RESEAUX DE NEURONES ARTIFICIELS

Rapport du projet

Diaby DRAME

Janvier 2023

Table des matières

Avant-propros

Ce rapport est rédigé dans le cadre du cours de Mme Annick VALIBOUZE sur les réseaux de neurones artificiels du master 2 ingénierie statistique et data science à l'ISUP. On retrouvera dans ce rapport une présentation de quelques réseaux de neurones artificiels qui seront accompagnés d'une illustration sous forme de travaux pratiques.

Afin de rendre la lecture de ce rapport fluide et ainsi de ne pas trop surchargé le lecteur ou la lectrice, j'ai fait le choix de mettre les bouts de codes qui sont utiles pour la comprehension. Tous les codes utilisés dans le cadre de ce projet sont disponibles dans leur entièreté dans l'annexe (y compris donc les pré-traitements des données).

1 Présentation des réseaux de neurones (Cas du Perceptron multi-couches)

Inspirés des neurones biologiques, les réseaux de neurones artificiels sont une des méthodes d'apprentissage statistiques.
Le premier réseau de neurones artificiel est le perceptron. Puis le perceptron multi-couches qui est une amélioration du perceptron.

1.1 Principe

Un neurone artificiel reçoit des données d'entrée et calcule une sortie qui est unique.

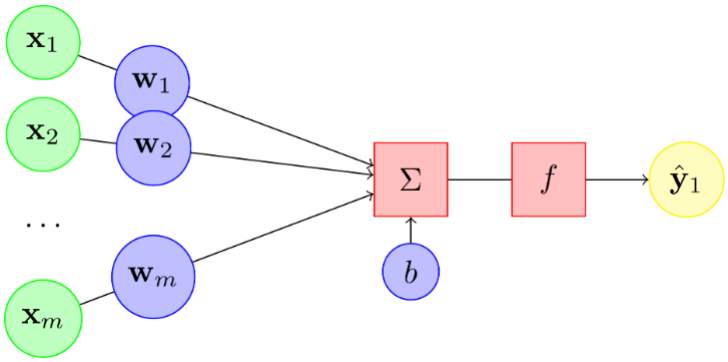

FIGURE 1 – Un neurone formel. *Source*

Le calcul de sa sortie se fait de la manière suivante : Calcul d'une équation linéaire $e = \sum_{i=1}^{m} wixi + b$ avec les $(x_1, ..., x_m)$ les données d'entrée du neurone, $(w_1, ..., w_m)$ et b les poids et biais associés respectivement. Puis cette valeur de e sera passée dans une fonction, appelée fonction d'activation (f sur la figure) et on obtiendra une sortie unique ($\widehat{y}_1$ sur la figure).

Le perceptron multi-couches (PMC) est une succession de réseaux formels connectés entre eux.

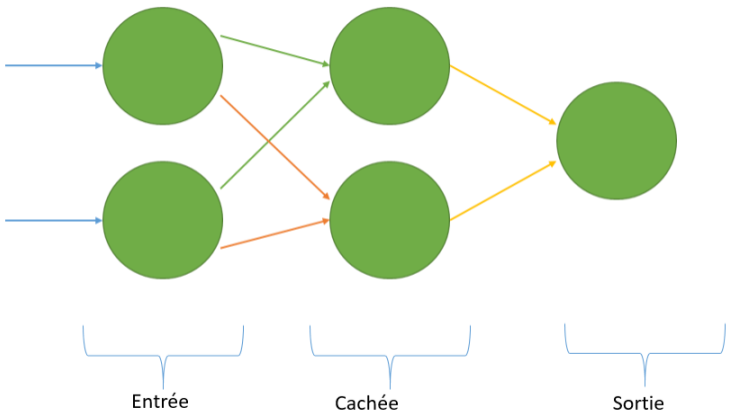

FIGURE 2 – Perceptron multi-couches. *Source*

Parmi les différentes fonctions d'activations, nous pouvons citer entre autres :
fonction identité $f(x) = x$
fonction sigmoïde $f(x) = \frac{1}{1+e^{-x}}$
fonction tangentielle $f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$
fonction RELU $f(x) = x^{+} = max(0, x)$
fonction softmax $f(a_i) = \frac{e^{a_i}}{\sum_{l=1}^{L} e^{a_l}}$, $(a_1, .., A_L)$ des activations. Cette fonction est principalement utilisé dans la dernière couche pour produire une sortie en probabilité.

1.2 Apprentisage

Les réseaux de neurones sont très souvent utilisés dans le cadre de l'apprentissage supervisé. C'est-à-dire qu'on dispose d'un label y sur nos données.
L'objectif est donc de minimiser la fonction coût qui mesure l'écart entre la sortie du réseau et les vraies données afin d'atteindre une meilleure approximation des données d'apprentissage.
La méthode utilisée pour l'optimisation est la descente du gradient. Puis on utilise l'algorithme de rétropropagation qui consiste à calculer le gradient de la fonction coût et à le propager en arrière de la couche de sortie vers la couche d'entrée.

