoires. Chaque observation mal prédite par l'arbre de l'étape n se voit attribuer un poids avant la construction de l'arbre de l'étape $n + 1$.

L'algorithme des forêts aléatoires a été choisi d'une part car XGBoost a déjà été mis en oeuvre dans la publication [8]. D'autre part, Random Forest est plus aisé à paramétrer : seuls *mtry*, le nombre de variables aléatoirement choisies pour chaque arbre, et *ntree*, le nombre d'arbres à agréger, doivent être indiqués. Le paramétrage d'XGBoost est plus complexe et requiert donc des ressources de calcul importantes.

Dans cet exemple, on conserve le nombre d'arbres par défaut ($ntree = 500$), et on procède à une recherche exhaustive du meilleur paramètre *mtry* par validation croisée. Dans une perspective d'efficacité, on utilise la méthode *Out-of-bag*, spécifique aux algorithmes de bagging. Ainsi on tire parti des individus qui n'appartiennent pas au sous-échantillon bootstrappé lors de la construction de chaque arbre de décision.

```
set.seed(42)

# Parallelisation des calculs sur 4 coeurs
cluster <- makePSOCKcluster(4)
registerDoParallel(cluster)

# Entrainement des modeles
# Selection du meilleur modele du point de vue RMSE
rf_model <- train(x = don_train_x_scaled,
                  y = don_train_y,
                  method = "rf",
                  tuneGrid = expand.grid( .mtry = 1:81 ),
                  metric = "RMSE",
                  trControl = trainControl(method = "oob")
                  )

# Sauvegarde du modele
saveRDS(rf_model, "~/model/rf_model.rds")

# Fin de la parallelisation
stopCluster(cluster)

don_pred_test <- predict( pls_model, don_test_x_scaled )
don_pred_train <- predict( pls_model, don_train_x_scaled )
metrics_test <- postResample( pred = don_pred_test,
                              obs = don_test_y ) %>%
                              t() %>%
                              as_tibble()
metrics_train <- postResample( pred = don_pred_train,
                               obs = don_train_y ) %>%
                               t() %>%
                               as_tibble()

> rf_model
Random Forest

15948 samples
   81 predictor
```

```
No pre-processing
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared
   1    10.026544  0.9148742
   2     9.521059  0.9232411
   3     9.488404  0.9237667
   4     9.471602  0.9240364
   5     9.436002  0.9246064
   6     9.429002  0.9247182
   7     9.427104  0.9247485
   8     9.441568  0.9245174
   9     9.402412  0.9251422
  10     9.420028  0.9248614
  11     9.422060  0.9248290
  12     9.393287  0.9252874
  13     9.400624  0.9251707
  14     9.401203  0.9251615
  15     9.402774  0.9251364
  16     9.393504  0.9252840
  17     9.384387  0.9254289
  18     9.393716  0.9252806
  19     9.405449  0.9250938
  20     9.412966  0.9249741
  21     9.399600  0.9251870
  22     9.382363  0.9254611
  23     9.395413  0.9252536
  24     9.370207  0.9256541
  25     9.408789  0.9250406
  26     9.387396  0.9253811
  27     9.387711  0.9253761
  28     9.390926  0.9253250
  29     9.393949  0.9252769
  30     9.398877  0.9251985
  31     9.393392  0.9252858
  32     9.387775  0.9253751
  33     9.402011  0.9251486
  34     9.387134  0.9253853
  35     9.393165  0.9252894
  36     9.395050  0.9252594
  37     9.393501  0.9252840
  38     9.402227  0.9251451
  39     9.391517  0.9253156
  40     9.409977  0.9250217
  41     9.397167  0.9252257
  42     9.415108  0.9249399
  43     9.399082  0.9251952
  44     9.408890  0.9250390
  45     9.422158  0.9248274
  46     9.407061  0.9250682
  47     9.424601  0.9247885
  48     9.411036  0.9250048
```

```
  49      9.397412  0.9252218
  50      9.413164  0.9249709
  51      9.414450  0.9249504
  52      9.424541  0.9247894
  53      9.438353  0.9245688
  54      9.450138  0.9243803
  55      9.411089  0.9250040
  56      9.420492  0.9248540
  57      9.443177  0.9244917
  58      9.447336  0.9244252
  59      9.429174  0.9247155
  60      9.424525  0.9247897
  61      9.430182  0.9246994
  62      9.437055  0.9245896
  63      9.450888  0.9243683
  64      9.455895  0.9242882
  65      9.447490  0.9244227
  66      9.450494  0.9243746
  67      9.465024  0.9241419
  68      9.465511  0.9241341
  69      9.472978  0.9240144
  70      9.465968  0.9241268
  71      9.470213  0.9240587
  72      9.462352  0.9241847
  73      9.467414  0.9241036
  74      9.472943  0.9240149
  75      9.489463  0.9237497
  76      9.471338  0.9240407
  77      9.477028  0.9239494
  78      9.503335  0.9235266
  79      9.496652  0.9236341
  80      9.520790  0.9232454
  81      9.508323  0.9234463

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 24.

> metrics_train
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  5.24    0.977  2.75

> metrics_test
# A tibble: 1 x 3
   RMSE Rsquared   MAE
  <dbl>    <dbl> <dbl>
1  9.38    0.924  5.35
```

Ce calcul distribué sur quatre coeurs a duré quelques heures. Les performances de ce modèle sont excellentes, équivalentes à celles d'XGBoost obtenues dans la publication [8], voire en deçà. L'erreur RMSE de test s'élève à 9,38 °K lorsque $mtry = 24$. Cette conclusion reste à confirmer, en réitérant la procédure sur d'autres séparations aléatoires du jeu de données en échantillons

d'apprentissages et de test. Néanmoins, le nombre d'individus conséquent confère d'ores et déjà une certaine robustesse aux présentes estimations d'erreurs de tests.

