

Projet de Réseau de Neurone

Douba JAFUNO

15 avril 2020

SOMMAIRE

Présentation succincte des réseaux de neurone	2
Introduction	2
Définitions.....	2
Structure d'un Neurone Artificiel	2
Exemples de fonction d'activation	3
Apprentissage	4
Règle d'apprentissage supervisé	4
Règle d'apprentissage non-supervisé.....	5
Démonstration logicielle utilisant un Perceptron (Multicouche).....	6
Prédiction sur une série temporelle	6
Création du "dataframe avec les données de la série décalée"	6
Comparaison du PMC avec d'autre méthode d'apprentissage	7
PMC	7
Régression Linéaire Multiple	8
SVM	9
Random Forest	10
Prévisions et Conclusion	10
Démonstration logicielle utilisant les réseaux de neurones avec Scikit-Learn de Python	11
Création d'un jeu de donnée de classification	11
Séparation pour notre modèle	12
Mise en place du modèle PMC	12
Entraînement du modèle.....	12
Prédiction et Conclusion.....	12
Démonstration logicielle utilisant les cartes de Kohonen	13
Présentation.....	13
Principe Générale.....	13
Apprentissage	14
Sur la notion de voisinage.....	14
Les cartes de Kohonen avec R (package « Kohonen ») sur les données Iris	15
Construction du modèle de la carte avec Supersom	15
Différents types de carte	15
Clustering et segmentation au-dessus de la carte auto-organisée.....	18
Nombre optimal de Cluster	18
Clustering des nœuds.....	19
Prédictions.....	19
Démonstration logicielle d'un réseau de neurone profond RNN avec Keras et Tensorflow	21
Présentation des RNNs.....	21
Présentation et Préparation des données MNIST	22
Mise en place du RNN	24
Entraînement et Score	25
Quelques Prédiction.....	27
Apprentissage par Renforcement.....	28
Introduction	28
Aperçu de l'apprentissage du renforcement	28
Définition : Processus de décision Markov (fini)	29
Définition : Fonction de valeur d'état et fonction de valeur d'action :	30
Politique optimale	30
Méthode de Monte Carlo	31
Technique d'apprentissage par renforcement :	32
SARSA.....	32
Q-Learning.....	32
Conclusion + Références.....	33

Présentation succincte des réseaux de neurone

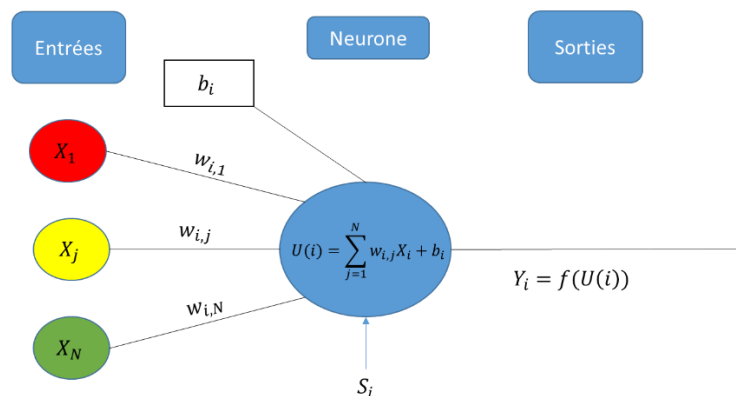
Introduction

Un « réseau neuronal » est un modèle mathématique appelé neurone artificiel qui représente les cellules neuronales (neurones) du cerveau humain et leurs connexions. Ces dernières années, le domaine de l'intelligence artificielle (IA) est en plein essor ont le trouve dans de nombreux domaine de notre quotidien selon plusieurs experts il s'agirait tout simplement de l'avenir. Les réseaux de neurones et l'apprentissage en profondeur sont de grands sujets en informatique et dans l'industrie des technologies, ils fournissent actuellement les meilleures solutions à de nombreux problèmes de reconnaissance d'image, de reconnaissance vocale et de traitement du langage naturel par exemple tout au long de ce projet nous verrons des exemples d'application des réseaux de neurone particulièrement en classification.

Définitions

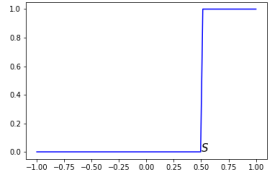
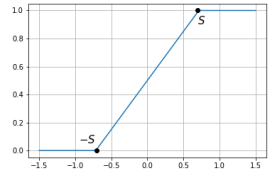
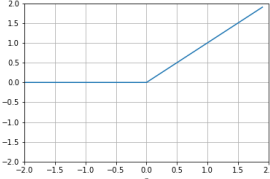
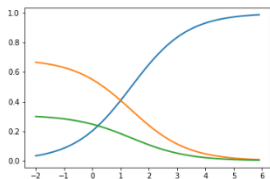
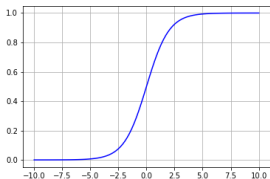
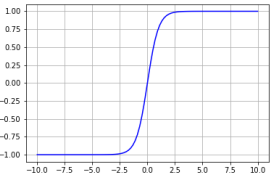
Un réseau de neurones est un assemblage interconnecté d'éléments, un graphe orienté pondéré, d'unités ou de nœuds (que l'on qualifie de neurone artificiel, modélisant mathématiquement le comportement d'un neurone biologique), qui réalise des traitements simples, structurées sous forme de couches successives interconnectées capables de recevoir et d'échanger des informations via des liens structurés.

Structure d'un Neurone Artificiel



Un neurone artificiel i est représenté par une fonction $U(i)$ de l'entrée $X = (X_1, \dots, X_N)$ avec les X_i des valeurs numériques, pondérée par un vecteur de poids de connexion $w_i = (w_{i,1}, \dots, w_{i,N})$ correspondant à la i ème ligne de la matrice $W = w_{i,j} \quad 1 \leq i, j \leq N$ ou $w_{i,j}$ est le poids de connexion du neurone j vers le neurone i , complétée par un biais b_i , et associée à une fonction d'activation f éventuellement muni d'un seuil S_i , et tel que nous avons en sortie $Y_i = f(\langle w_i, X \rangle + b_i)$.

Exemples de fonction d'activation

<p>Pas unitaire</p> $f(x) = \begin{cases} 0, & \text{si } x < S \\ 1, & \text{si } S \leq x \end{cases}$	
<p>Linéaire Seuillée</p> $f(x) = \begin{cases} 0, & \text{si } x < -S \\ m \cdot x + b, & \text{si } -S \leq x < S \\ 1, & \text{si } S \leq x \end{cases}$	
<p>Relu</p> $f(x) = \begin{cases} 0, & \text{si } x < S \\ x, & \text{si } S \leq x \end{cases}$	
<p>Softmax (pour 3 vecteurs)</p> $\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$ <p>Pour $x = (x_1, \dots, x_K)$</p> <p>Softmax en sort un vecteur de K nombres réels strictement positifs et de somme 1.</p>	
<p>Sigmoïde</p> $f(x) = \frac{1}{e^{-x} + 1}$	
<p>Tanh</p> $f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$	

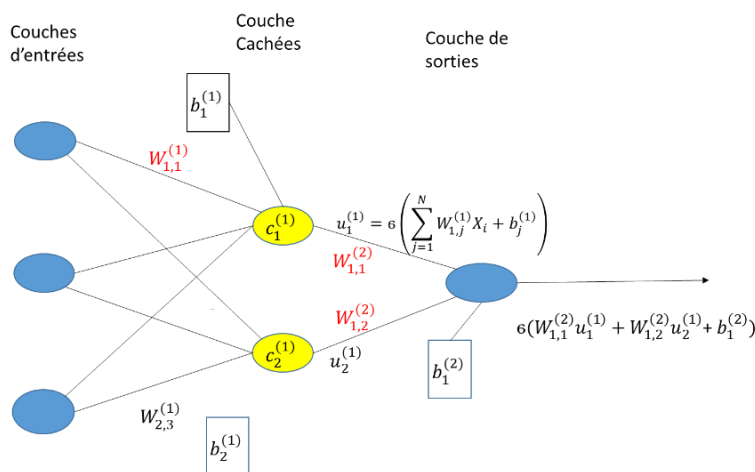
Apprentissage

La règle d'apprentissage ou le processus d'apprentissage d'un réseau de neurones artificiels est une méthode, mathématique ou un algorithme qui améliore les performances et / ou le temps de formation du réseau. Habituellement, cette règle est appliquée à plusieurs reprises sur le réseau. Cela se fait en mettant à jour les poids de connexion via la matrice W et les niveaux de biais d'un réseau lorsque celui-ci est simulé dans un environnement de données spécifique. Le comportement d'un neurone dépend en effet des poids mais aussi du seuil d'activation qui lui est associé. Il est difficile voire même impossible de prédire les valeurs de ces deux paramètres pendant la phase de conception d'où le recours inévitable à l'apprentissage et cela tout en faisant subir au système un ensemble de stimulations itératives. Une règle d'apprentissage peut accepter les conditions existantes (poids et biais) du réseau et comparera le résultat attendu et le résultat réel du réseau sous formes d'erreur pour donner des valeurs nouvelles et améliorées pour les poids et les biais tout en réduisant cette même erreur afin que le réseau n'ait plus besoin d'apprentissage après avoir eu le comportement le mieux adapté pour la fonction cible. On distingue deux catégories de règles d'apprentissage : l'apprentissage supervisé et le non-supervisé

Règle d'apprentissage supervisé

L'apprentissage supervisé consiste à créer une relation (correspondance) entre une excitation ou stimulus en entrée et la réponse associée. Ainsi, la base d'apprentissage est présentée sous forme d'un ensemble de couples : stimulus et réponse cible associée (x_i, c_i) . D'autre part, les neurones situés sur la couche de sortie du réseau sont étiquetés de manière à ce que pour une entrée de valeur bien déterminée x_i , le neurone activé à la fin de la phase d'apprentissage soit celui qui porte la cible c_i correspondante à la réponse associée attendue pour l'entrée x_i . Ici la technique consiste à évaluer l'erreur pour chaque stimulus et modifier les paramètres libres des neurones afin de minimiser l'erreur dans les prochaines itérations. En cas de discordance, il faut affaiblir les poids de connexion des neurones actifs. Par contre, en cas de correspondance entre la réponse et le stimulus, la procédure consiste à renforcer les poids des neurones actifs. Différents domaines d'application sont associés aux algorithmes à base de règle d'apprentissage supervisé Parmi les algorithmes d'apprentissage supervisé les plus répandus, nous pouvons citer ceux vu en cours comme la règle Delta et les algorithmes de retro-propagation du gradient très utilisés par le modèle Perceptron Multi Couches (PMC) que nous verrons ci-dessous :

Perceptron multicouche



- $W_{i,j}^{(l)}$: poids entre le neurone j de la couche $l-1$ et le neurone i de la couche l
- $b_j^{(l)}$: biais du neurone j de la couche l
- $u_j^{(l)}$: sortie du neurone j de la couche l
- $z_j^{(l)}$: entrée du neurone j de la couche l , de telle sorte que $u_j^{(l)} = \sigma(z_j^{(l)})$

Un Perceptron Multi Couches (PMC) est un réseau neuronal artificiel composé de plusieurs couches, de telle sorte qu'il a la capacité de résoudre des problèmes qui ne sont pas linéairement séparables. Les couches peuvent être classées en trois types :

- Couche d'entrée : composée de ces neurones qui introduisent des modèles d'entrée dans le réseau. Aucun traitement ne se produit dans ces neurones.
- Couches cachées : formées par les neurones dont les entrées proviennent des couches précédentes et dont les sorties vont aux neurones des couches ultérieures.
- Couche de sortie : Neurones dont les valeurs de sortie correspondent aux sorties de l'ensemble du réseau.