2 Perceptron multi-couches avec comparaison à une méthode statistique

2.1 Jeu de données

Pour illustrer le perceptron multi-couches, on va utiliser un jeu données disponible en open source sur OpenML sur la classifcation du genre. On a 7 variables explicatives qui sont : cheveux (longs ou pas longs), largeur du front, hauteur du front, nez (large ou pas large), nez (long ou pas long), lèvres (minces ou pas minces) et distance nez à lèvres (longoue ou pas longue).

	long_hair	forehead_width_cm	forehead_height_cm	nose_wide	nose_long	lips_thin	distance_nose_to_lip_long	gender
0	1	11.8	6.1	1	0	1	1	Male
1	0	14.0	5.4	0	0	1	0	Female
2	0	11.8	6.3	1	1	1	1	Male
3	0	14.4	6.1	0	1	1	1	Male
4	1	13.5	5.9	0	0	0	0	Female

FIGURE 3 – cinq premières lignes du jeu de donnée.

2.2 Scikit-learn

Scikit-learn est une bibliothèque de python qui permet de faire du machine learning. Elle contient plusieurs modèles (codés sous forme de classe orienté objet) tels que la régression logistique, l'arbre de décision, la forêt aléatoire, etc.
Entraîner un modèle de machine learning avec scikit-learn se fait en 3 grandes étapes.
• La phase d'entraînement (fit) :
Cette méthode permet d'entraîner le modèle à partir des données des variables explicatives et des données de la variable cible.

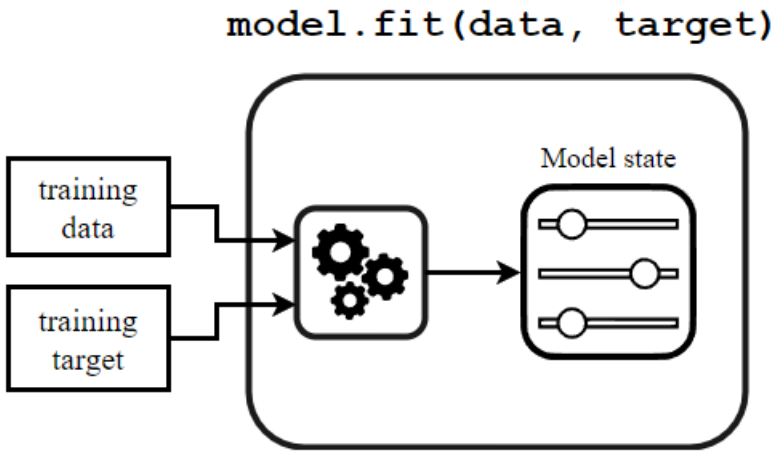

FIGURE 4 – Représentation de la méthode. Source [scikit-learn]

• La phase de prédiction (predict) :
Méthode permettant de faire des prédictions sur des données non utilisés dans la phase d'entraînement.

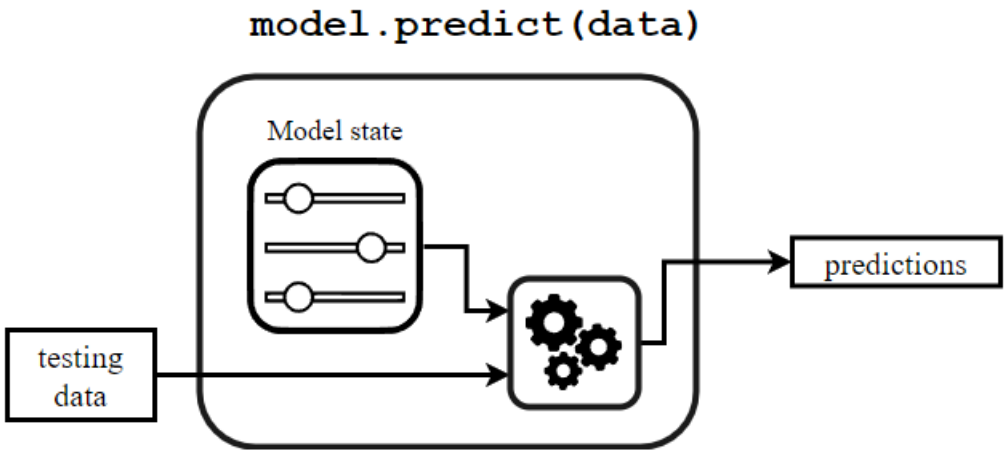

FIGURE 5 – Représentation de la méthode. Source [scikit-learn]

• La phase d'évaluation (score) :
Cette méthode nous renvoie le score obtenu sur les données de test.