On résume dans le tableau ci-dessous l'erreur RMSE sur échantillon de test obtenue pour chacun des trois modèles, exprimée en °K :

Erreur RMSE	Apprentissage	Validation	Test
Régression PLS	17,5	17,56	17,8
PMC à 1 couche cachée	7,78	11,67	11,0
Forêts aléatoires	5,24	9,37	9,38

5 *Python* et le module *Scikit-learn*

5.1 Scikit-learn

Le module libre *Scikit-learn* est dédié à l'apprentissage automatique en Python. Il est l'initiative de chercheurs de l'INRIA qui continuent de le développer dans une large mesure. Son objectif est d'harmoniser l'utilisation d'un grand nombre d'algorithmes tout en s'appuyant sur les modules scientifiques *NumPy* et *SciPy*. Le module *Scikit-learn* propose l'implémentation d'un perceptron multicouche, que ce soit pour la régression ou la classification : la classe *MLPRegressor* sera utilisée dans cette partie. Contrairement au package *nnet*, le nombre de couches cachées est quelconque. À noter qu'une multitude de packages *R* comme *neuralnet* proposent également cette fonctionnalité.

5.2 *Google Colaboratory*

Dans cette partie ainsi que celle consacrée aux réseaux profonds autres que le PMC, on travaille sur la plateforme *Google Colaboratory*. C'est un service proposé gratuitement par *Google* qui met à disposition des machines munies par défaut d'une installation *Python*. L'utilisation se fait directement dans un navigateur, sans aucune configuration particulière requise. L'intérêt majeur est la puissance de calcul, les machines possèdent en effet des GPUs pour la parallélisation. De plus, le partage de code source est très aisé.

5.3 Implémentation

On utilise directement les données séparées et normalisées sous *R* dans la partie précédente, et sauvegardées au format .csv :

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
import pickle

path = '/content/drive/My Drive/Data/'
train_data = pd.read_csv( path + 'train_data.csv' )
test_data = pd.read_csv( path + 'test_data.csv' )
train_labels = pd.read_csv( path + 'train_labels.csv' )
test_labels = pd.read_csv( path + 'test_labels.csv' )
```

Les paramètres du modèle *MLPRegressor* que l'on contrôle sont les suivants :

- *hidden_layer_sizes* : nombre de couches cachées / neurones. (6, 4) signifie 2 couches cachées, de 6 et 4 neurones.
- *alpha* : coefficient de pénalisation (régularisation L2).
- *tol, n_iter_no_change* : l'apprentissage s'arrête lorsque la diminution de l'erreur est inférieure à *tol* pendant *n_iter_no_change* itérations.
- *activation* : fonction d'activation pour les couches cachées (un seul choix possible).
- *solver* : algorithme d'apprentissage pour la correction des poids.
- *random_state* : graine du générateur aléatoire, pour assurer la reproductibilité.
- *early_stopping* : pour arrêter l'apprentissage avant *max_iter* si les critères de convergence (cf *tol, n_iter_no_change*) sont satisfaits.
- *max_iter* : nombre d'itérations au bout duquel l'apprentissage s'arrête.

Les 4 premiers paramètres listés ci-dessus font l'objet d'une *grid search* : un modèle est construit pour chaque combinaison possible, d'une façon similaire à ce qui a été fait précédemment. L'objectif est de déterminer la combinaison qui minimise l'erreur RMSE estimée par validation croisée. Les autres paramètres sont fixés pour éviter un temps de calcul trop conséquent.

On choisit la fonction d'activation *rectifier* définie par $x \to max(0, x)$ (ici désignée par *relu* pour *rectified linear unit*). Empiriquement, cette fonction d'activation donne de bons résultats [15]. De plus, elle a l'avantage de fournir un nombre positif, tout comme la température critique que l'on souhaite prédire en °K. On choisit d'utiliser le solver *adam* [16], qui est efficace notamment sur de larges jeux de données comme celui qui est étudié ici. On utilise le paramètre *early_stopping* pour stopper l'apprentissage si une convergence est atteinte, du point de vue des paramètres *tol* et *n_iter_no_change*. Cela limite le temps de calcul le cas échéant. Le nombre maximal d'itérations est fixé à 10 000 alors qu'il vaut 200 par défaut. Les paramètres qui ne sont pas décrits dans cette section prennent la valeur par défaut de la classe *MLPRegressor()*.

```
mlp_parameters = {
    'hidden_layer_sizes' : [(50,), (100,),
                            (16, 8), (128,32),
                            (32,32,8),
                            (64,32,32,16),
                            (64, 32, 16, 32, 8),
                            (32, 32, 8, 16, 32, 16, 8)],
    'alpha' : [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 5],
    'tol' : [1e-2, 1e-4, 1e-6],
    'n_iter_no_change' : [10, 30, 50]
}

mlp_gridsearch = GridSearchCV(
    MLPRegressor(activation = 'relu', solver = 'adam', random_state = 42, early_sto
    param_grid = mlp_parameters,
    cv = 5,
    n_jobs = -1,
    scoring = 'neg_mean_squared_error',
    verbose = 1
)
```

Le modèle est initialisé. On procède maintenant à l'apprentissage :