Rétropropagation du Gradient

On la surnomme aussi la règle delta généralisée. Les poids dans le réseau de neurones sont au préalable initialisé avec des valeurs aléatoires. On considère ensuite un ensemble de données qui vont servir à l'apprentissage. Chaque échantillon possède ses valeurs cibles qui sont celles que le réseau de neurones doit à terme prédire lorsqu'on lui présente le même échantillon. L'algorithme suit les étapes suivantes.

- Soient un échantillon \vec{x} que l'on présente à l'entrée du réseau de neurones et \vec{t} la sortie recherchée pour cet échantillon.
- On propage le signal en avant dans les couches du réseau de neurones : $x_k^{(n-1)} \mapsto x_j^{(n)}$, avec n le numéro de la couche.
- La propagation vers l'avant se calcule à l'aide de la fonction d'activation g , de la fonction d'agrégation h (souvent un produit scalaire entre les poids et les entrées du neurone) et des poids synaptiques \vec{w}_{jk} entre le neurone $x_k^{(n-1)}$ et le neurone $x_j^{(n)}$. La notation est alors inversée : \vec{w}_{jk} indique bien un poids de k vers j .

$$x_j^{(n)} = g^{(n)}(h_j^{(n)}) = g^{(n)}(\sum_k w_{jk}^{(n)} x_k^{(n-1)})$$

- Lorsque la propagation vers l'avant est terminée, on obtient à la sortie le résultat \vec{y} .
- On calcule alors l'erreur entre la sortie donnée par le réseau \vec{y} et le vecteur \vec{t} désiré à la sortie pour cet échantillon. Pour chaque neurone i dans la couche de sortie, on calcule (g' étant la dérivée de g):

$$e_i^{sortie} = g'(h_i^{sortie})(y_i - t_i)$$

- On propage l'erreur vers l'arrière $e_i^{(n)} \mapsto e_j^{(n-1)}$ grâce à la formule suivante :

$$e_j^{(n-1)} = g'^{(n-1)}(h_j^{(n-1)}) \sum_i w_{ij}^{(n)} e_i^{(n)}$$

$$\text{note: } e_i^{(n)} = e_i^{sortie} = (y_i - t_i) \frac{\partial y_i}{\partial h_i^{(n)}}$$

Règle d'apprentissage non-supervisé

Contrairement à la règle d'apprentissage supervisé, dans l'apprentissage non supervisé il n'existe pas de catégories préalables selon lesquelles les entrées doivent être regroupées. L'affectation d'un stimulus en entrée sera automatiquement gérée par le réseau afin de générer à la fin de la phase d'apprentissage une sorte de carte d'affectation permettant de regrouper les entrées. Pour cette raison, cette catégorie d'algorithmes d'apprentissage est souvent appelée algorithme de clustering (regroupement et séparation). A chaque itération, le rôle du réseau est d'analyser les relations entre les stimuli de chaque entrée et extraire les associations. Au cours de la phase d'apprentissage, les poids de connexion du réseau de neurones seront modifiés afin de fournir une régularité de classement. L'objectif du réseau est donc d'identifier la similarité des entrées présentes sur la base d'apprentissage afin de bien les regrouper. Ainsi, la finalité de l'apprentissage est de minimiser la similarité interclasses et maximiser la similarité intra classes. Après avoir

terminé la phase d'apprentissage, le réseau sera en mesure de regrouper les entrées aléatoires selon leurs similitudes pendant la phase d'utilisation, pour certaines applications, une opération d'étiquetage paraît obligatoire au cours de la transition entre la phase d'apprentissage et celle de décision. Cette dernière commencera par la présentation d'un ensemble de stimuli appelé base de test qui n'appartient pas à notre échantillon d'apprentissage en entrée. Cette base est composée des couples (stimulus, étiquette). A la fin de l'opération d'étiquetage, chaque neurone sur la couche de sortie aura étiqueté une classe. C'est une sorte de supervision de la règle dite non supervisée. Toutefois elle n'est nécessaire que pour certaines applications et peut ensuite être suivi par une « validation » validé par une base de validation que l'on retrouve notamment dans la validation croisée

Les algorithmes à base de règle d'apprentissage non supervisé sont souvent utilisés pour des fonctions de clustering Parmi les algorithmes d'apprentissage non supervisé les plus connus, on trouve la règle de Hebb, la règle K-moyenne et la règle des mémoires associatives proposées par Kohonen en 1977

Démonstration logicielle utilisant un Perceptron (Multicouche)

Prédiction sur une série temporelle

Sur R nous j'ai choisis une démonstration logicielle utilisant un perceptron multicouche avec la librairie **nnet**. Mon but a été de prédire une série temporelle $Y_t = 2 + 3 * \sin(2t + 4) + 0.1t^2 + Z$ avec 200 réalisations et des valeurs prises pour t dans [0,20] avec Z Bruit Blanc Gaussien $N(0,1)$

Le but a été de prédire Y_t à partir de ces valeur décalée (Lag package dplyr) de 1 : y_1 et de 2 : y_2 qui seront des échantillons de tailles 199 et 198. Pour cela j'ai comparé le PMC avec 3 autres méthodes d'apprentissage :

Régression linéaire multiple, **SVM** pour la régression et **Random Forest**

Création du "dataframe avec les données de la série décalé"

J'ai donc choisi de créer un data frame comprenant les colonnes y, x1 et x2 et réduit celui-ci de façons à ce qu'ils contiennent 198 réalisations de ces 3 échantillons.

On met en place nos donnée lié à la série temporelle sous forme de Data Frame :

```
#library(dplyr)
t <- seq(0,20,length=200) # time stamps
y <- 2 + 3*sin(2*t+4) + 0.1*t^2 + rnorm(200) # Série temporelle que L'on veut prédire
y1=lag(y,1) # y décalé de 1, 1 valeur manquante
y2=lag(y,2) # y décalé de 2, 2 valeur manquante

dat <- data.frame( y, y1, y2) # triplet avec des valeurs décalées
names(dat) <- c('y','y1','y2')
dat <- dat[c(3:200),] # Pas de prise en compte des lignes contenant des valeurs manquantes
head(dat)
```

	y	y1	y2
3	-1.2218516	-1.2296983	-0.9961246
4	-1.4322227	-1.2218516	-1.2296983
5	0.6871183	-1.4322227	-1.2218516
6	-1.7655098	0.6871183	-1.4322227
7	-0.2001248	-1.7655098	0.6871183
8	-1.9016468	-0.2001248	-1.7655098

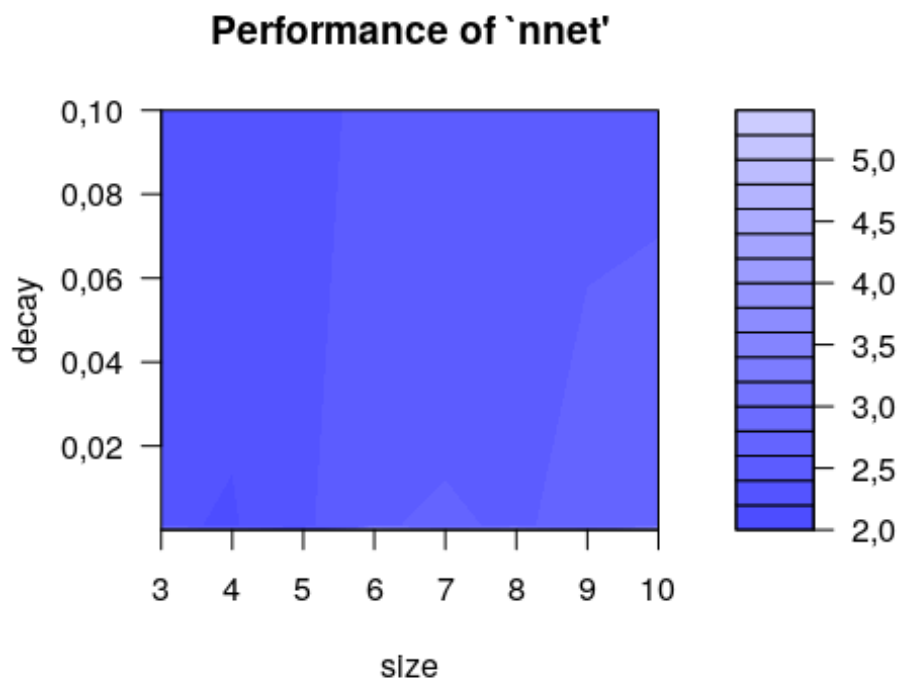
Comparaison du PMC avec d'autre méthode d'apprentissage

PMC

Pour le **Perceptron multicouche** nous utiliseront la librairie **nnet** de R. Tout d'abord le paramètre important à déterminer est **le nombre de neurones sur la couche cachée (size)** parallèlement aux conditions d'apprentissage (temps ou nombre de boucles), **j'ai choisi de faire une grille entre 3 et 10 pour size**. Une alternative ou un complément à la détermination du nombre de neurones est celle du **decay** qui est un paramètre de régularisation analogue à celui utilisé en régression ridge.

L'optimisation des paramètres nécessite encore le passage par la validation croisée. Il n'y a pas de fonction dans la librairie nnet permettant de le faire mais la fonction `tune.nnet` de la **librairie e1071** est adaptée à cette démarche nous l'utiliserons comme suit.

```
#library(nnet)
library(e1071)
#library(NeuralNetTools)
tune.model=tune.nnet(y~.,data=dat,size=3:10,decay=c(0.1,0.001,0.00001),maxit=200,linout=TRUE)
plot(tune.model)
```



```
tune.model
```

Parameter tuning of 'nnet':

- sampling method: 10-fold cross validation
- best parameters:
size decay
4 0,001

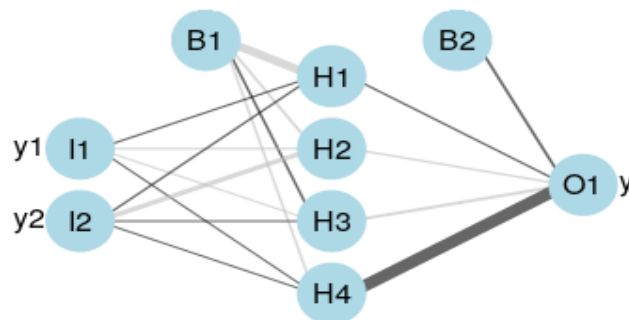
```
- best performance: 2,181611  
[1] "Pour le PMC nous choisissons donc une couche cachée avec 4 neurones et un decay  
de 0,001"
```

Nous définissons désormais notre modèle avec les paramètres retenus et avec le nombre de neurone dans la couche cachée

```
model <- nnet(y ~ y1+y2, dat, size=tune.model$best.parameters$size, decay=tune.model$best.parameters$decay, maxit=100, linout=TRUE)  
# weights: 17  
initial value 77162,824685  
iter 10 value 8623,392048  
iter 20 value 4429,926232  
iter 30 value 1606,891033  
iter 40 value 682,482175  
iter 50 value 420,047329  
iter 60 value 388,446942  
iter 70 value 383,156123  
iter 80 value 380,280751  
iter 90 value 378,394708  
iter 100 value 377,328215  
final value 377,328215  
stopped after 100 iterations
```

Voici un schéma de notre perceptron :

```
plotnet(model, alpha=0.6)
```



Régression Linéaire Multiple

Modèle de Régression Linéaire Multiple

```
myreg=lm(y~y1+y2, dat)  
leg <- predict(myreg, dat)
```