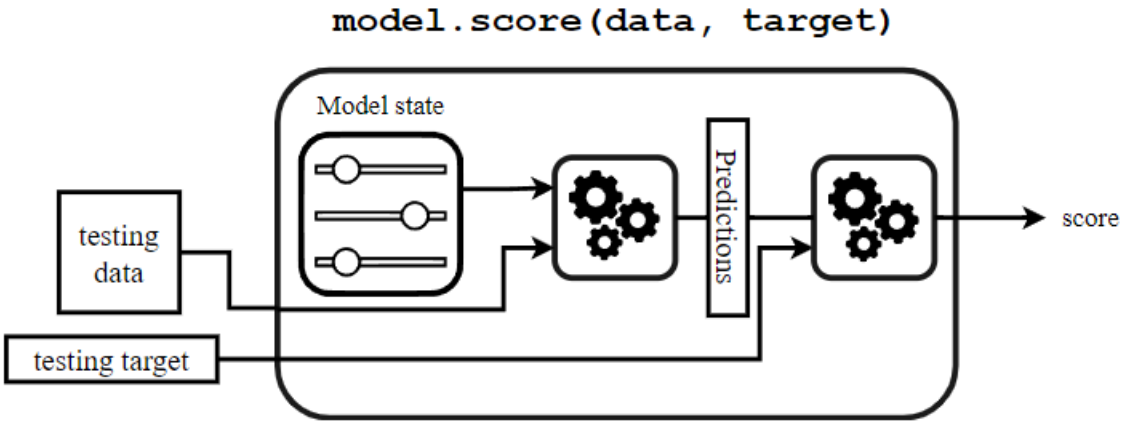

FIGURE 6 – Représentation de la méthode. Source [scikit-learn]

On utlisera le modèle MLPClassifier pour notre perceptron multi-couches.

2.3 Comparaison

Comme méthode statistique, on utilisera la forêt aléatoire. C'est un cas particulier des méthodes ensemblistes de type Bagging. Elle utilise comme estimateur de base l'arbre de décision.
Pour trouver les meilleurs paramètres de nos modèles, on utilisera GridSearchCV, une classe dans scikit-learn qui permet de trouver les meilleurs hyperparamètres d'un modèle. Elle prend deux arguments principaux, le modèle à tester et la liste des paramètres (sous forme de dictionnaire python) à tester.

Pour le perceptron multi-couches, on fait le test pour :
2 couches cachées de nombres de neurones 10,10 respectivement.
Plusieurs unique couche cachée de nombres de neurones 50,100,200,300.
Les fonctions d'activations qui seront testées : fonction identité, fonction sigmoïde, fonction tangentielle, fonction RELU.
Pour l'optimisation des poids, on utilisera les 3 méthodes disponibles dans le modèle MLPClassifier, à savoir : 'lbfgs' optimiseur de la famille des méthodes quasi-Newton, 'sgd'la descente de gradient stochastique et 'adam' une autre extension de la descente de gradient stochastique.

Remarque : le nombre de neurones dans la couche d'entrée et de sortie est automatiquement défini en fonction du jeu de données. Dans notre cas, la première couche aura 7 neurones (nombre de variables explicatives) et la dernière 2 neurones (pour les deux types mâle et femelle).

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
mlp_modele = MLPClassifier(random_state=0)

parametre = {'hidden_layer_sizes': [(10,10), (50,), (100,), (200,), (300,)],
             'activation': ['identity', 'logistic', 'tanh', 'relu'],
             'solver': ['lbfggs','sgd', 'adam']}

grid = GridSearchCV(mlp_modele, parametre)

grid.fit(data_train, target_train)

print(f"Les meilleurs paramètres sont : {grid.best_params_}")
```

```
Les meilleurs paramètres sont : {'activation': 'logistic', 'hidden_layer_sizes': (200,), 'solver': 'sgd'}
```

FIGURE 7 – Résultat du GridSearchCV.

Pour la forêt aléatoire, on fait le test pour :
Le nombre d'arbre dans la forêt :[10,20,30,40,50,60,70,80,90,100].
La profondeur maximale de l'arbre : [1,2,3,4,5,6,7,8,9,10].

```
from sklearn.ensemble import RandomForestClassifier

modele = RandomForestClassifier(random_state=0)
parametre = {'n_estimators' : [10,20,30,40,50,60,70,80,90,100],
             'max_depth' : [1,2,3,4,5,6,7,8,9,10]}
grid = GridSearchCV(modele,parametre)
grid.fit(data_train,target_train)