```
mlp_gridsearch.fit(train_data, train_labels.values.ravel())
```

```
Fitting 5 folds for each of 504 candidates, totalling 2520 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed: 10.3min
[Parallel(n_jobs=-1)]: Done 196 tasks      | elapsed: 46.8min
[Parallel(n_jobs=-1)]: Done 446 tasks      | elapsed: 114.3min
[Parallel(n_jobs=-1)]: Done 796 tasks      | elapsed: 200.1min
[Parallel(n_jobs=-1)]: Done 1246 tasks      | elapsed: 310.7min
[Parallel(n_jobs=-1)]: Done 1796 tasks      | elapsed: 450.8min
[Parallel(n_jobs=-1)]: Done 2446 tasks      | elapsed: 610.5min
[Parallel(n_jobs=-1)]: Done 2520 out of 2520 | elapsed: 633.5min finished
```

2520 modèles ont été entraînés pour une durée de calcul de 10 heures 33 minutes et 30 secondes. L'expression "Parallel(n_jobs=-1)" indique que tous les coeurs disponibles ont été mobilisés. La commande ci-dessous donne l'ensemble de paramètres qui minimisent l'erreur RMSE :

```
mlp_gridsearch.best_estimator_
```

```
MLPRegressor(activation='relu', alpha=5, batch_size='auto', beta_1=0.9,
             beta_2=0.999, early_stopping=True, epsilon=1e-08,
             hidden_layer_sizes=(32, 32, 8, 16, 32, 16, 8),
             learning_rate='constant', learning_rate_init=0.001, max_fun=15000,
             max_iter=10000, momentum=0.9, n_iter_no_change=10,
             nesterovs_momentum=True, power_t=0.5, random_state=42,
             shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
             verbose=False, warm_start=False)
```

Ce sont à nouveau des valeurs extrêmes qui fournissent le meilleur modèle : 7 couches cachées et un coefficient de pénalisation égal à 5 (la plus grande valeur testée). La commande suivante donne l'erreur RMSE estimée par validation croisée en fonction des paramètres du modèle :

```
pd.DataFrame(mlp_gridsearch.cv_results_)
```

Une inspection rapide de ce tableau tend à montrer que les paramètres *tol* et *n_ iter_ no_ change* ne semblent pas avoir une grande influence sur la qualité du modèle. Venons-en aux performances du modèle obtenu :

```
# Erreur sur l'echantillon d'apprentissage :
(-mlp_gridsearch.score(train_data, train_labels))**(1/2)
10.77081747278793

# Erreur estimee par validation croisee :
(-mlp_gridsearch.best_score_)**(1/2)
14.418759131856719

# Erreur sur l'echantillon de test :
(-mlp_gridsearch.score(test_data, test_labels))**(1/2)
12.013637382804031
```

Les résultats sont corrects, mais moyens au regard de ce qui a été obtenu précédemment. L'erreur de test est supérieure d'un degré kelvin à celle du PMC à une couche cachée implémentée par le package *nnet* sous R. Ce résultat pourrait très probablement être amélioré en expérimentant sur le nombre de couches cachées et de neurones, la méthode de rétropropagation, la pénalisation, etc...

6 Un autre modèle de réseau de neurones profond : les réseaux RBF

6.1 Introduction

Les réseaux de neurones RBF sont similaires aux perceptrons multicouches. La principale différence est qu'ils possèdent au moins une couche cachée RBF pour Radial Basis Function. Une fonction RBF est une fonction $\varphi : \mathbb{R}^d \to \mathbb{R}$, $d \in \mathbb{N}$, qui ne dépend que de la distance du vecteur $\boldsymbol{x}$ à un point fixé $\boldsymbol{c} \in \mathbb{R}^d$: $\forall \boldsymbol{x} \in \mathbb{R}^d, \varphi(\boldsymbol{x}) = \phi(\| \boldsymbol{x} - \boldsymbol{c} \|)$.

Dans une couche RBF, le signal d'entrée des neurones est une RBF. En reprenant les notations de l'introduction, le **signal d'entrée** du neurone j à l'instant t s'écrit $in_j(t) := in(W_j, A(t)) = \varphi(\| A(t) - W_j \|)$. Les poids W_j correspondent donc aux coordonnées du centre associé à la RBF du neurone j. Ils peuvent être initialisés aléatoirement, par la méthode des K-moyennes, les cartes de Kohonen ou d'autres algorithmes d'apprentissage non-supervisés. Dans le cas présent, les centres seront initialisés par la méthode des K-moyennes.

Classiquement, la fonction d'activation d'un neurone RBF est gaussienne, du type $\boldsymbol{x} \to e^{-\frac{\varphi(\boldsymbol{x})}{\sigma}}$.

6.2 Implémentation d'un réseau profond RBF

Dans cette partie, on s'appuie sur l'interface de programmation *Tensorflow 2.0*. C'est un outil open-source d'apprentissage automatique développé par *Google*. C'est un outil plus puissant et plus évolutif que *Scikit-learn* à plusieurs égards, en particulier dans le domaine du *Deep Learning*. En effet, la conception du réseau de neurones est complètement libre grâce à la bibliothèque intégrée *Keras*, développée par François Chollet [20] : nombre de couches, de neurones, fonctions d'activation, signal d'entrée, etc... Les types de couches les plus courantes sont déjà implémentées (couche classique, convolutive, récurrente...) et l'intégration d'une couche "maison" se fait sans difficulté. La couche RBF n'étant pas nativement disponible dans Keras, nous avons eu recours à l'implémentation proposée par Petra Vidnerová [17] :

```
# Installation de Tensorflow 2.0
!pip install -q git+https://github.com/tensorflow/docs

Building wheel for tensorflow-docs (setup.py) ... done

# Import des modules necessaires
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_squared_error

import tensorflow as tf
import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.initializers import RandomUniform, Initializer, Constant

# Verifions que Tensorflow est bien installe en version 2.0
print(tf.__version__)
```

```
2.2.0-rc2
```

```
# Couche RBF [17]

from keras.initializers import Initializer
from sklearn.cluster import KMeans

class InitCentersRandom(Initializer):
    """ Initializer for initialization of centers of RBF network
        as random samples from the given data set.
    # Arguments
        X: matrix, dataset to choose the centers from (random rows
          are taken as centers)
    """

    def __init__(self, X):
        self.X = X
        super().__init__()

    def __call__(self, shape, dtype=None):
        assert shape[1:] == self.X.shape[1:]  # check dimension

