Avrile Floro (n° étudiant : 22000086)

IA & Apprentissage

Le 29 septembre 2024

IA & Apprentissage

Projet

Avrile Floro

Étudiante n°22000086

Table des matières

1 Projet - Implémentation d'un algorithme SOM en C **1**
1.1 Introduction . . . 1
1.2 Présentation du projet . . . 1
1.3 Présentation du jeu de données utilisé . . . 2
1.3.1 Caractéristiques du jeu de données . . . 2
1.4 Compilation du programme et exemple . . . 3
1.5 Détails des modules du programme . . . 5
1.5.1 elements.h : Structures de données et déclarations des fonctions . . . 5
1.5.2 main.c : Le coeur du programme . . . 7
1.5.3 dataset.c : La gestion du jeu de données . . . 9
1.5.4 neural_net.c : Initialisation du réseau de neurones . . . 12
1.5.5 apprentissage_SOM.c : Entraînement du SOM . . . 14
1.5.6 resultats.c : Vérification et affichage des résultats . . . 18
1.5.7 utils.c : Les fonctions utilitaires . . . 19
1.6 Conclusion . . . 21

1 Projet - Implémentation d'un algorithme SOM en C

1.1 Introduction

Une carte auto-organisée (SOM de l'anglais `Self-Organizing Map`) est une technique d'apprentissage automatique non supervisé. Elle permet de représenter des données multidimensionnelles grâce à une carte bidimensionnelle. Sur la carte, des clusters d'observations ayant des valeurs similaires aux exemples étudiés apparaissent. Au sein des clusters, les observations situées proches les unes des autres ont des valeurs semblables. Cela permet d'organiser facilement des données à plusieurs dimensions (source : https://en.wikipedia.org/wiki/Self-organizing_map).
Le SOM est un réseau de neurones qui est entraîné par un apprentissage compétitif (le gagnant l'emporte en totalité). Il est inspiré des modèles biologiques des systèmes neuronaux. Son avantage est qu'il permet de représenter un espace multidimensionnel en une carte bidimensionnelle.

1.2 Présentation du projet

Dans le cadre de ce projet, nous avons implémenté un algorithme SOM et nous l'avons testé sur le jeu de données `wine.data`.

Nous avons généré une grille de neurones, où les poids des neurones sont initialisés de manière aléatoire, mais avec des valeurs faibles (comprises entre 0.1 et 0.5). Ensuite, les vecteurs de poids sont normalisés.
Les données du jeu d'entraînement `wine.data` sont extraites et normalisées, attribut par attribut. Chaque exemple du jeu de données est ensuite présenté à la grille de neurones. Le neurone dont les poids sont les plus proches du vecteur d'entrée est sélectionné comme meilleure correspondance (Best Matching Unit ou BMU).
Tous les neurones situés dans le voisinage du BMU voient leurs poids ajustés pour se rapprocher du vecteur d'entrée. Cette mise à jour est réalisée en fonction d'un facteur d'apprentissage qui diminue au fil des itérations.
Le jeu de données est présenté à la carte de neurones à travers 1500 itérations.
À la fin de l'entraînement, les labels des neurones sont représentés dans une matrice. Cela permet d'illustrer la classification effectuée par l'algorithme. Finalement, le jeu de données est à nouveau présenté à la carte et la justesse de la classification est évaluée en comparant les labels antérieurs des BMU avec ceux des vecteurs d'entrée.

Notre programme a été conçu de façon modulaire afin d'améliorer sa lisibilité et de faciliter sa maintenance.