print(f"Les meilleurs paramètres sont : {grid.best_params_}")
```

```
Les meilleurs paramètres sont : {'max_depth': 4, 'n_estimators': 90}
```

FIGURE 8 – Résultat du GridSearchCV.

En entraînant nos deux modèles avec les meilleurs paramètres trouvés par GridSearchCV, nous obtenons comme score :

Modèles	Score sur les données d'entraînement	Score sur les données de test
PMC	0.9654285714285714	0.9700199866755497
Forêt aléatoire	0.9771428571428571	0.9706862091938707

On constate donc que la forêt aléatoire fait un petit peu mieux que le perceptron multi-couches. À présent, regardons la matrice de confusion de nos modèles.

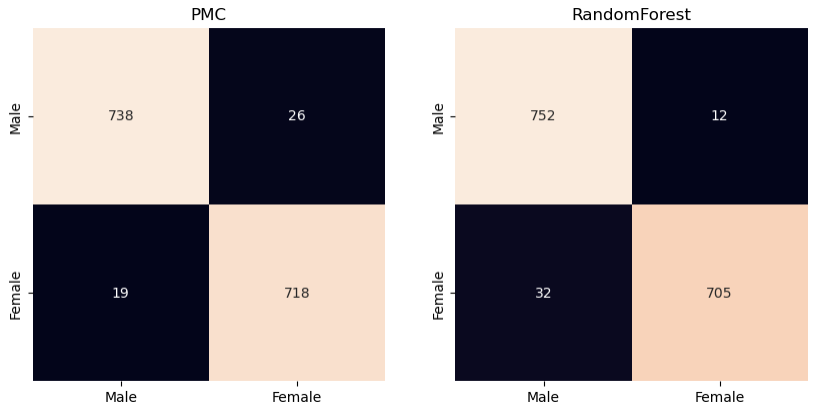

FIGURE 9 – Matrice de confusion.

le PMC fait donc 26 erreurs dans la classification des mâles, contrairement à la forêt aléatoire qui en fait 12. Dans la classification des femelles, il fait moins d'erreurs que la forêt aléatoire (19 contre 32).

3 Démonstration du logiciel Keras

Précédemment, on a vu comment entraîner le perceptron multi-couches avec scikit-learn. Mais scikit-learn est principalement utilisé pour l'application de ses différents algorithmes de machine learning. Il n'est pas totalement adapté pour les réseaux de neurones. Il est par exemple impossible de voir les détails du fonctionnement de son réseaux de neurones.
Dans cette section, on va utiliser Keras sur le même jeu de données que précédemment. Créée par François Chollet, Keras est une bibliothèque python très puissante permettant de faire du deep learning. Elle est très simple d'utilisation.

3.1 Construction du modèle

On crée d'abord le réseaux. On optera pour *Sequentiel*() qui est un modèle de plusieurs successions de couches.
La fonction *add*() permet d'ajouter des couches au réseaux.
La fonction *Dense*() pour spécifier des couches totalement connectées.
La fonction *Dropout*() permet d'éviter le sur-apprentissage en diminuant les données d'entrées.

3.2 Applicaton

On fait le test avec une couche en entrée de 200 neurones et une couche en sortie composée d'un seul neurone.
La fonction d'activation est la sigmoïde.

```
from keras import layers, models

modele = models.Sequential()
modele.add(layers.Dense (200,activation='sigmoid',input_shape=(data_train.shape[1],)))
modele.add(layers.Dropout (0.3))
modele.add(layers.Dense(1,activation ='sigmoid'))
```

FIGURE 10 – Confuguration du réseau.

On compile le modèle avec comme fonction coût 'binary crossentropy' qui correspond à la classification binaire, 'adam' comme optimiseur et une métrique sur le score. Puis, on lance l'apprentissage et faire l'évaluation.
Note : *batchsize* correspond à la taille du lot pour estimer le gradient de la fonction coût.

```
modele.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
modele.fit(data_train,target_train ,batch_size=40, epochs=50)


modele.evaluate(data_test , target_test)
```

```
Epoch 1/50
59/59 [==============================] - 1s 3ms/step - loss: 0.0889 - accuracy: 0.9586
Epoch 2/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0852 - accuracy: 0.9603
Epoch 3/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0860 - accuracy: 0.9571
Epoch 4/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0827 - accuracy: 0.9620
Epoch 5/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0844 - accuracy: 0.9600
Epoch 6/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0841 - accuracy: 0.9626
Epoch 7/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0866 - accuracy: 0.9611
Epoch 8/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0850 - accuracy: 0.9603
Epoch 9/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0860 - accuracy: 0.9611
Epoch 10/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0833 - accuracy: 0.9609
Epoch 11/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0879 - accuracy: 0.9560
Epoch 12/50
59/59 [==============================] - 0s 3ms/step - loss: 0.0872 - accuracy: 0.9591
```

FIGURE 11 – Phase d'apprentissage.

On obtient un score de **0.9694** qui n'est pas un mauvais score pour un reséau composé juste d'une couche en entrée et d'une couche en sortie. Regardons la matrice de confusion :

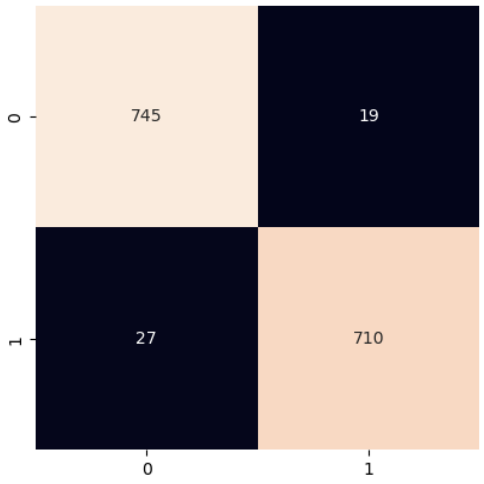

FIGURE 12 – Matrice de confusion.

4 Carte de Kohonen

Les cartes de Kohonen sont un type de réseau de neurones utilisés en apprentissage non supervisé. Ces réseaux permettent de projeter dans le plan ou dans l'espace des données multi-dimensionnelle tout en gardant les caractéristiques topologiques de ces données. Les cartes de Kohonen sont principalement utilisées pour la visualisation des données.

Fonctionnement : on dispose de neurones sur une grille avec des vecteurs poids associés. Chaque neurone de la grille est relié à l'ensemble des données d'entrée. Une notion de voisinage basée sur la distance est mise en place sur la grille.

Nous allons l'illustrer sur nos données de classification. On utilisera le package kohonen de R.