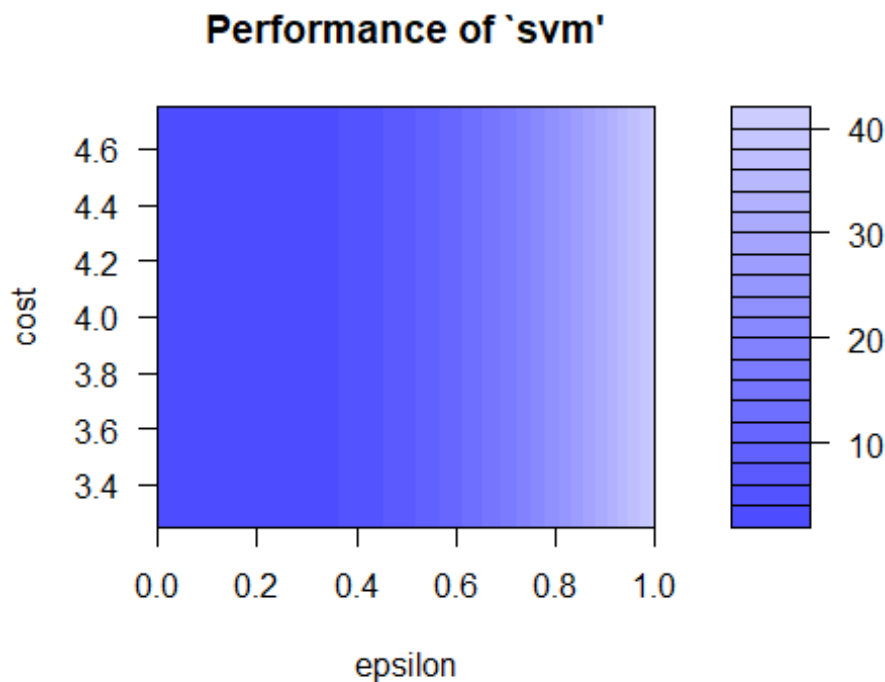

SVM

Afin d'améliorer les performances de la régression du SVM, nous devons sélectionner les meilleurs paramètres pour le modèle, il s'agit du paramètre epsilon de la régression et le cout (cost). J'ai donc utilisé comme pour nnet la commande tune de la librairie (e1071) pour avoir une grille de recherche de ses 2 paramètres.

SVM

Afin d'améliorer les performances de la régression du SVM, nous devons sélectionner les meilleurs paramètres pour le modèle. Il y a un paramètre. J'ai donc utilisé comme pour nnet la commande tune de la librairie (e1071) pour avoir une grille de recherche de ses 2 paramètres.

```
tune.result <- tune(svm,y~.,data = dat,  
ranges = list(epsilon = seq(0,1,0.1),  
cost = c(3.25,3.5,4,4.5,4.75)))  
plot(tune.result)
```



```
tune.result
```

```
Parameter tuning of 'svm':
```

```
- sampling method: 10-fold cross validation
```

```
- best parameters:
```

```
epsilon cost  
0.2 4.5
```

```
- best performance: 2.352079
```

```
[1] "Pour SVM nous choisissons epsilon= 0,2 et un coût de 4.5"
```

Modèle SVM

```
svmodel <- svm(y ~ y1+y2,data=dat, type="eps-regression",kernel="radial",cost=tune.result$best.parameters$cost,epsilon= tune.result$best.parameters$epsilon)
```

Random Forest

Modèle Random Forest

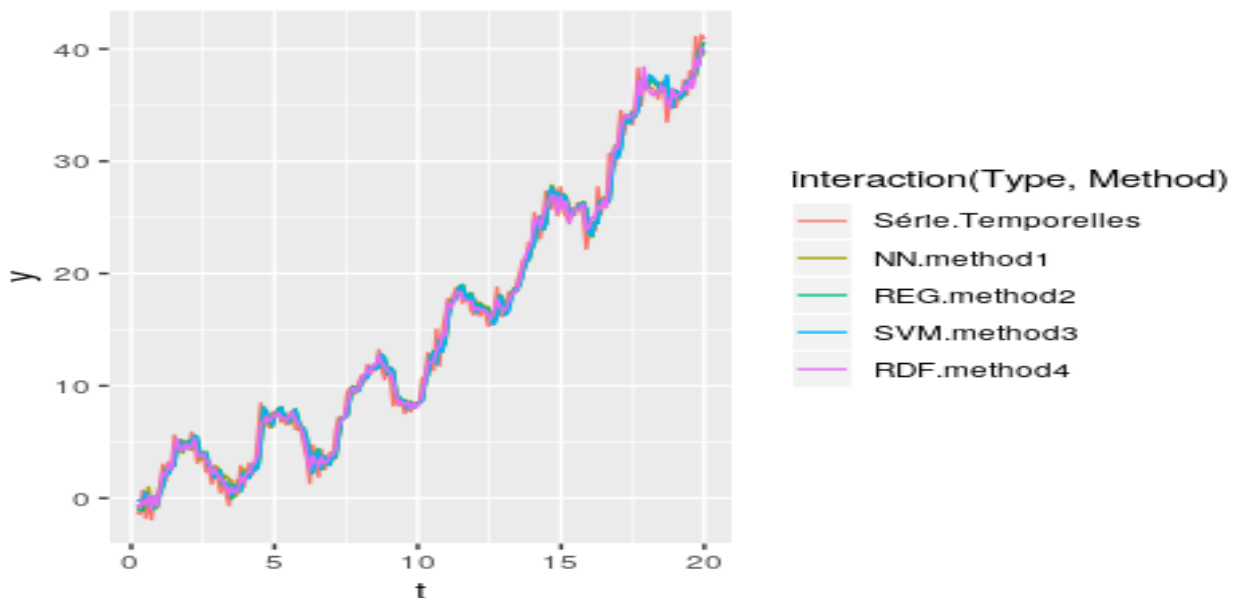
```
#library(randomForest)
rdf <- randomForest(y ~ y1+y2,data=dat)
```

Prévisions et Conclusion

Nous pouvons afficher l'ensemble des prévisions de la série temporelle Y_t pour nos 4 modèles :

```
ps <- predict(model, dat)
leg <- predict(myreg, dat)
svmp <- predict(svmodel, dat)
rdfp <- predict(rdf, dat)
#library(ggplot2)
t <- seq(0,20,length=200)
t<-t[-c(1:2)]
df1<-data.frame(t,y=y[-c(1:2)], Type = as.factor("Série"), Method = as.factor("Temporelles"))
df2<-data.frame(t,y=ps, Type = as.factor("NN"), Method = as.factor("method1"))
df3<-data.frame(t,y=leg, Type = as.factor("REG"), Method = as.factor("method2"))
df4<-data.frame(t,y=svmp, Type = as.factor("SVM"), Method = as.factor("method3"))
df5<-data.frame(t,y=rdfp, Type = as.factor("RDF"), Method = as.factor("method4"))
df.merged <- rbind(df1, df2, df3, df4,df5)

ggplot(df.merged, aes(t, y, colour = interaction(Type, Method))) + geom_line()
```



Erreur quadratique moyenne de prévision : $MSE = \frac{1}{n} \sum_{i=1}^n (\hat{Y}_i - Y_i)^2$

	MSE	Grille param
Random Forest	"0,622787686998201"	"Non"
Perceptron	"1,89085928494969"	"Oui"
SVM	"1,96653208706865"	"Oui"
Régression	"1,97251740575528"	"Non"

Nous voyons que **Random forest** est le plus performant

Démonstration logicielle utilisant les réseaux de neurones avec Scikit-Learn de Python

Scikit-Learn est une grande librairie Python dédié à l'apprentissage automatique : Dans cette partie nous allons utiliser la classe **MLPClassifier** qui implémente un algorithme de perceptron multicouche (MLP).

Création d'un jeu de donnée de classification

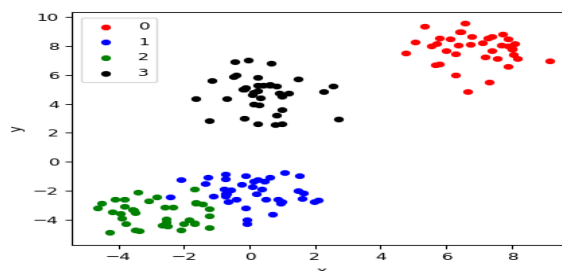
Dans cette partie nous allons traiter un simple problème de classification. Tout d'abord nous utiliserons la classe `make_blobs` de `scikit-learn.datasets` afin de générer un jeu de donnée de 150 variables et 2 observations, une classification comportant 4 cluster.

A l'aide de Pandas on peut créer ce dataset de la manière suivante :

```
from sklearn.datasets import make_blobs
import pandas as pd
X, y = make_blobs(n_samples=150, centers=4, n_features=3)
df = pd.DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
df.head()
##          x          y  label
## 0  0.302746  3.916803      3
## 1  0.097529  4.655703      3
## 2  7.899422  6.610820      0
## 3  6.581347  9.601528      0
## 4  7.865936  8.498602      0
```

On peut représenter nos 4 clusters défini par `make_blobs` comme suivant grâce à `matplotlib`

```
from matplotlib import pyplot
colors = {0:'red', 1:'blue', 2:'green', 3:'black'}
fig, ax = pyplot.subplots()
grouped = df.groupby('label')
for key, group in grouped:
    group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
pyplot.show()
```



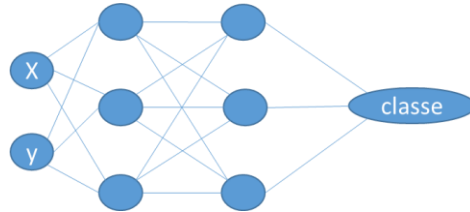
Séparation pour notre modèle

J'ai choisi de séparer les données avec `train_test_split` avec 70% pour nos données d'entraînement (train) et 30% pour nos données de test

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Mise en place du modèle PMC

Pour le MLP Classifier j'ai choisi de mettre 3 neurones sur 2 couches cachés



```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

model = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(3, 3), random_state=1)
```

Entraînement du modèle

On utilise la méthode `fit` pour entraîner notre modèle.

```
model.fit(X_train, y_train)
## MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto', beta_1=0.9,
##               beta_2=0.999, early_stopping=False, epsilon=1e-08,
##               hidden_layer_sizes=(3, 3), learning_rate='constant',
##               learning_rate_init=0.001, max_fun=15000, max_iter=200,
##               momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
##               power_t=0.5, random_state=1, shuffle=True, solver='lbfgs',
##               tol=0.0001, validation_fraction=0.1, verbose=False,
##               warm_start=False)
```

Prédiction et Conclusion

On fait nos prédictions à l'aide de la méthode `predict`.

```
y_pred = model.predict(X_test)
```

On peut afficher les classes prédites et les véritables classes de la manière suivante

```
print(y_pred)
## [2 1 3 0 3 1 1 0 3 3 2 1 2 2 3 0 1 2 3 1 0 0 0 3 0 2 0 2 0 0 2 0 2 3 2 0 2 2 0 1 1
##   3 3 0 1]
print(y_test)
## [2 1 0 0 3 1 1 0 3 3 2 1 2 2 3 0 1 2 3 1 0 0 0 3 0 2 0 2 0 0 2 0 2 3 2 0 2 2 0 1 1
##   3 3 0 1]
accuracy = accuracy_score(y_test, y_pred)
print('Précision: {0:.2f}'.format(accuracy * 100.0))
## Précision: 97.78
```

```

print('Rapport de classification :')
## Rapport de classification :
print(classification_report(y_test, y_pred))
##
##          precision    recall  f1-score   support
##
##         0           1.00      0.93      0.97         15
##         1           1.00      1.00      1.00          9
##         2           1.00      1.00      1.00         12
##         3           0.90      1.00      0.95          9
##
##    accuracy                0.98         45
##   macro avg           0.97      0.98      0.98         45
##  weighted avg           0.98      0.98      0.98         45
print('Matrice de confusion :')
## Matrice de confusion :
print(confusion_matrix(y_test, y_pred))
## [[14  0  0  1]
##   [ 0  9  0  0]
##   [ 0  0 12  0]
##   [ 0  0  0  9]]
print('')