        # np.random.randint returns ints from [low, high) !
        idx = np.random.randint(self.X.shape[0], size=shape[0])

        return self.X[idx, :]

class InitCentersKMeans(Initializer):
    """ Initializer for initialization of centers of RBF network
        by clustering the given data set.
    # Arguments
        X: matrix, dataset
    """

    def __init__(self, X, max_iter=100):
        self.X = X
        self.max_iter = max_iter

    def __call__(self, shape, dtype=None):
        assert shape[1] == self.X.shape[1]

        n_centers = shape[0]
        km = KMeans(n_clusters=n_centers, max_iter=self.max_iter, verbose=0)
        km.fit(self.X)
        return km.cluster_centers_
```

```
class RBFLayer(tf.keras.layers.Layer):
    """ Layer of Gaussian RBF units.
    # Example
    ```python
        model = Sequential()
        model.add(RBFLayer(10,
                           initializer=InitCentersRandom(X),
                           betas=1.0,
                           input_shape=(1,)))
        model.add(Dense(1))
    ```
    # Arguments
        output_dim: number of hidden units (i.e. number of outputs of the
                    layer)
        initializer: instance of initiliazer to initialize centers
        betas: float, initial value for betas
    """

    def __init__(self, output_dim, initializer=None, betas=1.0, **kwargs):

        self.output_dim = output_dim

        # betas is either initializer object or float
        if isinstance(betas, Initializer):
            self.betas_initializer = betas
        else:
            self.betas_initializer = Constant(value=betas)

        self.initializer = initializer if initializer else RandomUniform(
            0.0, 1.0)

        super().__init__(**kwargs)

    def build(self, input_shape):

        self.centers = self.add_weight(name='centers',
                                       shape=(self.output_dim, input_shape[1]),
                                       initializer=self.initializer,
                                       trainable=True)
        self.betas = self.add_weight(name='betas',
                                     shape=(self.output_dim,),
                                     initializer=self.betas_initializer,
                                     # initializer='ones',
                                     trainable=True)

        super().build(input_shape)

    def call(self, x):

        C = tf.expand_dims(self.centers, -1) # inserts a dimension of 1
        H = tf.transpose(C-tf.transpose(x)) # matrix of differences
        return tf.exp(-self.betas * tf.math.reduce_sum(H**2, axis=1))
```

```
    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)

    def get_config(self):
        # have to define get_config to be able to use model_from_json
        config = {
            'output_dim': self.output_dim
        }
        base_config = super().get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

Ensuite, on importe les données et on initialise le réseau de neurones RBF :

```
# Import des donnees
path = '/content/drive/My Drive/Data/'
train_data = pd.read_csv( path + 'train_data.csv' )
test_data = pd.read_csv( path + 'test_data.csv' )
train_labels = pd.read_csv( path + 'train_labels.csv' )
test_labels = pd.read_csv( path + 'test_labels.csv' )

# On convertit les donnees dans un format (numpy.array) compatible
# avec la classe RBFLayer implementee ci-dessus
train_data_array = np.array(train_data)
test_data_array = np.array(test_data)

from numpy.random import seed
# Initialisation de la graine du generateur aleatoire
seed(42)

# Initialisation du reseau de neurones RBF
model_rbf = keras.Sequential()

# Initialisation de la couche RBF,
# constituee de 64 neurones RBF,
# prenant les 81 variables expicatives en entree
rbflayer = RBFLayer(output_dim = 64,
                    initializer = InitCentersKMeans(train_data_array),
                    betas=2.0, input_shape=(81,)
                    )

# Ajout des differentes couches du reseau profond
model_rbf.add(rbflayer)
model_rbf.add(layers.Dense(32, activation = 'relu'))
model_rbf.add(layers.Dense(16, activation = 'relu'))
# La derniere couche a un output de dimension 1
model_rbf.add(layers.Dense(1, activation = 'relu'))

# Initialisation de l'optimisateur
optimizer = tf.keras.optimizers.RMSprop(0.001)

# La metrique pour l'apprentissage est l'erreur MSE
model_rbf.compile( loss='mean_squared_error',
                   optimizer=optimizer,
```

```
                  metrics='mean_squared_error')

print(model_rbf.summary())

Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
rbf_layer_2 (RBFLayer)       (None, 64)                5248
_________________________________________________________________
dense_10 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_11 (Dense)             (None, 16)                528
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 17
=================================================================
Total params: 7,873
Trainable params: 7,873
Non-trainable params: 0
_________________________________________________________________
None
```