```r
```{r}
library(kohonen)

data = read.csv("gender.csv",stringsAsFactors = F)

data = scale(data[,-8]) #On enlève l'étique sur nos données tout en normalisant les valeurs

set.seed(0) # fixé la graine

grille = somgrid(4,4,topo="rectangular")

map = som(data,grid=g,rlen = 200) # nombre d'itération 200

plot(map,type = 'change')
plot(map,type='mapping')
plot(map, type = "codes", main = "Codes Plot", palette.name = rainbow)
plot(map, type = "dist.neighbours", main = "Distance voisinage")
```
```

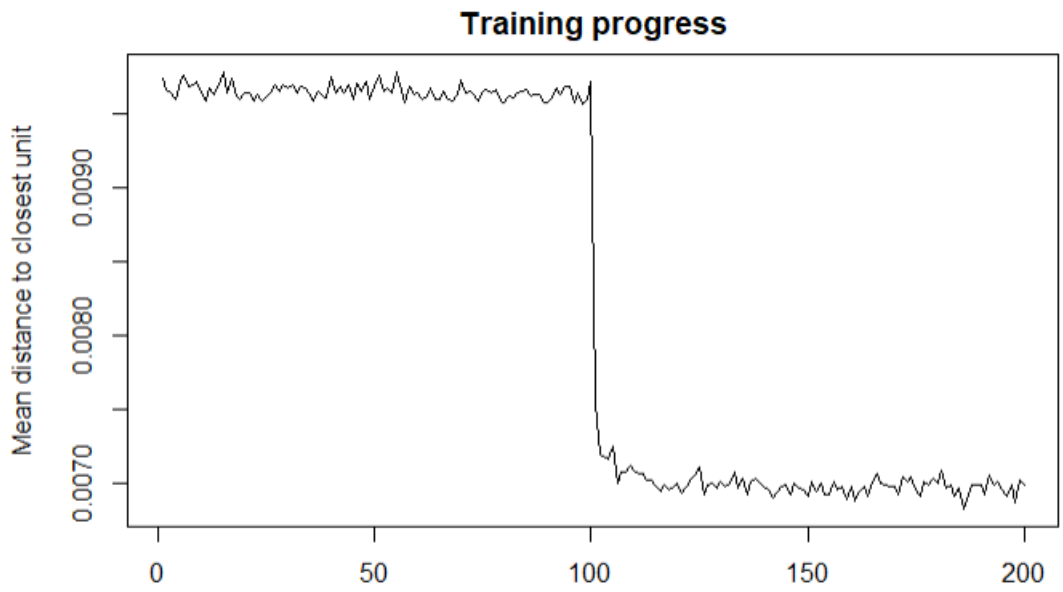

FIGURE 13 – Courbe de progression.

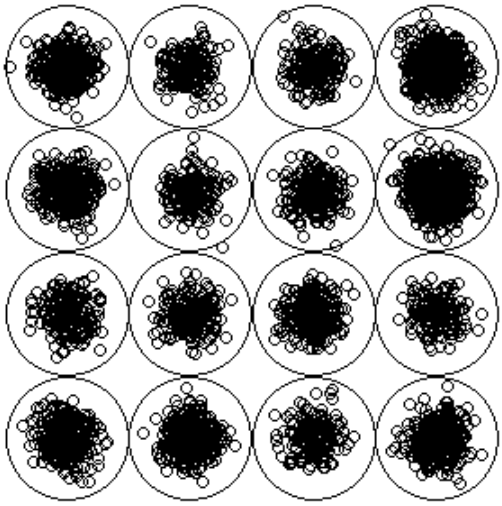

FIGURE 14 – Représentation des données sur la carte.

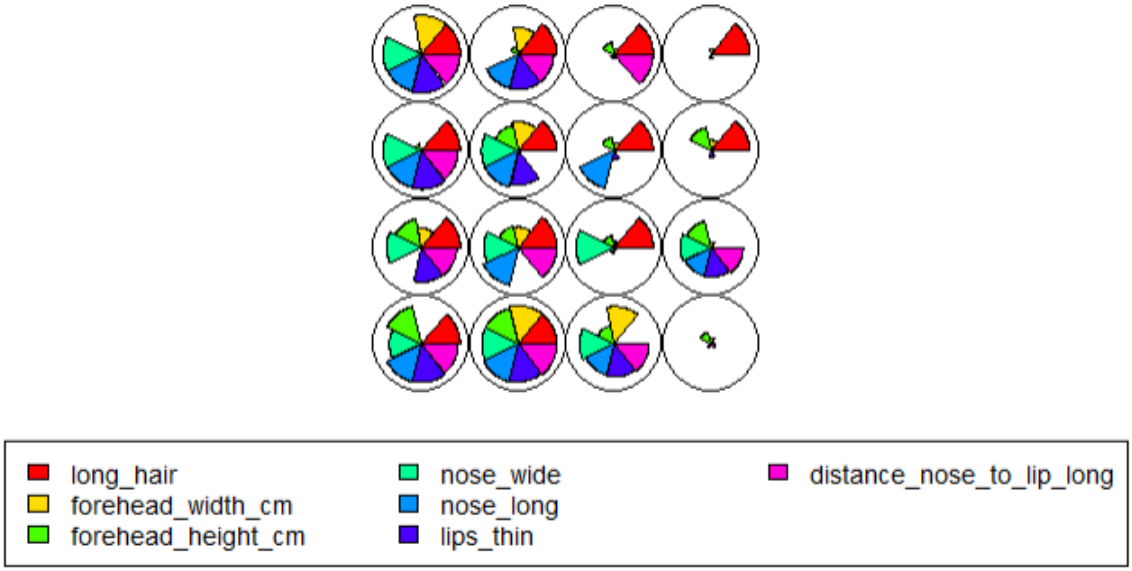

FIGURE 15 – Distribution à l'aide des poids des vecteurs de pondérations.

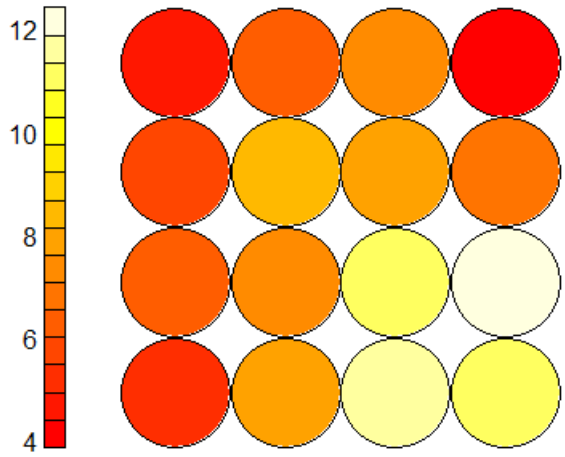

FIGURE 16 – Distance de voisinage.

5 Machine de Botzmann restreinte

La machine de Boltzmann restreinte est un réseau de neurones utilisé dans le cadre de l'apprentissage non supervisé. Il est composé de deux couches, une couche visible et une couche cachée. Tous les neurones de la couche visible sont connectés à ceux de la couche cachée et vise versa.

5.1 Fonctionnement

Le réseau fonctionne comme suit : Au début la couche visible reçoit des données d'entrée qui passeront dans les neurones de la couche cachée. Ceux-ci multiplient les données par un certain poids puis, un biais y sera ajouté. Le résultat passe donc dans une fonction d'activation afin de générer une sortie. Ensuite, la valeur de sortie générée au niveau des neurones de la couche cachée deviendra une nouvelle entrée en répétant la même procédure que précédemment mais cette fois-ci dans le sens inverse. Ensuite, l'entrée régénérée sera comparée à l'entrée d'origine si elle correspond ou non. Ce processus continuera jusqu'à ce que l'entrée régénérée soit alignée avec l'entrée d'origine.