```

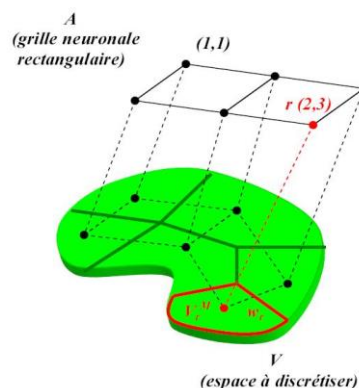
On a obtenu une précision de **97.78%** et on voit une confusion entre la classe 1 et la classe 4

Démonstration logicielle utilisant les cartes de Kohonen

Dans les parties précédentes nous avons vu des applications utilisant les réseaux de neurones dans le cadre de l'apprentissages supervisée cependant les réseaux de neurones peuvent aussi être utilisé en apprentissage non supervisée et aussi utilisé des méthodes de clustering, c'est ce que nous allons voir avec **les cartes de Kohonen**

Présentation

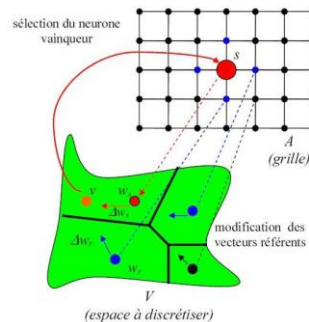
Principe Générale



D'un point de vue architectural la figure ci-dessus illustre bien le principe des cartes auto-organisatrices de Kohonen, les cartes auto-organisatrices de Kohonen sont constituées d'une grille (le plus souvent uni- ou bidimensionnelle). Dans chaque nœud de la grille se trouve un « neurone » : $\mathbf{r}(2,3)$. Chaque neurone est lié à un vecteur référent w_r , responsable d'une zone dans l'espace des données V (appelé encore espace d'entrée),

un point x de cette espace munie d'une certaine distance correspond donc à une de nos données que l'on aimerait bien représenter sur la grille.

Apprentissage



Chaque neurone a un vecteur référent qui le représente dans l'espace d'entrée. Tout d'abord un vecteur d'entrée v de l'espace des données V est présenté, v sélectionne ensuite le neurone vainqueur s tel que :

$$s = \operatorname{argmin}_{r \in A} \|v - w_r\|$$

s est donc le point le plus proche de v dans l'espace d'entrée V .

Les vecteurs référents des autres neurones voisins de s et s sont ensuite déplacés vers v , mais les vecteurs référents avec une amplitude moins importante que s .

En répétant tout ce qui a été dit avant pour chaque vecteur v de l'espace des données V , c'est toute la région de la carte autour du neurone gagnant qui se spécialise. En fin d'algorithme, lorsque les neurones ne bougent plus, ou seulement très peu, à chaque itération, la carte auto-organisatrice recouvre toute la topologie des données.

Sur la notion de voisinage

Trois types de voisinages couramment utilisés pour les cartes de Kohonen sont les voisinages linéaires, rectangulaire et triangulaire.

Les neurones sont reliés les uns aux autres, c'est la **topologie** de la carte. La forme de la carte définit les voisinages des neurones et donc les liaisons entre neurones elle est s .

La fonction de voisinage décrit comment les neurones dans la proximité du vainqueur " s " sont entraînés dans le mouvement de correction. On utilise en général :

$$h(r, s, t) = \exp\left(-\frac{\|\vec{r} - \vec{s}\|}{2\sigma^2(t)}\right),$$

où σ s'appelle "coefficient de voisinage". Son rôle est de déterminer un rayon de voisinage autour du neurone vainqueur.

La fonction de voisinage " h " force les neurones qui se trouvent dans le voisinage de " s " à rapprocher leurs vecteurs référents du vecteur d'entrée " v ". Moins un neurone est proche du vainqueur dans la grille, moins son déplacement est important.

Les cartes de Kohonen avec R (package « Kohonen ») sur les données Iris

Nous allons faire une démonstration logiciel des cartes de Kohonen avec la librairie Kohonen de R

```
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1         5.1         3.5         1.4         0.2   setosa
2         4.9         3.0         1.4         0.2   setosa
3         4.7         3.2         1.3         0.2   setosa
4         4.6         3.1         1.5         0.2   setosa
5         5.0         3.6         1.4         0.2   setosa
6         5.4         3.9         1.7         0.4   setosa
```

Tout d'abord j'ai découpé les données en 2 avec 100 données dans le corpus d'entraînement (**Train**) et 50 dans le corpus de test (**Test**) et tout cela dans une liste, on en verra l'utilité.

```
index<- sample(1:150, 100)
train_ <- list( x = as.matrix(iris[index,-5]), Species = as.factor(iris[index,5]))
#Taille 100
test_<- list(x = as.matrix(iris[-index,-5]), Species = as.factor(iris[-index,5]))
#Taille 50
```

Construction du modèle de la carte avec Supersom

```
carte <- supersom(train_, somgrid(xdim = 4, ydim = 5, topo = "hexagonal") , rlen = 500, alpha = c(0.05, 0.01))
```

Le but de l'application du modèle Carte est de prédire le type d'iris en utilisant 4 valeurs numériques de "Sepal.Length", "Sepal.Width", "Petal.Length" et "Petal.Width" comme variables indépendantes.

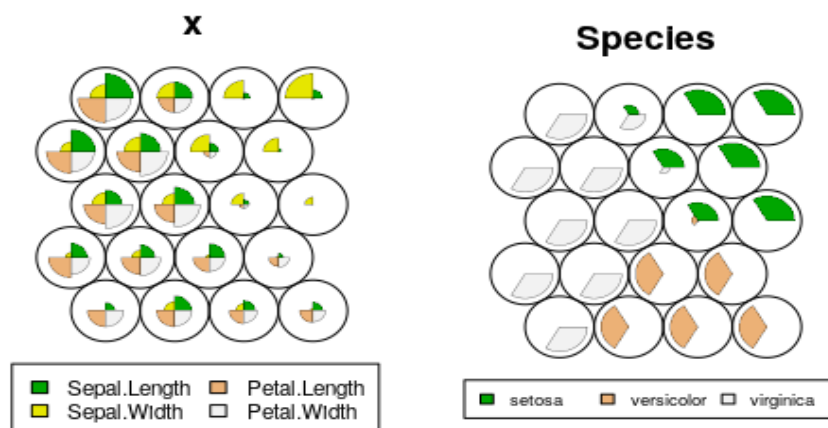
Les fonctions **somgrid()** et **Supersom()** du package kohonen sont des fonctions qui entraînent notre modèle carte. La fonction somgrid () est une fonction qui définit les types x, y et topologie de la couche de neurones (couche de sortie) rectangulaire ou hexagonal par exemple j'ai choisis** une carte de taille (4,5) et hexagonal, **et la fonction Supersom()**** est une fonction qui apprend le poids de chaque neurone via des données d'entrée ici j'ai choisis de séparer les données et les types d'iris (X et Species) afin de mieux gérer les prédictions à venir pour obtenir des prédictions pour toutes les couches utilisées dans la formation pour les mesures et les fleurs d'iris. **rlen** correspond au Nombre d'apprentissage ici 500 et **alpha** au Coefficient d'apprentissage. On peut utiliser la commande plot pour afficher la carte celle-ci comporte plusieurs **types** nous allons en voir 4 exemples suivantes :

Différents types de carte

Vecteur de poids

Les vecteurs de poids, ou **codes**, sont constitués de valeurs normalisées des variables originales utilisées pour générer le SOM. Le vecteur de poids de chaque nœud est représentatif/similaire des échantillons mis en correspondance avec ce nœud. En visualisant les vecteurs de poids sur la carte, nous pouvons voir des modèles dans la distribution des échantillons et des variables. La visualisation par défaut des vecteurs de poids est un "diagramme en éventail", dans lequel des représentations individuelles en éventail de l'ampleur de chaque variable du vecteur de poids sont affichées pour chaque nœud. Ici on peut voir qu'aucune des quatre variables n'est réduite où le vert est une variable de longueur de sépale, le rose est une variable de longueur de pétale, le jaune est une variable de largeur de sépale et le blanc est une variable de largeur de pétale.

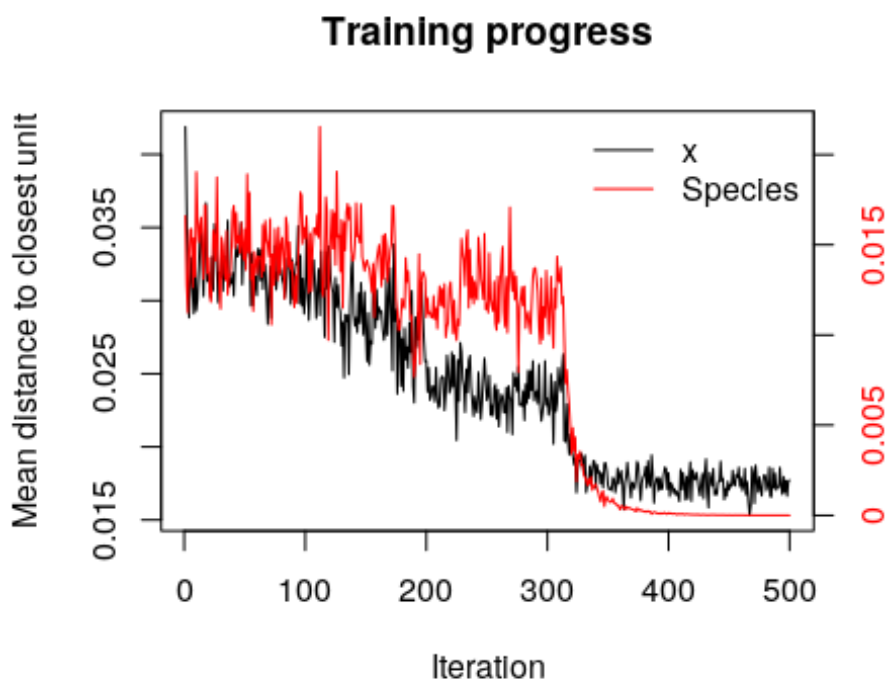
```
par(mfrow=c(1,2))
plot(carte, type="code")
```



Temps d'apprentissage

Au fur et à mesure que les itérations de formation SOM progressent (**Training Progress**), la distance entre les poids de chaque nœud et les échantillons représentés par ce nœud est réduite. Idéalement, cette distance devrait atteindre un plateau minimum. Cette option de tracé montre la progression dans le temps. Si la courbe diminue continuellement, il faut plus d'itérations. On utilise le type **changes**

```
plot(carte, type="changes")
```



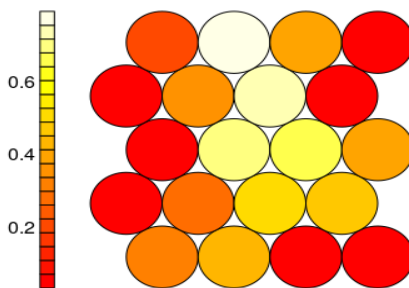
Le graphique des changements indique que **la courbe atteint à peu près un plateau au niveau de 300 itérations ce qui semble être un bon modèle.**