Le réseau de neurone profond commence avec une couche RBF de taille 64, puis trois couches de 64, 32 et 1 neurones dont la fonction d'activation est *rectifier*, explicitée précédemment. On effectue maintenant l'apprentissage. On fixe le nombre d'itérations (*epochs*) à 1000, avec arrêt prématuré si l'erreur de validation ne diminue pas au bout de 5 époques.

```
seed(42)

early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

# Fit du modele avec les donnees
history = model_rbf.fit(train_data_array,
                        train_labels,
                        epochs=1000,
                        validation_split = 0.2,
                        verbose=1,
                        callbacks=[early_stop, tfdocs.modeling.EpochDots()])

loss_train, mean_squared_error_train = model_rbf.evaluate(train_data_array,
                                                          train_labels,
                                                          verbose=2)
print("RMSE train : {:5.2f}".format(
mean_squared_error_train**(1/2))
)

499/499 - 1s - loss: 223.5224 - mean_squared_error: 223.7690
RMSE train : 14.96

loss_test, mean_squared_error_test = model_rbf.evaluate(test_data_array,
                                                        test_labels,
                                                        verbose=2)
print("RMSE test : {:5.2f}".format(
mean_squared_error_test**(1/2))
```

```
)

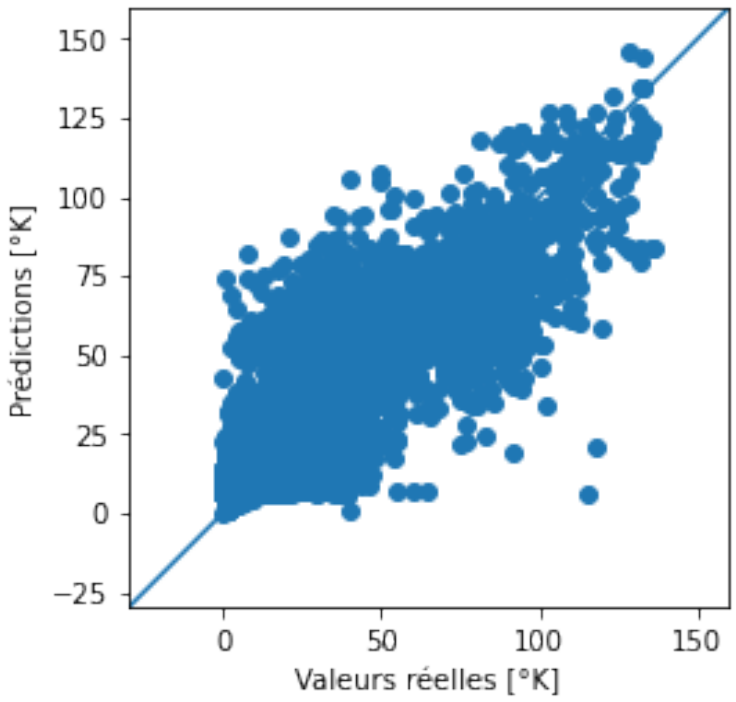
167/167 - 0s - loss: 228.1039 - mean_squared_error: 228.7108
RMSE test : 15.12
```

Le calcul s'arrête au bout de 46 époques. Comme on peut le voir ci-dessus, l'erreur RMSE de test se situe aux alentours de 15 °K. On représente ci-dessous les valeurs prédites par le réseau RBF, ainsi que le distribution de l'erreur de prédiction :

```
test_predictions = model_rbf.predict(test_data).flatten()

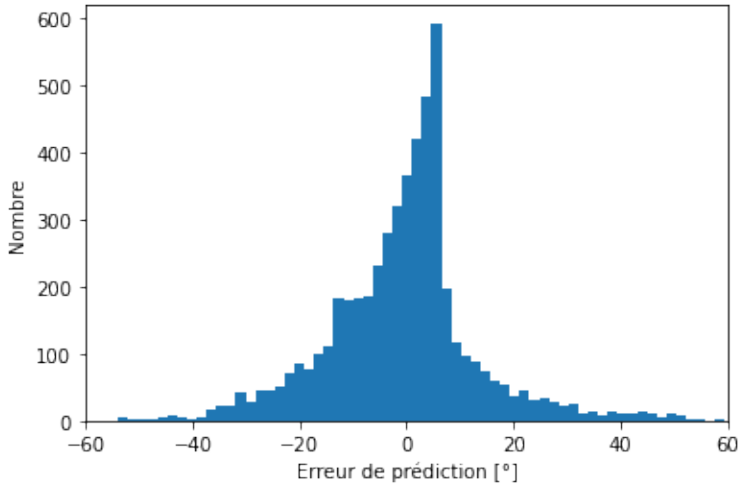
a = plt.axes(aspect='equal')
plt.scatter(test_labels, test_predictions)
plt.xlabel('Valeurs reelles [K]')
plt.ylabel('Predictions [K]')
lims = [-30, 160]
plt.xlim(lims)
plt.ylim(lims)
_ = plt.plot(lims, lims)
```

FIGURE 5 – Températures critiques réelles/prédites




```
error = test_predictions - np.array(test_labels).flatten()
plt.hist(error, bins = 100)
plt.xlim([-60,60])
plt.xlabel("Erreur de prediction")
_ = plt.ylabel("Nombre")
```

FIGURE 6 – Distribution de l'erreur de prédiction



On constate que le système a tendance à sous-estimer la température critique des molécules.

Finalement, les performances du réseau de neurones RBF profond sont moyennes au regard de ce qui a été fait précédemment, bien qu'acceptables. Il est possible qu'une couche RBF en entrée du réseau ne soit pas le bon choix pour le problème de la température critique d'une molécule, ou que l'architecture du réseau profond ne soit pas adaptée. Néanmoins, cette partie dédiée à la pratique de l'outil *Tensorflow* et plus spécifiquement de la bibliothèque *Keras* laisse entrevoir la diversité des architectures permises par cette plateforme.

7 Cartes de Kohonen / SOMs

7.1 Présentation

Les cartes de Kohonen ou SOMs (*Self-Organizing Maps*) [18] sont des réseaux de neurones entraînés par apprentissage non-supervisé. L'objectif est de générer une représentation plane de données d'un espace de grande dimension, à la manière de l'Analyse en Composantes Principale. Dans cette perspective, les neurones du réseau sont représentés sous la forme d'une grille rectangulaire ou hexagonale. Le nombre de neurones et la forme de la grille doivent être déterminés en amont par l'utilisateur, comme le nombre de classes avec les K-moyennes par exemple.