6 Réseaux de neurones à convolution

Les réseaux de neurones à convolution sont les types de réseaux de neurones construits pour le traitement et la classification d'image.
Son architecture est composée principalement d'une couche de convolution, d'une couche de pooling et la fonction d'activation RELU.

• Couche de convolution :
Première couche d'un réseau neuronal convolutif, elle consiste à faire glisser une matrice (appelée feature) sur l'image et faire le produit de convolution.

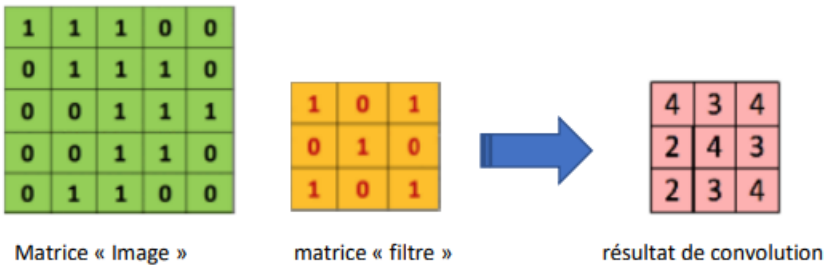

FIGURE 17 – Image d'illustration. *Source*

• Fonction RELU :
Après avoir fait passer une image à travers une couche convolutive, la sortie passe normalement dans la fonction RELU afin de transformer les valeurs négatives en zéro et conserver celles déjà positives.

• Couche de Pooling : C'est la couche qui permet de réduire la taille de l'image tout en conservant les informations importantes. La méthode qui est souvent utilisée est le max pooling C'est-à-dire créer des groupes de pixels sur l'image et pour chaque groupe prendre le maximum.

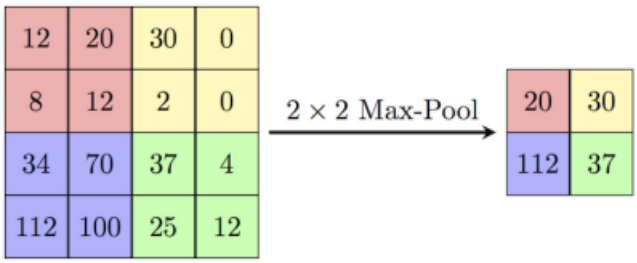

FIGURE 18 – une opération du max pooling de taille 2x2. *Source*

6.1 Illustration

Pour illustrer le réseau de neurones à convolution, nous allons utiliser le jeu de données mnist. Il comprend 60.000 exemples d'entraînement et 10.000 exemples de test, chacun étant une image de niveaux de gris

de 28x28 pixels représentant un chiffre manuscrit allant de 0 à 9. Pour chaque image, il y a 785 variables quantitatives dont 784 variables pour chaque pixel qui représentera le niveau de gris et une variable "label" qui représente l'étiquette du chiffre manuscrit.

FIGURE 19 – Quelquques images du jeu de donnée.

On va utiliser comme logiciel Keras. L'architecture du réseau ne diffère de celle faite dans le PMC quand quelques lignes.