Voici les différents modules utilisés :

- `elements.h` : Ce module contient les bibliothèques, définit les structures de données utilisées par le programme et déclare les fonctions.
- `main.c` : Ce module est le coeur du programme. Il est en charge de l'exécution des différentes parties du projet. C'est la fonction `main` qui lance la lecture des données et fait procéder à l'apprentissage et à l'affichage des résultats.
- `dataset.c` : Ce module est en charge de lire le fichier de données et d'en normaliser les attributs. Cela permet la préparation du jeu de données avant son traitement.
- `neural_net.c` : Ce module initialise et configure la structure du réseau de neurones. Des vecteurs aléatoires sont créés et leurs poids sont normalisés.
- `apprentissage_SOM.c` : Ce module contient les fonctions qui entraînent la carte SOM, y compris la sélection du neurone gagnant (le BMU) et la mise à jour des poids des différents neurones.
- `resultats.c` : Ce module permet l'affichage des résultats relatifs à l'évaluation de la performance de l'algorithme SOM après l'apprentissage.
- `utils.c` : Ce module contient des fonctions utilitaires permettant l'initialisation des tableaux annexes, la libération de la mémoire et la gestion des erreurs.

1.3 Présentation du jeu de données utilisé

Le jeu de données que nous avons utilisé dans ce projet s'appelle *Wine Dataset* (source : https://archive.ics.uci.edu/dataset/109/wine). Il s'agit des résultats d'une analyse chimique de vins produits dans la même région en Italie mais provenant de trois cépages différents. Nous avons utilisé ce jeu de données afin d'évaluer la performance de notre algorithme. Ce dataset est considéré comme un classique de la classification car il possède trois classes distinctes et une structure relativement simple.

Nous avons sélectionné ce jeu de données en raison de la présence de classes bien définies et de l'absence de valeurs manquantes. Les vins sont décrits à travers 13 caractéristiques, ce qui permet de les distinguer aisément tout en limitant la complexité dimensionnelle. De plus, la littérature indique que l'utilisation des réseaux de neurones pour la classification de ce jeu de données permet d'atteindre un taux de précision d'environ 97,78 %.