U-MATRIX

Si nous voulons voir les nœuds qui ont les voisins les plus proches ou les plus éloignés, nous pouvons tracer un graphique basé sur les voisins éloignés **dist.neighbours** somme des distances aux voisins immédiats pour chaque nœud. Les nœuds qui ont des couleurs plus sombres signifient que les nœuds ont une entrée vectorielle plus proche, tandis que les nœuds qui ont des couleurs plus claires signifient que les nœuds ont des entrées vectorielles plus éloignées. Les zones avec de grandes distances indiquent que les nœuds sont beaucoup plus dissemblables - et indiquent les frontières naturelles entre les groupes de nœuds. La matrice U peut être utilisée pour identifier les groupes de nœuds dans la carte SOM, **ici on voit à peu près 3 groupes se dégager avec des palettes grises (Species) voire blanches.**

```
plot(carte,type="dist.neighbours", main = "Distance entre chaque nœud et ses voisins")
```

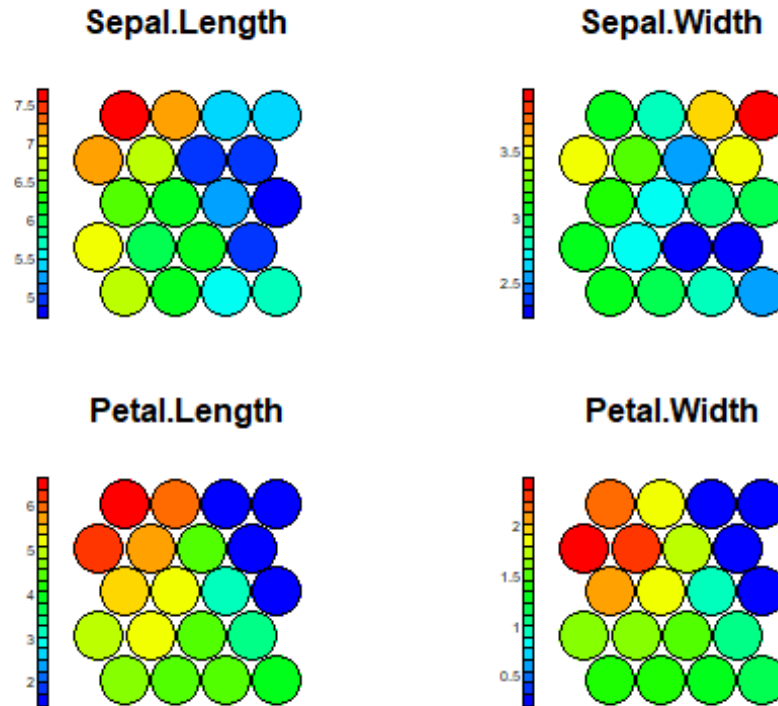
Distance entre chaque nœud et ses voisins



Les cartes thermiques

Les cartes thermiques **Heatmaps** sont peut-être la visualisation la plus importante possible pour les cartes auto-organisées. Une carte thermique SOM **permet de visualiser la répartition d'une seule variable sur la carte.** En général, un processus d'investigation SOM implique la création de plusieurs cartes thermiques, puis la comparaison de ces cartes thermiques pour identifier les zones intéressantes sur la carte. Il est important de se rappeler que les positions individuelles des échantillons ne passent pas d'une visualisation à l'autre, la carte est simplement colorée par différentes variables. La carte thermique de Kohonen par défaut est créée en utilisant le type **"property"**, puis en fournissant une des variables de l'ensemble des poids des nœuds. Dans ce cas, nous visualisons le niveau d'éducation moyen sur le SOM.

```
coolBlueHotRed <- function(n, alpha = 1) {  
  rainbow(n, end=4/6, alpha=alpha)[n:1]  
}  
  
par(mfrow=c(2,2))  
for (j in 1:4){  
  plot(carte,type="property",property=carte$codes$x[,j],palette.name=coolBlueHotRed,m  
ain=colnames(iris)[j],cex=0.5)  
}
```



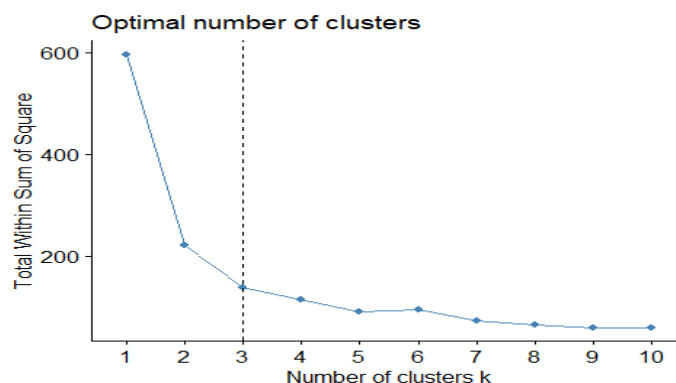
Clustering et segmentation au-dessus de la carte auto-organisée

Une méthode de clustering peut être effectuée sur les nœuds SOM pour isoler des groupes d'échantillons ayant des mesures similaires. L'identification manuelle des clusters est complétée par l'exploration des cartes thermiques pour un certain nombre de variables et la rédaction d'un historique sur les différentes zones de la carte. Une estimation du nombre de cluster qui conviendrait peut-être établie à l'aide d'un algorithme **kmeans** par la méthode du coude on le voit dans le graphique ci-dessous et on sait déjà que le nombre de cluster vaut 3.

Nombre optimal de Cluster

J'ai utilisé la fonction **fviz_nbclust** de la librairie **factoextra**, wss correspond à la méthode du coude

```
#library(factoextra)
fviz_nbclust(scale(iris[, -5]), kmeans, method = "wss")+
geom_vline(xintercept = 3, linetype = 2)
```



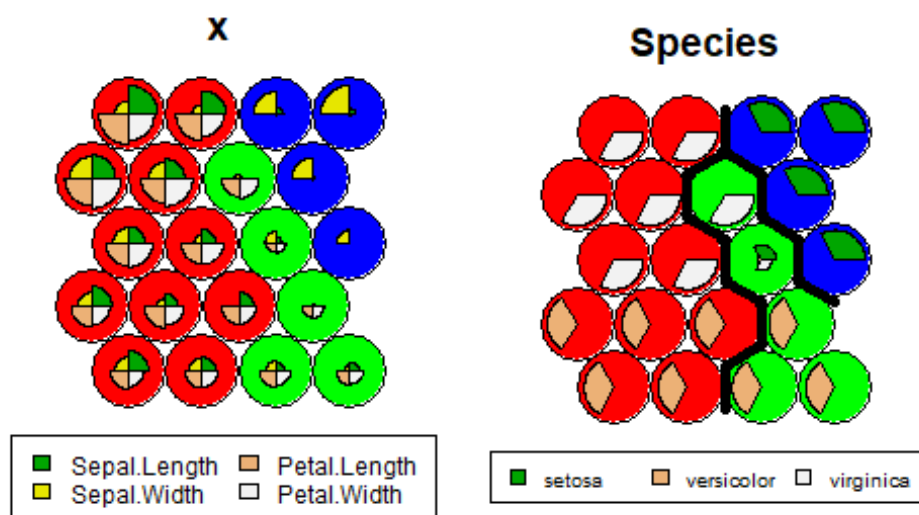
Le nombre de cluster optimal est donc de 3 sur le graphe ci dessus ont aurait aussi pu choisir 2 mais nous savons déjà qu'il y'en a 3 pour les données iris.

Clustering des nœuds.

La classification automatique est un intérêt des cartes topologiques de Kohonen et les nœuds sont un excellent point de départ pour un regroupement itératif en classes. La **classification hiérarchique ascendante (CAH)** une méthode de clustering qui est souvent utilisée dans ce contexte.

Nous calculons la distance entre les nœuds (entre les livres de codes) dans la carte (dist), puis nous effectuons un CAH (hclust) et nous regrouperons avec 3 couleurs

```
iris.sc = scale(iris[index, 1:4]) #on normalise
coolBlueHotRed <- function(n, alpha = 1) {rainbow(n, end=4/6, alpha=alpha)[n:1]} # n
os 3 couleurs
groups = 3
iris.hc = cutree(hclust(dist(carte$codes$x)), groups)
# plot
par(mfrow=c(1,2))
plot(carte, type="codes", bgcol=rainbow(groups)[iris.hc])
add.cluster.boundaries(carte, iris.hc) # lignes noires pour séparer nos 3 clusters
```



Prédictions

On peut obtenir des prédictions pour les 3 iris en se basant uniquement sur la cartographie des caractéristiques de l'échantillon. (Apprentissage non supervisé)

```
predicted_train <- predict(carte, newdata = test_, whatmap = "x")
```

```

mat<-table(predicted_train$predictions$Species, iris[-index,5])

taux = sum(diag(mat))/sum(mat)
mat
##
##      setosa versicolor virginica
## setosa      14         0         0
## versicolor  0         16         0
## virginica   0          1        14
taux
## [1] 0.9777778

```

On obtient une confusion. entre virginica et versicolor avec un taux de **97.7%**

On peut aussi effectuer voir les prédictions pour toutes les couches utilisées dans la formation, donc en tenant compte de Species (apprentissage supervisé)

```

predicted_train <-predict(carte, test_)
mat<-table(predicted_train$predictions$Species, iris[-index,5])

taux = sum(diag(mat))/sum(mat)
mat
##
##      setosa versicolor virginica
## setosa      14         0         0
## versicolor  0         22         0
## virginica   0          0        14
taux
## [1] 1

```

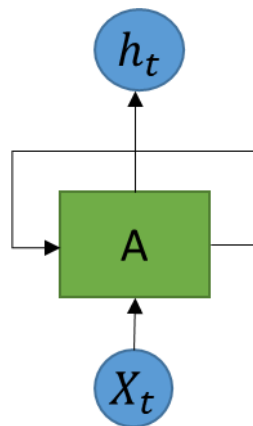
On peut voir qu'il n'y a pas eu d'erreur taux de **100%**

Démonstration logicielle d'un réseau de neurone profond RNN avec Keras et Tensorflow

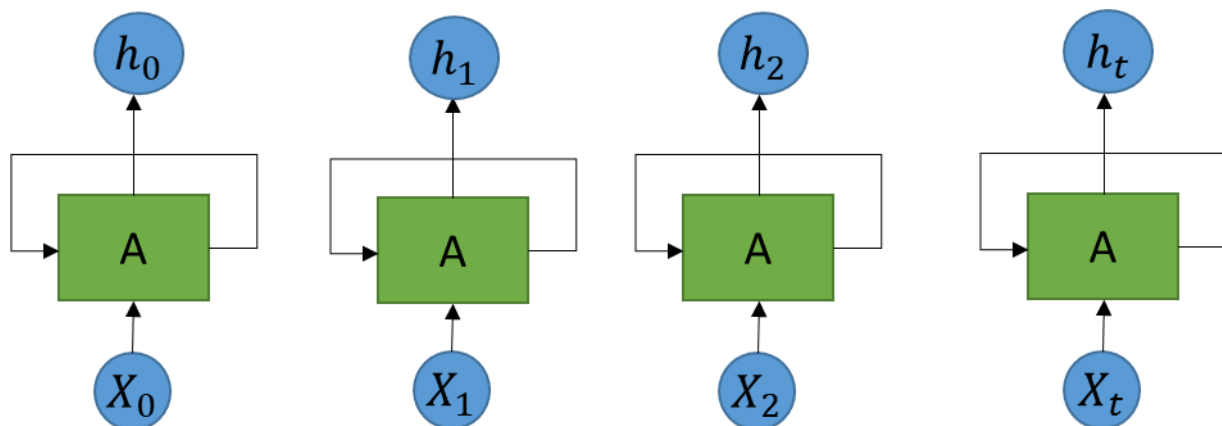
Présentation des RNNs

Les réseaux de neurones récurrents, ou RNN anglais Recurrent Neural Networks, est un réseau de neurones avec des boucles sur les neurones. Sa structure permet de conserver des informations, ce qui n'est pas possible avec les réseaux de neurones traditionnels comme le PMC, ce qui peut être l'un des principaux inconvénients des réseaux de neurones traditionnels. Les principaux domaines d'application des réseaux de neurones récurrents sont la reconnaissance vocale, la modélisation du langage, la traduction, les sous-titres d'images et l

Modèle neuronal du réseau neuronal récurrent :



L'auto-transmission (cette boucle) sur le neurone permet aux informations de conserver les informations entre les différentes étapes de la formation du réseau. En fait, ce modèle n'est pas compliqué, car il peut être étendu à une série de neurones ordinaires,



Comme le montre la figure ci – dessus chaque neurone représente un état, et d'un état à l'autre, les informations enregistrées par le neurone peuvent être transmises .