```r
```{r}
library(keras)
modele <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu', input_shape =
c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(0.2) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')
```

FIGURE 20 – Conception du réseau.

On remarque donc une première couche de convolution à deux dimensions avec 32 features de taille 3x3. la méthode de pooling utilisée est le max pooling de taille 2x2. On rajoute après une autre couche de convolution et de pooling. Puis on rajoute une couche dense de 100 neurones (flaten() permet d'aplatir les données). La couche de sortie prend 10 neurones pour représenter la probabilité d'appartenir à chacune de nos 10 classes (0 à 9).
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)            (None, 26, 26, 32)        320
 max_pooling2d_3 (MaxPooling2D)  (None, 13, 13, 32)     0
 conv2d_2 (Conv2D)            (None, 11, 11, 32)        9248
 max_pooling2d_2 (MaxPooling2D)  (None, 5, 5, 32)       0
 dropout_10 (Dropout)         (None, 5, 5, 32)          0
 flatten_1 (Flatten)          (None, 800)               0
 dense_11 (Dense)             (None, 100)               80100
 dropout_9 (Dropout)          (None, 100)               0
 dense_10 (Dense)             (None, 10)                1010
=================================================================
Total params: 90,678
Trainable params: 90,678
Non-trainable params: 0
_________________________________________________________________
```

FIGURE 21 – Un resumé de l'architecture.

```
```{r}
modele %>% compile(loss = loss_categorical_crossentropy,
                   optimizer = optimizer_adadelta(),
   metrics = c('accuracy'))

modele %>% fit(x_train,y_train,batch_size = 70,epochs = 20)
```
```

```
Epoch 1/20
858/858 [==============================] - 30s 33ms/step - loss: 0.9357 - accuracy: 0.6997
Epoch 2/20
858/858 [==============================] - 24s 28ms/step - loss: 0.9172 - accuracy: 0.7101
Epoch 3/20
858/858 [==============================] - 22s 26ms/step - loss: 0.9074 - accuracy: 0.7095
Epoch 4/20
858/858 [==============================] - 27s 31ms/step - loss: 0.8879 - accuracy: 0.7175
Epoch 5/20
858/858 [==============================] - 24s 28ms/step - loss: 0.8709 - accuracy: 0.7240
Epoch 6/20
858/858 [==============================] - 21s 25ms/step - loss: 0.8621 - accuracy: 0.7234
Epoch 7/20
858/858 [==============================] - 25s 30ms/step - loss: 0.8393 - accuracy: 0.7349
Epoch 8/20
858/858 [==============================] - 24s 28ms/step - loss: 0.8264 - accuracy: 0.7368
Epoch 9/20
```

FIGURE 22 – Apprentissage.

```
```{r}
modele%>% evaluate(x_test, y_test)
```
```

```
     loss  accuracy
0.4409341 0.8925000
```

FIGURE 23 – Evaluation.

On a donc un score autour de 90.

7 Conclusion

À l'issue de ce projet, on a pu constater que le monde des réseaux de neurones artificiels est vaste, qu'il existe différents types de réseaux et chacun avec ses particularités.

On a pu voir que les réseaux de neurones pouvaient être utilisés pour l'apprentissage supervisé et non supervisé.

La plupart du temps on a obtenu des scores qui n'étaient pas mauvais avec nos modèles, cela montre la puissance de l'apprentissage profond.

Les temps de calculs étaient longs comparés aux temps que met un modèle classique de machine learning.

8 Annexe

8.1 Perceptron multi-couches et forêt aléatoire

```
import os
os.chdir("path")

import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

## Pré-traitement

data0 = pd.read_excel("gender.xlsx")
# 5 premières lignes
data0.head()

target = data0["gender"]
data = data0.drop(columns = 'gender')
```

```
#separation en train et test

# 80% train et 30% test
data_train, data_test, target_train, target_test = train_test_split(data, target, test_size = 0.3, random_state=0)
print(f"Données d'entraînement de taille {data_train.shape}")
print(f"Données de test de taille {data_test.shape}")

## perceptron
mlp_modele = MLPClassifier(random_state=0)

parametre = {'hidden_layer_sizes': [(10,10), (50,), (100,), (200,), (300,)],
             'activation': ['identity', 'logistic', 'tanh', 'relu'],
             'solver': ['lbfggs','sgd', 'adam']}

grid = GridSearchCV(mlp_modele, parametre)

grid.fit(data_train, target_train)

print(f"Les meilleurs paramètres sont : {grid.best_params_}")