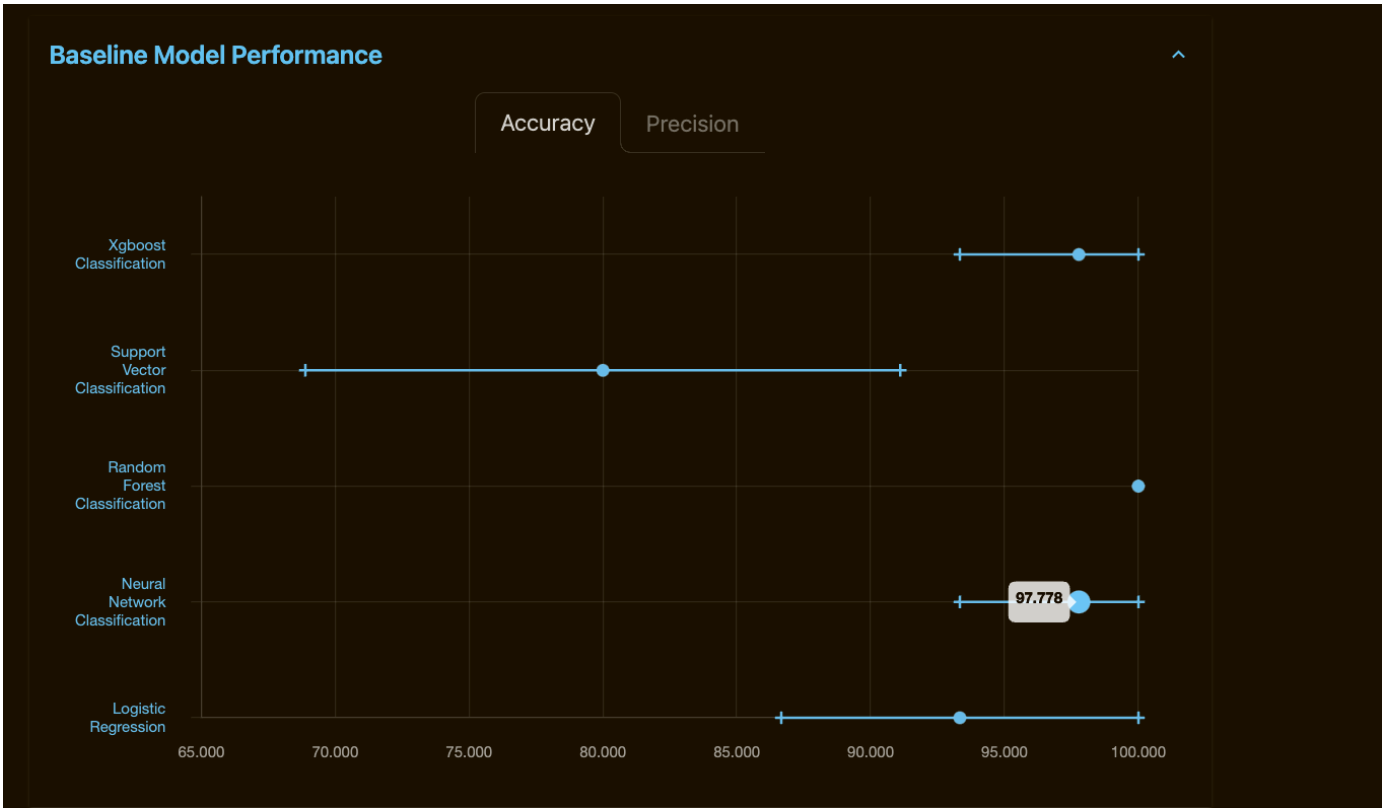

FIGURE 1 – L'utilisation des réseaux de neurones pour classifier le jeu de données Wine Dataset permet d'obtenir un taux de précision d'environ 97,78% (source : https://archive.ics.uci.edu/dataset/109/wine).

1.3.1 Caractéristiques du jeu de données

Le *Wine Dataset* présente les caractéristiques suivantes (source : https://archive.ics.uci.edu/dataset/109/wine) :

- **Type de données** : Données tabulaires
- **Domaine** : Physique et Chimie
- **Tâche associée** : Classification
- **Type des attributs** : Entiers et réels
- **Nombre d'instances** : 178 échantillons
- **Nombre de caractéristiques** : 13 attributs

- **Présence de valeurs manquantes** : Aucune

Les attributs du jeu de données sont les résultats d'analyses chimiques effectuées sur chaque vin. Voici la liste des attributs mesurés, chacun représentant une caractéristique chimique :

1. `Alcool` : Teneur en alcool du vin.
2. `Acide malique` : Quantité d'acide malique.
3. `Cendres` : Quantité de cendres présentes.
4. `Alcalinité des cendres` : Mesure de l'alcalinité des cendres.
5. `Magnésium` : Concentration de magnésium.
6. `Phénols totaux` : Quantité totale de phénols.
7. `Flavonoïdes` : Quantité de flavonoïdes.
8. `Phénols non-flavonoïdes` : Quantité de phénols non-flavonoïdes.
9. `Proanthocyanidines` : Concentration de proanthocyanidines.
10. `Intensité de la couleur` : Intensité de la couleur du vin.
11. `Teinte (Hue)` : Teinte du vin.
12. `OD280/OD315 des vins dilués` : Mesure d'absorption des vins dilués.
13. `Proline` : Concentration de proline (un acide aminé).

1.4 Compilation du programme et exemple

Nous avons préparé un makefile `Makefile` permettant la compilation et l'exécution du programme avec la commande `make`. Le programme n'attend aucun paramètre supplémentaire pour s'exécuter. La configuration du programme peut être modifiée dans le fichier `main.c` mais cela n'est pas nécessaire pour illustrer le bon fonctionnement de l'algorithme SOM avec le jeu de données `Wine.dataset`.