Une fois que la taille de la carte de Kohonen a été définie, chaque neurone se voit attribuer un vecteur de poids appartenant à l'espace des données. Dans le cas des données de supraconduction, cet espace est $\mathbb{R}^{82}$ en incluant la température critique (la variable à expliquer). Cette première étape peut être effectuée aléatoirement ou par d'autres méthodes qui exploitent par exemple les vecteurs propres de la SVD (*singular value decomposition*) des données. Ensuite, les poids des neurones sont ajustés avec l'échantillon d'apprentissage. Pour chaque individu de l'échantillon passé au réseau en signal d'entrée, un neurone BMU (*Best Matching Unit*) est identifié : c'est le neurone dont le poids est le plus proche de l'individu du point de vue de la distance euclidienne dans $\mathbb{R}^{82}$. Le poids du neurone BMU et des neurones adjacents sont alors modifiés de façon à se rapprocher du signal d'entrée. Plusieurs paramètres de convergence rendent l'ajustement de plus en plus faible au fur et à mesure que le nombre d'individus passés au réseau augmente, de façon à ce que la représentation fournie par la carte de Kohonen se stabilise.

Une fois l'apprentissage terminé, il est possible de représenter de nouveaux individus dans la carte. Il est important de noter que les SOMs conservent la topologie [19] : si des individus sont proches dans l'espace initial, alors ils sont proches dans la SOM. Par ailleurs, l'identification du neurone BMU fait des SOMs un algorithme d'apprentissage compétitif.

7.2 Application au jeu de données sur la température critique de supraconduction

On réalise une SOM à partir des données normalisées dans le but de supprimer le biais induit par les unités dans lesquelles sont exprimées les différentes variables. Dans le but de favoriser la lecture et l'analyse, on ne retient que les 10 variables suivantes : moyennes pondérées des huit caractéristiques atomiques relevées (cf partie 3 - Présentation du jeu de données), nombre d'atomes et température critique. On utilise le package *R kohonen* disponible sur le CRAN.

```
set.seed(42)
# Fonction de normalisation
scale2 <- function(x, na.rm = FALSE){
    (x - mean(x, na.rm = na.rm)) / sd(x, na.rm)
}

# On travaille avec des donnees normalisees pour construire la SOM
don_som <- don %>%
  select_at(
    vars(number_of_elements,
         starts_with("wtd_mean"),
         critical_temp
         )
  ) %>%
  mutate_all( scale2, na.rm = TRUE )

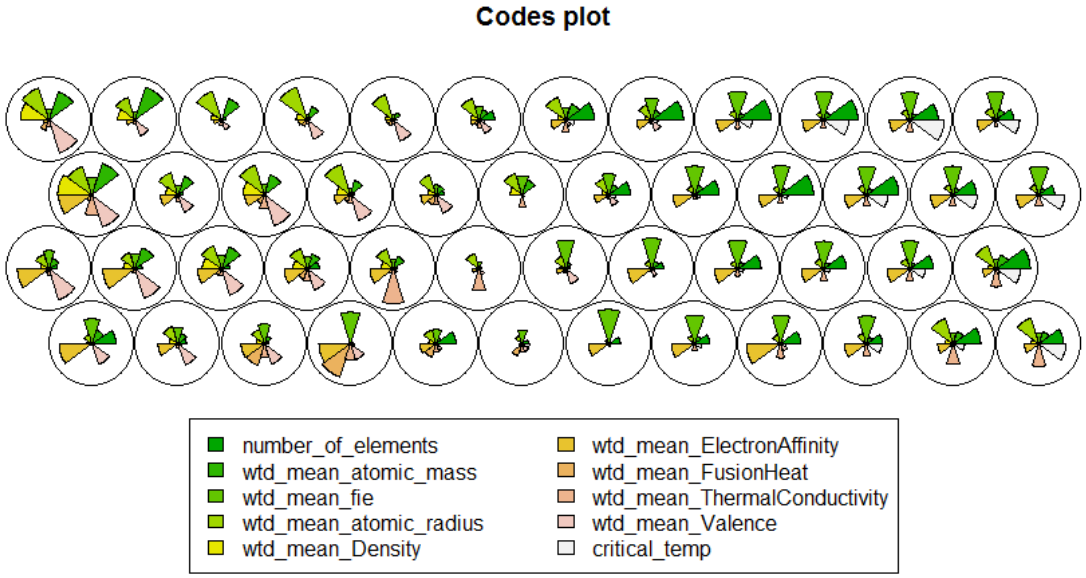
# Determinons d'abord la taille de la carte.
# Pour cela on utilise le rapport entre les deux premieres valeurs propres
# de la SVD des donnees :

library(FactoMineR)
acp <- don_som %>% PCA(ncp = 2, scale.unit = TRUE, graph = FALSE )
ratio <- signif(acp$eig[1,1] / acp$eig[2,1], 1)
width <- 5

# Entrainement de la SOM
som <- som( data.matrix(don_som),
            grid = somgrid( ratio * width, width, "hexagonal")
          )
```

Dans la représentation graphique ci-dessous, on représente les vecteurs de poids associés à chaque neurone de la SOM. Leurs coordonnées sont représentées par des secteurs angulaires : plus le rayon est grand, plus la coordonnée selon la variable concernée est importante.

```
plot(som, type =  c( "codes" ) )
```