# modele avec les meilleurs parametres
mlp_modele = MLPClassifier(hidden_layer_sizes=(200,),activation='logistic',solver='sgd',random_state=0)

mlp_modele.fit(data_train, target_train)

mlp_prediction = mlp_modele.predict(data_test)

print(f"Score de train : {mlp_modele.score(data_train,target_train)}")
print(f"Score de test : {mlp_modele.score(data_test,target_test)}")
```

```
## foret aléatoire
modele = RandomForestClassifier(random_state=0)
parametre = {'n_estimators' : [10,20,30,40,50,60,70,80,90,100],
             'max_depth' : [1,2,3,4,5,6,7,8,9,10]}
grid = GridSearchCV(modele,parametre)
grid.fit(data_train,target_train)

print(f"Les meilleurs paramètres sont : {grid.best_params_}")

# modele avec les meilleurs parametres
modele = RandomForestClassifier(n_estimators = 90, max_depth= 4, random_state=0)
modele.fit(data_train,target_train)
rf_prediction = modele.predict(data_test)
print(f"Score de train : {modele.score(data_train,target_train)}")
print(f"Score de test : {modele.score(data_test,target_test)}")
```

```
# modele avec les meilleurs parametres
modele = RandomForestClassifier(n_estimators = 90, max_depth= 4, random_state=0)
modele.fit(data_train,target_train)
rf_prediction = modele.predict(data_test)
print(f"Score de train : {modele.score(data_train,target_train)}")
print(f"Score de test : {modele.score(data_test,target_test)}")

## matrice de confusion

fig, ax = plt.subplots(1,2,figsize = (10,10))
sns.heatmap(confusion_matrix(target_test, mlp_prediction),square=True,fmt='', annot=True, cbar=False
            , xticklabels=['Male','Female']
            , yticklabels=['Male','Female'],ax=ax[0])

ax[0].set_title("PMC")


sns.heatmap(confusion_matrix(target_test, rf_prediction), square=True,fmt='', annot=True, cbar=False
            , xticklabels=['Male','Female']
            , yticklabels=['Male','Female'],ax=ax[1])

ax[1].set_title("RandomForest")

plt.show()
```

8.2 PMC avec Keras

```
# bibliothèque

import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from keras import layers, models

# On convertie la variable à expliquer de mâle, femelle en 1 et 0
label_encoder = preprocessing.LabelEncoder()
data0['gender'] = label_encoder.fit_transform(data0['gender'])

target = data0["gender"]
data = data0.drop(columns = 'gender')

Seaparation des données en train et test

# 80% train et 30% test
data_train, data_test, target_train, target_test = train_test_split(data, target, test_size = 0.3, random_state=0)

# architecture
modele = models.Sequential()
modele.add(layers.Dense (200,activation='sigmoid',input_shape=(data_train.shape[1],)))
modele.add(layers.Dropout (0.3))
modele.add(layers.Dense(1,activation ='sigmoid'))

# apprentissage
modele.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
modele.fit(data_train,target_train ,batch_size=40, epochs=50)

modele.evaluate(data_test , target_test)
```

```
# matrice de confusion
predictions = modele.predict(data_test)
r = [round(x[0]) for x in predictions] #arroundir les prédictions pour des valeur en 0 ou 1

sns.heatmap(confusion_matrix(target_test, r),square=True,fmt='', annot=True, cbar=False)
```

8.3 Réseaux à convolution

```r
```{r}
data <- dataset_mnist()

## Pré-triatement

x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

x_train <- array_reshape(x_train, c(nrow(x_train),28,28, 1))
x_test <- array_reshape(x_test, c(nrow(x_test), 28,28, 1))


x_train <- x_train / 255
x_test <- x_test / 255


y_train <- to_categorical(y_train,10)
y_test <- to_categorical(y_test,10)

library(keras)
```

```r
# architecture

modele <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu', input_shape =
c(28,28,1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(0.2) %>%
  layer_flatten() %>%
  layer_dense(units = 100, activation = 'relu') %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(modele)

# Evaluation

modele %>% compile(loss = loss_categorical_crossentropy,
                   optimizer = optimizer_adadelta(),
   metrics = c('accuracy'))

modele %>% fit(x_train,y_train,batch_size = 70,epochs = 20)

modele%>% evaluate(x_test, y_test)
```
```

Références

[1] cours réseaux de neurones artificiels, Annick VALIBOUZE.

[2] cours apprentissage statistique, Claire BOYER

[3] Comprendre les réseaux de neurones, La revue IA

[4] What Are Self Organizing Maps, Simplilearn

[5] Qu'est ce qu'un réseau de neurones convolutif (ou CNN), Openclassrooms

[6] The Self-Organizing Map, Gisbert Schneider

[7] Restricted Boltzmann Machines, Geoffrey Hinton

[8] Introduction to Restricted Boltzmann Machines, Vidéo Coursera

[9] Convolutional Neural Networks, Explained, Mayank Mishra

[10] Fundamentals of Deep Learning, Nikhil Buduma

[11] Documentation scikit-learn

[12] Documentation python

[13] Documentation Keras

[14] Documentation R