La considération de base de RNN est que vous pouvez utiliser les informations précédentes pour améliorer la compréhension des informations actuelles, telles que l'utilisation des premières images d'une vidéo complète pour identifier le contenu qui apparaît dans les quelques images suivantes, et par exemple, en utilisant

l'intrigue de la première demi-heure d'un film pour prédire l'arrière Terrain. Si RNN peut y parvenir, cela sera sans aucun doute très utile.

Le RNN ordinaire ne peut traiter que des relations étroites dans le temps, par exemple, utiliser les informations de la trame précédente pour prédire la trame suivante et les deux dernières trames, mais ne peut pas traiter les relations à long terme, telles que la prédiction de la trame actuelle avec une trame il y a une heure. Cadre. En effet, le gradient disparaîtra rapidement à mesure que le nombre de couches s'approfondit, similaire à celui rencontré dans les réseaux de neurones.

Il existe une structure spéciale de réseau RNN qui peut surmonter les problèmes ci-dessus et rendre possible le contact à long terme. Le nom de ce réseau est appelé LSTM (Long Short Term Memory networks).

Présentation et Préparation des données MNIST

Dans cette partie nous utiliserons l'ensemble de données MNIST. Cet ensemble de données contient des images représentant des chiffres manuscrits. Chaque image mesure 28 x 28 pixels et chaque pixel est représenté par un nombre (niveau de gris). Ces tableaux peuvent être aplatis en vecteurs de $28 \times 28 = 784$ nombres.

Tout d'abord j'ai commencé par importer les données Mnist cela est possible avec **keras.datasets** de la librairie Tensorflow de python.

```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

J'ai ensuite normalisé les données

```
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
## (60000, 28, 28)
## (60000,)
## (10000, 28, 28)
## (10000,)
```

On a 60000 images dans nos donnée train et 10000 dans nos données de test. Voici la répartition des chiffres dans ces données :

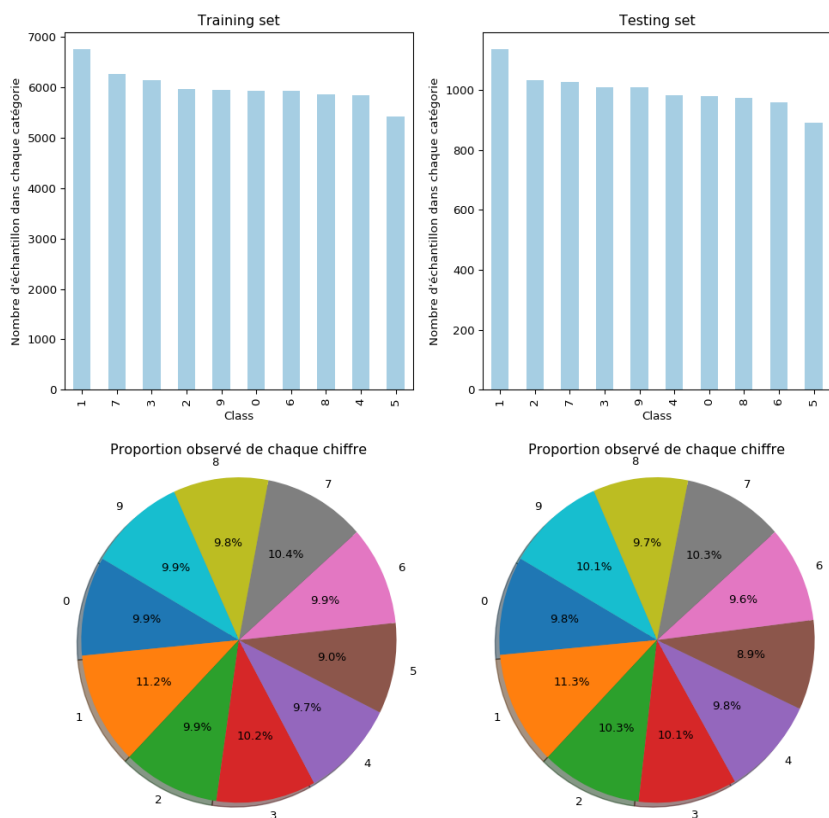
```
class_ = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
import matplotlib.pyplot as plt
import pandas as pd
plt.figure(figsize=(12,12))
y_train_df = pd.DataFrame(data = y_train, columns = ['class'])
y_test_df = pd.DataFrame(data = y_test, columns = ['class'])
plt.subplot(221)
y_train_df['class'].value_counts().plot(kind = 'bar', colormap = 'Paired')
plt.xlabel('Class')
plt.ylabel("Nombre d'échantillon dans chaque catégorie")
plt.title('Training set')
plt.subplot(222)
y_test_df['class'].value_counts().plot(kind = 'bar', colormap = 'Paired')
plt.xlabel('Class')
plt.ylabel("Nombre d'échantillon dans chaque catégorie")
```

```

plt.title('Testing set')
plt.subplot(223)
sizes = np.bincount(y_train)
explode = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
plt.pie(sizes, explode=explode, labels=class_,
autopct='%1.1f%%', shadow=True, startangle=150)
plt.axis('equal')
plt.title('Proportion observé de chaque chiffre')
plt.subplot(224)
sizes = np.bincount(y_test)
explode = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
plt.pie(sizes, explode=explode, labels=class_,
autopct='%1.1f%%', shadow=True, startangle=150)
plt.axis('equal')
plt.title('Proportion observé de chaque chiffre')

plt.show()

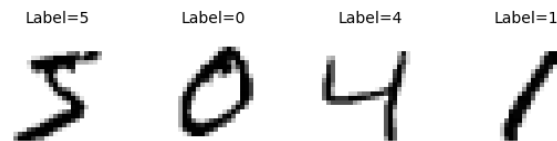
```



Affichage de 4 images et de leur classe

```
plt.figure(figsize=(8, 2))
```

```
for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.imshow(X_train[i].reshape(28, 28),
               interpolation="none", cmap="gray_r")
    plt.title('Label=%d' % y_train[i], fontsize=14)
    plt.axis("off")
plt.tight_layout()
```

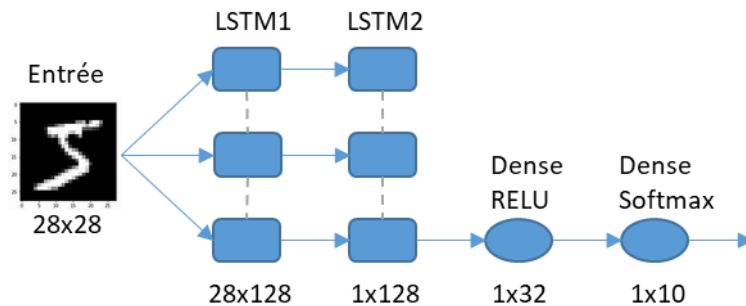


Mise en place du RNN

Pour la mise en place du modèle nous utiliserons un modèle séquentiel

```
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM
from tensorflow.keras.optimizers import Adam
```

L'architecture de notre modèle que nous avons choisie est principalement constituée de 4 couches différentes :



- Les 2 premières couches sont des couches **LSTM**, qui sont connectées en conséquence et alimentées avec des rangées d'images d'entrée. C'est-à-dire que chaque image d'entrée est codée en tant que vecteur de ligne de forme (28, 128) par la première couche LSTM, puis la deuxième couche effectuera un traitement supplémentaire pour sortir un vecteur d'image représentant l'image entière.,

- 2 couches **Dense** La mise en commun des couches (pour les caractéristiques de sous-échantillonnage)

- 1 abandon (**Dropout**, pour éviter le sur-ajustement ou overffiting des modèles)

Pour les fonctions d'activation de notre réseau, j'ai choisis la fonction **Relu** et **Softmax** en sortie pour les couches denses et pour la fonction de perte nous allons utiliser la fonction `categorical_crossentropy` et pour l'optimizer: `adam`

```
model = Sequential()
#Ajout d'une seconde Couche de réseau LSTM
model.add(LSTM(128,input_shape=(X_train.shape[1:]), return_sequences=True))
```



```

#Ajout d'une seconde Couche de réseau LSTM
model.add(LSTM(128))
#Ajout d'une couche caché Dense
model.add(Dense(64, activation='relu'))
#Abandon pour éviter Le sur-apprentissage
model.add(Dropout(0.2))
#couche de sortie softmax pour les multiclass
model.add(Dense(10, activation='softmax'))
#Compilation
model.compile( loss='sparse_categorical_crossentropy',optimizer=Adam(lr=0.001, decay=
1e-6),metrics=['accuracy'] )

```

Entrainement et Score

J'ai choisis de faire 10 epochs

```

#Entrainement
trained=model.fit(X_train,y_train, epochs=10, validation_data=(X_test, y_test))
## Epoch 1/10
1875/1875 [=====] - 14s 8ms/step - loss: 0.3532 - accuracy:
0.8892 - val_loss: 0.1070 - val_accuracy: 0.9689
## Epoch 2/10
1875/1875 [=====] - 14s 7ms/step - loss: 0.1038 - accuracy:
0.9702 - val_loss: 0.0786 - val_accuracy: 0.9766
## Epoch 3/10
1875/1875 [=====] - 14s 7ms/step - loss: 0.0723 - accuracy:
0.9793 - val_loss: 0.0719 - val_accuracy: 0.9786
## Epoch 4/10
1875/1875 [=====] - 14s 7ms/step - loss: 0.0575 - accuracy:
0.9832 - val_loss: 0.0605 - val_accuracy: 0.9819
## Epoch 5/10
1875/1875 [=====] - 14s 7ms/step - loss: 0.0454 - accuracy:
0.9870 - val_loss: 0.0532 - val_accuracy: 0.9840
## Epoch 6/10
1875/1875 [=====] - 13s 7ms/step - loss: 0.0381 - accuracy:
0.9892 - val_loss: 0.0494 - val_accuracy: 0.9860
## Epoch 7/10
1875/1875 [=====] - 13s 7ms/step - loss: 0.0332 - accuracy:
0.9902 - val_loss: 0.0471 - val_accuracy: 0.9873
Epoch 8/10
1875/1875 [=====] - 13s 7ms/step - loss: 0.0293 - accuracy:
0.9916 - val_loss: 0.0406 - val_accuracy: 0.9879
## Epoch 9/10
1875/1875 [=====] - 14s 7ms/step - loss: 0.0266 - accuracy:
0.9923 - val_loss: 0.0388 - val_accuracy: 0.9890
## Epoch 10/10
1875/1875 [=====] - 13s 7ms/step - loss: 0.0234 - accuracy:
0.9933 - val_loss: 0.0381 - val_accuracy: 0.9899

```

On regarde maintenant les scores