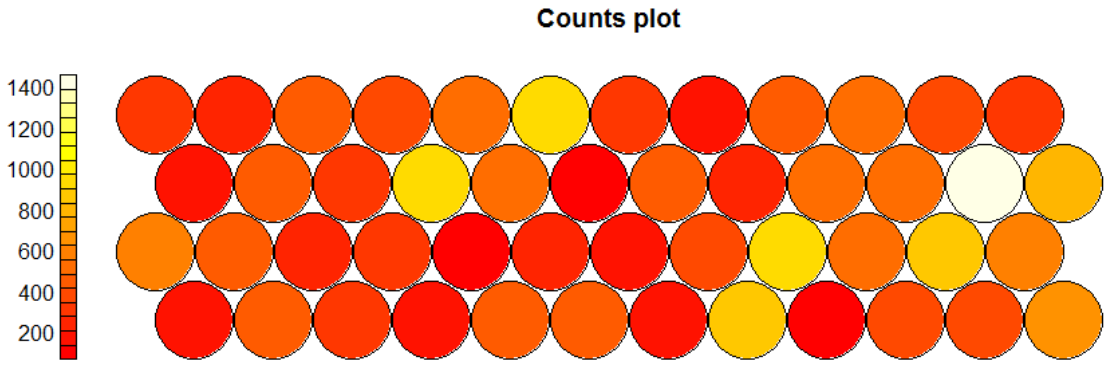
On constate que les poids des neurones situés dans la partie gauche de la carte ont des coordonnées particulièrement importantes selon les variables suivantes : masse atomique, rayon atomique et densité. Cela semble cohérent puisqu'un rayon élevé implique intuitivement une masse et une densité importantes. La partie gauche supérieure présente une valence élevée, tandis que la partie inférieure présente une affinité électronique importante.

Dans la partie droite de la carte, les poids des neurones sont caractérisés par une température critique élevée et un nombre d'éléments important, parfois accompagnés par une grande affinité électronique. La zone Sud-Est est caractérisée par une conductivité thermique significative.

Enfin on remarque que le poids du neurone (6,1) i.e en 6 en "abscisse" / 1 en "ordonnée" est proche du vecteur nul.

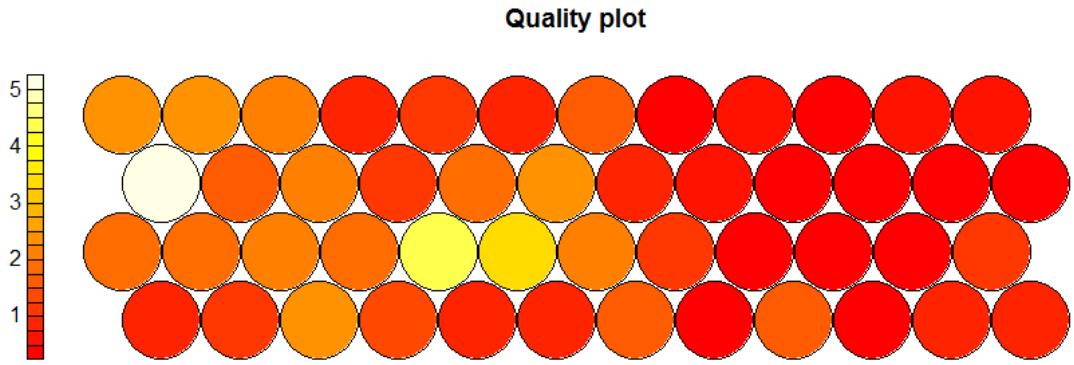
Pour un individu (ici une molécule) donné, le neurone associé est celui dont le poids est le plus proche au sens de la distance euclidienne dans $\mathbb{R}^{82}$. On peut alors représenter le nombre d'individus associés à chaque neurone pour évaluer l'homogénéité de la SOM :

```
plot(som, type =  c( "counts" ) )
```



Dans le cas présent, la SOM est relativement homogène, hormis quelques neurones qui comportent un nombre d'individus particulièrement important. Un autre paramètre est la distance moyenne entre les individus associés à un neurone et son poids. Plus distance est faible, plus la représentation des individus par leur neurone est de bonne qualité. Cette notion est similaire à la variance intra-classe pour l'algorithme des K-moyennes par exemple.

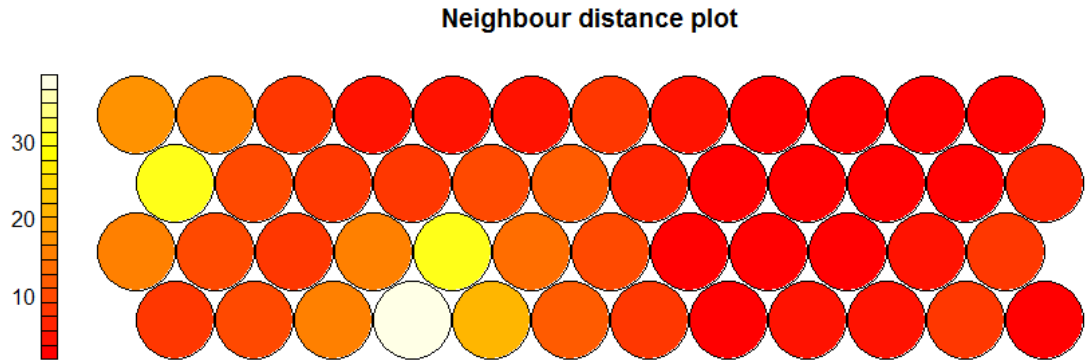
```
plot(som, type =  c( "quality" ) )
```



Ici la qualité de la SOM est plutôt satisfaisante, hormis pour les neurones (1,3) et (5,2). Empiriquement, on constate que les neurones de moindre qualité ont tendance à se situer en bordure de la SOM. Une des raisons avancées par la littérature est le fait qu'ils agrègent des individus loins de tous les neurones, i.e pas particulièrement proches du poids de leur neurone.

La figure suivante représente la distance moyenne des neurones aux neurones adjacents.

```
plot(som, type =  c( "dist.neighbours" ) )
```



On distingue deux clusters à gauche et en bas de la SOM. En comparant cette représentation au premier graphique qui a été présenté, on observe que le premier cluster est celui des masses atomiques, rayons atomiques et densités élevées. Le second cluster correspond aux molécules dont les caractéristiques physiques présentent de faibles valeurs.