```

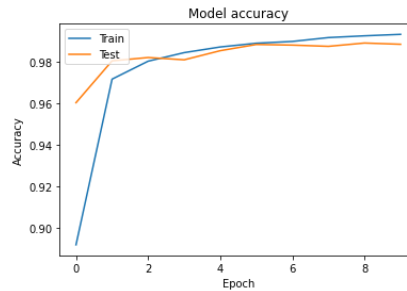
scores = model.evaluate(X_train, y_train)
scores2 = model.evaluate(X_test, y_test)
print("Training set Accuracy: %s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print("Testing set Accuracy: %s: %.2f%%" % (model.metrics_names[1], scores2[1]*100))

```

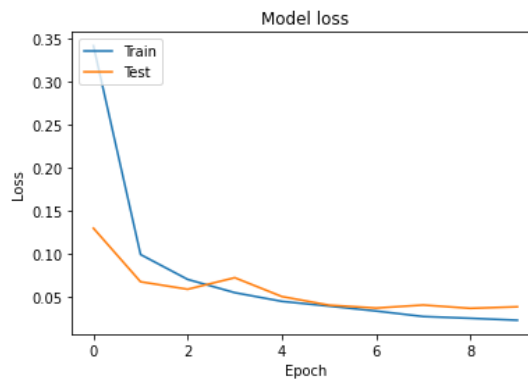
```
print("Training set Error: %s: %.2f%%" % (model.metrics_names[1], 100-scores[1]*100))
print("Testing set Error: %s: %.2f%%" % (model.metrics_names[1], 100-scores2[1]*100))
```

On obtient un score de **98.99%**! dans nos données de Test et un score de **99.57%** avec des pertes de 0.0381 et 0.0143, ces petits écarts semblent nous montrer qu'il n'y a pas d'overfitting nous le voyons dans les graphes suivants où les courbes bleu et orange ne s'éloignent pas trop:

```
plt.plot(trained.history['accuracy'])
plt.plot(trained.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



```
plt.plot(trained.history['loss'])
plt.plot(trained.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



Quelques Prédiction

En utilisant gridspec de matplotlib j'ai affiché le résultat de 10 prédictions comme suivant :

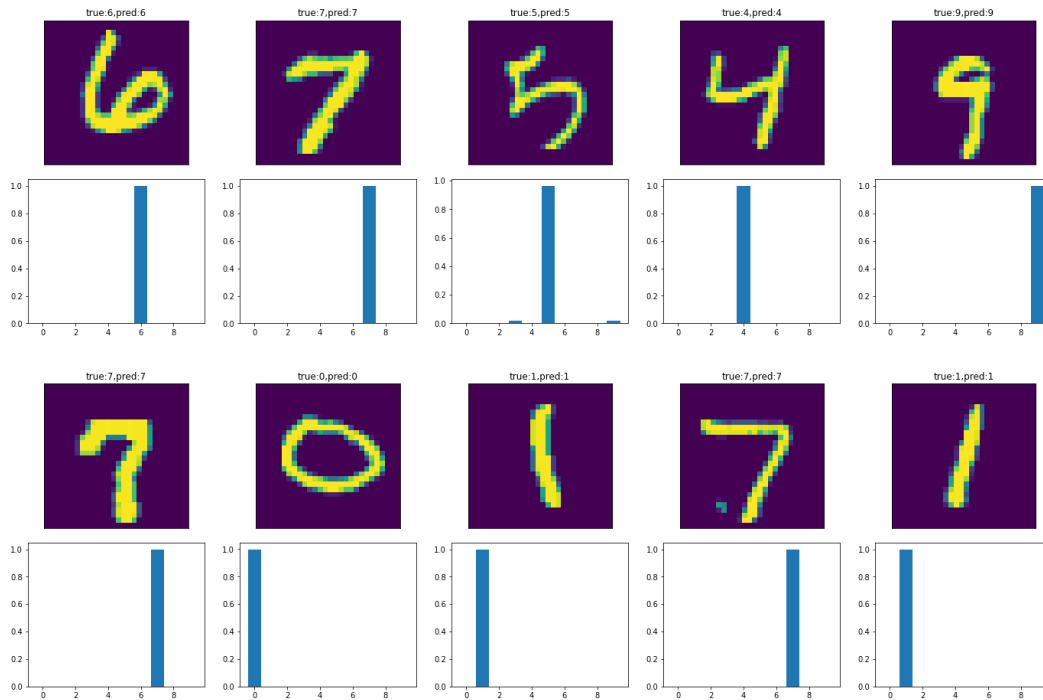
```
import matplotlib.gridspec as gridspec
from random import randint

def get_classlabel(class_code):
    labels = {0:"0", 1:"1", 2:"2",3:"3", 4:"4",5:"5",6:"6", 7:"7", 8:"8",9:"9"}

    return labels[class_code]

fig = plt.figure(figsize=(24, 16))
outer = gridspec.GridSpec(2, 5, wspace=0.2, hspace=0.2)