8 Conclusion

Les réseaux de neurones forment un domaine de recherche très actif, caractérisé par une profusion d'architectures et d'algorithmes différents. Ils peuvent ainsi être conçus et paramétrés pour résoudre une grande variété de problèmes : réseaux à convolution pour la reconnaissance d'images [24], GANs (*Generative Adversarial Network*) pour la création d'images photoréalistes [25], réseaux récurrents pour la reconnaissance vocale [26], apprentissage par renforcement pour la réalisation de tâches complexes, etc... Les possibilités sont innombrables et constituent à la fois un avantage et un inconvénient : les résultats sont potentiellement excellents, à condition d'effectuer un travail de conception / paramétrage conséquent et spécifique au problème à résoudre.

Cela a pu être constaté pour la prédiction de la température critique de composés chimiques. Les performances des différents réseaux de neurones implémentés ont été correctes mais en-deçà de celles d'autres algorithmes simples à paramétrer et moins gourmands en ressources de calcul. L'objectif était l'expérimentation avec plusieurs interfaces, davantage que la recherche d'un (hyper)-paramétrage optimal. Néanmoins, on peut penser que des résultats de grandes qualité peuvent être accomplis avec l'architecture idoine.

Le présent rapport a également permis de constater la diversité des outils consacrés aux réseaux de neurones. Les bibliothèques *Tensorflow/Keras*, déployées en *Python* sur la plateforme *Google Colaboratory* ont particulièrement été appréciées pour leur facilité d'usage, la diversité des configurations de réseaux possibles et la puissance de calcul disponible.

Références

[1] Greg Kohs (29/09/2017). *AlphaGo.*
https ://youtu.be/WXuK6gekU1Y

[2] Vincent, James (27/11/2019). "Former Go champion beaten by DeepMind retires after declaring AI invincible". *The Verge.*

[3] Alberto Quesada, *neuraldesigner.*
https ://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network

[4] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. *In S. Thrun, L. Saul, and B. Schoelkopf (Eds.), Advances in Neural Information Processing Systems (NIPS) 17, 2004.*
Autonomous helicopter flight via reinforcement learning.

[5] David Silver. *UCL Course on RL*
https ://www.davidsilver.uk/teaching/

[6] Superconductivty Data Set
http ://archive.ics.uci.edu/ml/datasets/Superconductivty+Data

[7] Dua, D. and Graff, C. (2020). *Irvine, CA : University of California, School of Information and Computer Science.*
UCI Machine Learning Repository [http ://archive.ics.uci.edu/ml]

[8] Hamidieh, Kam, A data-driven statistical model for predicting the critical temperature of a superconductor,
Computational Materials Science, Volume 154, November 2018, Pages 346-354
https ://arxiv.org/pdf/1803.10260.pdf

[9] Chen, T. and Guestrin, C. (2016). Xgboost : A scalable tree boosting system.
https ://arxiv.org/abs/1603.02754

[10] Sebastian Raschka, Why do we need to re-use training parameters to transform test data ?
https ://sebastianraschka.com/faq/docs/scale-training-test.html

[11] Richard P. Brent, 1991, Parallel Algorithms in Linear Algebra. *Australian National University*
https ://arxiv.org/pdf/1004.5437.pdf

[12] Pierre-André Cornillon, Eric Matzner-Løber, 2011. Régression avec R. *Springer*

[13] Robin Genuer, Jean-Michel Poggi, 2019. Les forêts aléatoires avec R. *Presses Universitaires de Rennes*

[14] François Husson, 2018. R pour la statistique et la science des données. *Presses Universitaires de Rennes*

[15] Dabal Pedamonti, 2018. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. *Department of Computer Science, University of Edinburgh*
https ://arxiv.org/pdf/1804.02763.pdf

[16] Diederik P. Kingma, Jimmy Lei Ba, 2017. ADAM : A Method For Stochastic Optimization. *University of Amsterdam, OpenAI & University of Toronto*
https ://arxiv.org/pdf/1412.6980.pdf

[17] Petra Vidnerová, 2019. RBF-Keras : an RBF Layer for the Keras Library.
The Czech Academy of Sciences, Institute of Computer Science
https ://github.com/PetraVidnerova/rbf_keras

[18] Marie Cottrell, Madalina Olteanu, Fabrice Rossi, Nathalie Villa-Vialaneix, 2018. Self-Organizing Maps, theory and applications. *Revista de Investigacion Operacional, 2018, 39 (1), pp.1-22.* ffhal-01796059f https ://hal.archives-ouvertes.fr/hal-01796059/document

[19] Th. Villmann, R. Der, M. Herrmann, Th. Martinetz, 1997. Topology Preservation in Self-Organizing Feature Maps : General Definition and Efficient Measurement.

[20] François Chollet, 2017. Deep Learning with Python. *Manning*

[21] Ben Kröse, Patrick van der Smagt, 2017. An Introduction to Neural Networks. *University of Amsterdam*

[22] Trevor Hastie, Robert Tibshirani, Jerome Friedman, 2008. The Elements of Statistical Learning. *Springer*

[23] Ian Goodfellow, Yoshua Bengio, Aaron Courville, 2016. Deep Learning. *MIT Press.*
https ://www.deeplearningbook.org/

[24] Linda Wang and Alexander Wong, 2020. COVID-Net : A Tailored Deep Convolutional Neural Network Design for Detection of COVID-19 Cases from Chest Radiography Images. *University of Waterloo, Waterloo Artificial Intelligence Institute, DarwinAI Corp., Canada.*
https ://arxiv.org/pdf/2003.09871.pdf

[25] Christian Ledig *et al*, 2017. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.
https ://arxiv.org/pdf/1609.04802.pdf

[26] Takaki Makino, Hank Liao *et al*, 2019. Recurrent Neural Network Transducer for Audio-Visual Speech Recognition. *Google Inc., DeepMind.*
https ://arxiv.org/pdf/1911.04890.pdf