for i in range(10):
    inner = gridspec.GridSpecFromSubplotSpec(2, 1, subplot_spec=outer[i], wspace=0.1,
    hspace=0.1)
    rnd_number = randint(0,len(X_test))
    X_test1 = np.array([X_test[rnd_number]])
    pre_labels = model.predict(X_test1)
    pred_prob = pre_labels.reshape(10)
    for j in range(2):
        if (j%2) == 0:
            ax = plt.Subplot(fig, inner[j])
            ax.imshow(X_test1[0])
            ax.set_title('true:{},pred:{}'.format(get_classlabel(y_test[rnd_number]),
            get_classlabel(int(np.argmax(pre_labels, axis = 1)))))
            ax.set_xticks([])
            ax.set_yticks([])
            fig.add_subplot(ax)
        else:
            ax = plt.Subplot(fig, inner[j])
            ax.bar([0,1,2,3,4,5,6,7,8,9],pred_prob)
            fig.add_subplot(ax)
```



Notre modèle fonctionne donc très bien ! seul le 5 est très très peu confondu avec un 3 ou un 9. Nous Voyons ici l'efficacité des RNN LSTM dans la reconnaissance de chiffre.

Apprentissage par Renforcement

Introduction

L'apprentissage par renforcement, juxtaposé à l'apprentissage supervisé et à l'apprentissage non supervisé, sont les trois principales catégories d'apprentissage automatique. L'apprentissage par renforcement est peut-être un terme peu familier, mais c'est en fait l'une des technologies de base de l'intelligence artificielle, et ses applications sont très variées. Parmi les exemples célèbres, citons l'IA qui bat les humains au jeu de Go et l'IA qui fonctionne dans des voitures à conduite automatique. Les autres domaines d'application de l'apprentissage par renforcement comprennent la marche bipède robotisée et la finance. Quel type de technologie est donc l'apprentissage par renforcement ?

Aperçu de l'apprentissage du renforcement

L'apprentissage par renforcement, l'étude est que l'agent apprend du processus d'interaction avec l'environnement et apprend à agir sur l'environnement, afin que les meilleures incitations puissent être obtenues de l'environnement

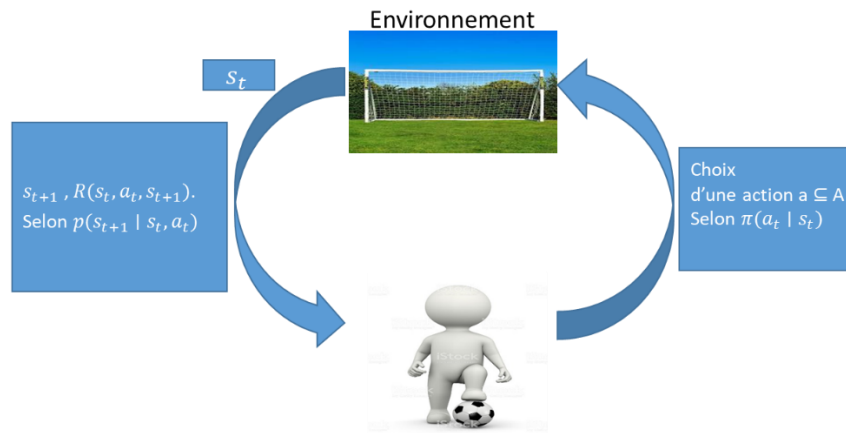
A titre d'exemple, imaginez que vous vous entraînez à tirer au football. Le flux de la pratique devrait ressembler à quelque chose comme ça.

- 1-Savoir où se trouvent le ballon et le but
- 2-Réfléchir à l'angle de vue à adopter
- 3-Prendre une décision de Tir.
- 4-Réfléchir à la question de savoir si le tir est celui que vous pensiez qu'il serait
- 5-Retour au point 1.

C'est exactement ce que nous essayons de réaliser par tâtonnements (dans ce cas, en nous améliorant dans le tir). L'idée de l'apprentissage par renforcement est la même que ci-dessus, mais pour la traiter mathématiquement, nous allons définir les termes suivants:

- 1-Connaitre la position de la balle et le but (dans l'environnement au moment t à l'État s_t)
- 2-Réfléchir à l'angle sous lequel il faut donner le coup de pied (Déterminer l'action à effectuer a_t selon la probabilité conditionnelle $\pi(a_t | s_t)$ appelée **la politique***)
- 3-Prise de décision du Tir (passage à l'état s_{t+1} au temps $t+1$, déterminé par la probabilité de transition d'état $p(s_{t+1} | s_t, a_t)$)
- 4-Regardez en arrière et réfléchissez si vous avez tiré comme prévu (Obtenez la récompense par la fonction de récompense $R(s_t, a_t, s_{t+1})$)
- 5-Retour au point 1.

Ce processus peut être décrit comme suit :



Les termes définis ici sont résumés comme le processus de décision Markov. (Bien que les mesures ne soient pas initialement incluses dans la définition, elles devraient être revues.

Définition : Processus de décision Markov (fini)

(fini) Espace d'**état**: $S = \{s_1, s_2, \dots\}$

(fini) Espace d'**action**: $A = \{a_1, a_2, \dots\}$

Politique $\pi(a_t | s_t)$

Probabilité de transition d'état $P: S \times A \times S \rightarrow [0,1] : p(s_{t+1} | s_t, a_t)$

Fonction de récompense $R: S \times A \times S \rightarrow \mathbb{R} : R(s_t, a_t, s_{t+1})$

L'agent apprend une **politique** π qui maximise la somme des futures récompenses dans l'itération ci-dessus.

En d'autres termes, la fonction objective pour l'Agent ou la récompense future est:

$$G_t = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$$

Dans ce cas, il sera le même que celui de l'année précédente. Un taux d'actualisation $\gamma \in [0,1]$ a été introduit ici.

Il devient un paramètre qui détermine si l'accent est mis sur les récompenses immédiates ou futures. En cas de tâtonnement dans le cadre du processus Markov, les données (s_t, a_t, s_{t+1}, r_t) peuvent être représentées par un ensemble.

C'est ce qu'on appelle les **données épisodiques**. Cependant, nous avons fixé $r_t = R(s_t, a_t, s_{t+1})$.

Pour les besoins de la discussion, définissons deux fonctions.

Définition : Fonction de valeur d'état et fonction de valeur d'action :

Fonction de valeur d'État $V^\pi(s) = \mathbb{E}_{\pi,P}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s]$

Fonction de valeur comportementale $Q^\pi(s, a) = \mathbb{E}_{\pi,P}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s, a_0 = a]$

Toutefois, si $\mathbb{E}_{\pi,P}[\]$ représente l'opération moyenne sur les données épisodiques lorsque l'agent essaie selon une politique π et une probabilité de transition d'état P . Comme son nom l'indique, la fonction de valeur d'état indique la valeur de chaque état S , et la fonction de valeur d'action indique la valeur de chaque action A dans chaque état S respectivement.

Les deux expressions sont très similaires, mais ont en fait une relation récursive comme suit:

$$\begin{aligned}
 Q^\pi(s, a) &= \mathbb{E}_{\pi,P} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s, a_0 = a \right] \\
 &= \mathbb{E}_{\pi,P} \left[R(s, a, s_1) + \sum_{t=1}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \right] \\
 &= R(s, a, s_1) + \mathbb{E}_{\pi,P} \left[\gamma \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \right] \\
 &= \mathbb{E}_{s_1 \sim P(s_1|s,a)} \left[R(s, a, s_1) + \mathbb{E}_{\pi,P} \left[\gamma \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s_1 \right] \right] \\
 &= \mathbb{E}_{s_1 \sim P(s_1|s,a)} [R(s, a, s_1) + \gamma V^\pi(s_1)] \\
 \\
 V^\pi(s) &= \mathbb{E}_{\pi,P} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid s_0 = s \right] \\
 &= \mathbb{E}_{a \sim \pi(a|s)} [Q^\pi(s, a)]
 \end{aligned}$$

Politique optimale

Maintenant, le but de l'apprentissage par renforcement est de découvrir **la politique optimale** π^* .

Maintenant, considérons la fonction de valeur d'état optimale et la fonction de valeur d'action. La fonction de valeur d'état optimale est $V^*(s) = \max_{\pi} V^\pi(s)$, et la fonction de valeur comportementale optimale est $Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$. De plus, la fonction de valeur optimale de l'état est $V^*(s) = \max_a Q^*(s, a)$ puisque l'action optimale a est adoptée dans l'état s et les transitions ultérieures sont basées sur les mesures optimales. De ce qui précède, on peut déduire l'équation de relation suivante :

$$\begin{aligned}
 Q^*(s, a) &= \mathbb{E}_{s_1 \sim P(s_1|s,a)} [R(s, a, s_1) + \gamma V^*(s_1)] \\
 &= \mathbb{E}_{s_1 \sim P(s_1|s,a)} [R(s, a, s_1) + \gamma \max_{a_1} Q^*(s_1, a_1)]
 \end{aligned}$$

Cette équation dans laquelle la fonction optimale action-valeur Q^* est satisfaite est appelée **équation de Bellman**, et la politique optimale $\pi^*(a \mid s) = \delta(a - \arg\max_a Q^*(s, a))$

Par exemple, la fonction de valeur d'action pour un processus de Markov fini peut être représentée par la matrice suivante :

$$Q = \begin{pmatrix} Q(s_1, a_1) & Q(s_1, a_2) & Q(s_1, a_3) \\ Q(s_2, a_1) & Q(s_2, a_2) & Q(s_2, a_3) \\ Q(s_3, a_1) & Q(s_3, a_2) & Q(s_3, a_3) \end{pmatrix}$$

C'est ce qu'on appelle une table de recherche, qui est intuitive et facile à comprendre.

Une fois que vous avez la fonction de valeur optimale, vous pouvez facilement déterminer la politique optimale, c'est-à-dire que pour n'importe quel état, vous pouvez choisir l'action qui maximise la fonction de valeur. Toute politique gourmande concernant la fonction de valeur optimale est considérée comme la politique optimale.

Il semble que tant que l'équation d'optimalité de Bellman est résolue explicitement, la politique optimale est trouvée. Mais cette méthode n'est qu'une théorie et ne fonctionne fondamentalement que si au moins les hypothèses suivantes sont remplies :

- Bonne capacité à modéliser avec précision l'environnement
- Des ressources suffisantes pour un calcul énorme
- Satisfaire le Processus de décision de Markov

De toute évidence, dans la pratique, ces trois hypothèses sont difficiles à satisfaire et il est généralement impossible de résoudre directement l'équation d'optimalité de Bellman ci-dessus pour obtenir la politique optimale. Toutes les méthodes approximatives sont utilisées pour résoudre l'équation d'optimalité de Bellman, comme les méthodes de Monte Carlo par exemple.

Méthode de Monte Carlo

Rappelons et réécrivons d'abord que $V(s) = E[G_t | S_t = s]$ avec $G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, la récompense future, également appelée retour, est une somme totale de récompenses actualisées à l'avenir. Les méthodes de Monte-Carlo (MC) utilisent une idée simple : elles apprennent des épisodes d'expérience brute sans modéliser la dynamique environnementale et calculent le rendement moyen observé comme une approximation du rendement attendu. Pour calculer le retour empirique G_t , les méthodes MC doivent apprendre des épisodes complets $S_1, A_1, R_2, \dots, S_T$ pour calculer $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$ et tous les épisodes doivent finalement se terminer.

Le rendement empirique moyen pour l'état s est:

$V(s) = \frac{\sum_{t=1}^T \mathbb{1}[S_t=s] G_t}{\sum_{t=1}^T \mathbb{1}[S_t=s]}$ où $\mathbb{1}[S_t = s]$ est une fonction d'indicateur binaire. Nous pouvons compter la visite des États à chaque fois afin qu'il puisse exister plusieurs visites d'un État dans un épisode (« chaque visite »), ou seulement la compter la première fois que nous rencontrons un État dans un épisode (« Première visite »).

Ce mode d'approximation peut être facilement étendu aux fonctions valeur-action en comptant (s, a) paire :

$$Q(s, a) = \frac{\sum_{t=1}^T \mathbb{1}[S_t=s, A_t=a] G_t}{\sum_{t=1}^T \mathbb{1}[S_t=s, A_t=a]}$$

Pour apprendre la politique optimale par MC, nous l'itérons de la manière suivante :

- 1-Améliorer la politique avec gourmandise par rapport à la fonction valeur actuelle : $\pi(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$
- 2-. Générez un nouvel épisode avec la nouvelle politique π (c'est-à-dire que l'utilisation d'algorithmes comme ϵ -greedy nous aide à équilibrer l'exploitation et l'exploration.)
- 3-Estimez Q en utilisant le nouvel épisode: $q_{\pi}(s, a) = \frac{\sum_{t=1}^T (\mathbb{1}[S_t=s, A_t=a] \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1})}{\sum_{t=1}^T \mathbb{1}[S_t=s, A_t=a]}$

Technique d'apprentissage par renforcement :

Tout d'abord introduisons **Algorithme ϵ -Greedy**

L'algorithme ϵ -Greedy prend la meilleure action la plupart du temps, mais effectue occasionnellement une exploration aléatoire.

La valeur de l'action est estimée en fonction de l'expérience passée en faisant la moyenne des récompenses associées à l'action cible a que nous avons observées jusqu'à présent (jusqu'au pas de temps actuel t):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^t r_\tau \mathbb{1}[a_\tau = a]$$

où $\mathbb{1}$ est une fonction d'indicateur binaire et $N_t(a)$ est le nombre de fois où l'action a a été sélectionnée jusqu'à présent, $N_t(a) = \sum_{\tau=1}^t \mathbb{1}[a_\tau = a]$.

Selon l'algorithme ϵ -Greedy, avec une faible probabilité ϵ nous prenons une action aléatoire, mais sinon (qui devrait être la plupart du temps, probabilité $1-\epsilon$) nous choisissons la meilleure action que nous avons apprise jusqu'à présent : $\hat{a}_t^* = \operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}_t(a)$.

SARSA

« SARSA » fait référence à la procédure de mise à jour de la valeur Q en suivant une séquence de $\dots, S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \dots$: État-action-récompense-État-Action. En SARSA, l'agent commence à l'état 1, effectue l'action 1, et obtient une récompense (récompense 1). Maintenant, il est dans l'état 2 et effectue une autre action (action 2) et obtient la récompense de cet état (récompense 2) avant qu'il ne remonte et mette à jour la valeur de l'action 1, effectuée dans l'état 1.

Voici l'algorithme de Sarsa :

- 1) Au pas de temps t , nous partons de l'état S_t et sélectionnons l'action en fonction des valeurs Q , $A_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(S_t, a)$; ϵ -greedy est couramment appliqué.
- 2) Avec l'action A_t , nous observons la récompense R_{t+1} et passons à l'état suivant S_{t+1} .
- 3) Choisissez ensuite l'action suivante de la même manière qu'à l'étape 1 : $A_{t+1} = \operatorname{argmax}_{a \in \mathcal{A}} Q(S_{t+1}, a)$.
- 4) Mettez à jour la fonction de valeur d'action : $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$.
- 5) $t = t + 1$ et répétez à partir de l'étape 1.

Dans chaque mise à jour de SARSA, nous devons choisir des actions pour deux étapes en suivant deux fois la politique actuelle (aux étapes 1. et 3.).

Q-Learning

Le développement du Q-learning (Watkins & Dayan, 1992) est une grande percée dans les premiers jours de l'apprentissage par renforcement.

En Q-learning de l'agent commence dans l'état 1, effectue l'action 1 et obtient une récompense (récompense 1). Il regarde ensuite, et voit que la récompense maximale possible pour une action est en état 2. Il l'utilise alors pour mettre à jour la valeur de l'action : effectuer l'action 1 dans l'état 1.

Voici l'algorithme Q-learning :

- 1) Au pas de temps t , nous partons de l'état S_t et sélectionnons l'action en fonction des valeurs Q , $A_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(S_t, a)$; ϵ -greedy est couramment appliqué.

- 2) Avec l'action A_t , nous observons la récompense R_{t+1} et passons à l'état suivant S_{t+1} .
- 3) Mettez à jour la fonction de valeur d'action : $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t))$.
- 4) $t = t + 1$ et répétez à partir de l'étape 1.

Les deux premières étapes sont les mêmes que dans SARSA.

À l'étape 3., Le Q-Learning ne suit pas la politique actuelle pour sélectionner la deuxième action mais estime plutôt Q^* parmi les meilleures valeurs Q indépendamment de la politique actuelle.

Conclusion + Références

Tout au long de ce projet nous avons pu voir l'apport des réseaux de neurones dans plusieurs problèmes d'apprentissage, que ce soit avec la régression dans la première partie ou avec la classification dans les autres, des méthodes comme les cartes de Kohonen s'avèrent être simple et particulièrement efficace par leur rapidité dans la classification, notamment avec les réseaux de neurone profond qui sont aujourd'hui particulièrement efficaces et très puissants, on a vu le RNN avec les LSTM mais on aurait aussi pu voir les CNN ou les DNN, nul ne devrait douter que l'intelligence artificielle appuyée sur les réseaux de neurone représente une solution d'avenir pour contribuer à faciliter la vie des hommes et aussi rendre meilleur notre perception par rapport à plusieurs problématiques.

<https://tel.archives-ouvertes.fr/tel-01868313/document>
https://fr.wikipedia.org/wiki/R%C3%A9tropropagation_du_gradient
<http://www.di.fc.ul.pt/~jpn/r/neuralnets/neuralnets.html>
https://scikit-learn.org/stable/modules/neural_networks_supervised.html
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
https://rtavenar.github.io/teaching/neuralnets_td/html/rnn.html
<https://cran.r-project.org/web/packages/kohonen/kohonen.pdf>
<https://meritis.fr/ia/cartes-topologiques-de-kohonen/>
<https://www.shanelynn.ie/self-organising-maps-for-customer-segmentation-using-r/>
<https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4>
<https://keras.io/>
http://renom.jp/id/notebooks/tutorial/reinforcement_learning/DQN-theory/notebook.html
<https://mitpress.mit.edu/books/reinforcement-learning>
http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf
<http://chercheurs.lille.inria.fr/~ghavamza/RL-EC-Lille/Lecture2.pdf>
https://www-igm.univ-mlv.fr/~dr/XPOSE2014/Machine_Learning/A_Q-Learning.html