Les directives de compilation du Makefile sont : `gcc -Wall -Wextra main.c utils.c resultats.c apprentissage_SOM.c neural_net.c dataset.c -o SOM -lm`. Voici l'intégralité du Makefile :

```
# nom ......... : Makefile
# rôle........ : Compiler le projet et lancer l'exécutable SOM
# auteur ...... : Avrile Floro
# version ..... : v1 du 27/09/2024
# licence ..... : réalisé dans le cadre du cours d'IA
# usage :        pour compiler et exécuter : make
#          pour nettoyer : make clean

all: compile run

compile:
  gcc -Wall -Wextra main.c utils.c resultats.c apprentissage_SOM.c neural_net.c dataset.c -o
    SOM -lm

run:
  ./SOM

error:
  -./SOM arg_inutile

clean:
  rm -f SOM
```

Code Listing 1 – Code source du Makefile

Nous illustrons la compilation et l'exécution du programme avec la commande `make`. Le programme s'exécute et la grille représentant la classification des vins par l'algorithme SOM s'affiche. Ensuite, nous constatons que le pourcentage de justesse de la classification est de 97.75%. Cela correspond à l'exactitude de la classification et ce chiffre est très proche de la moyenne constatée sur ce jeu de données dans la littérature (établie à 97.78%).

```
gcc -Wall -Wextra main.c utils.c resultats.c apprentissage_SOM.c
neural_net.c dataset.c -o SOM -lm
./SOM
[1] = Cépage 1
[2] = Cépage 2
[3] = Cépage 3

 [1]  [1]  [1]  [1]  [3]  [3]  [3]  [3]  [3]  [3]
 [1]  [1]  [1]  [1]  [3]  [3]  [3]  [3]  [3]  [3]
 [1]  [1]  [1]  [1]  [3]  [3]  [3]  [3]  [3]  [3]
 [1]  [1]  [1]  [1]  [1]  [3]  [3]  [2]  [3]  [3]
 [1]  [1]  [1]  [1]  [1]  [2]  [3]  [2]  [3]  [3]
 [1]  [1]  [1]  [1]  [2]  [2]  [2]  [2]  [2]  [2]
 [1]  [1]  [1]  [1]  [2]  [2]  [2]  [2]  [2]  [2]
 [1]  [1]  [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]
 [1]  [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]
 [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]  [2]

Pourcentage de justesse des catégories des vins: 97.75%
avrile@vmubuntu:~/Desktop/SOM/Source$
```

FIGURE 2 – Nous lançons la compilation et l'exécution du programme avec la commande "make". La grille de classification obtenue est affichée et le taux d'exactitude de la classification obtenue est de 97.75%.

Nous illustrons également une erreur avec la commande `make error`, lorsque nous tentons de passer un argument au programme, qui n'en reçoit pas. Une erreur est renvoyée.

```
avrile@vmubuntu:~/Desktop/SOM/Source$ make error
./SOM arg_inutile
Erreur : Ce programme ne prend pas d'arguments.

make: [Makefile:18: error] Error 1 (ignored)
avrile@vmubuntu:~/Desktop/SOM/Source$
```

FIGURE 3 – Nous illustrons une erreur avec la commande "make error". Le programme ne reçoit pas d'arguments et renvoie une erreur si nous tentons de lui en passer un.

1.5 Détails des modules du programme

Nous allons détailler les différents modules du programme en présentant les fonctions qu'ils utilisent.

1.5.1 elements.h : Structures de données et déclarations des fonctions

Le fichier `elements.h` contient les bibliothèques nécessaires au programme, les structures de données utilisées et les déclarations des fonctions.

Bibliothèques utilisées. Voici la liste des bibliothèques utilisées :

```
/*# nom ........ : elements.h
 * rôle......... : Header pour les structures, les bibliothèques et les déclarations
 * auteur ...... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make        */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>     // pour sqrt
#include <float.h>    // pour FLT_MAX et FLT_MIN
#include <string.h>   // pour strtok
#include <time.h>     // pour time
```

Code Listing 2 – Code source du fichier elements.h

Structures de données. Le fichier `elements.h` contient également les structures de données utilisées par le programme. Elles vont permettre le fonctionnement du réseau de neurones SOM que nous utilisons afin de classifier les vins.

La structure `VecteurDonnees` représente un vecteur de données. Un vecteur de données est composé d'attributs de type flottant. Ces derniers représentent les attributs numériques décrivant les 13 caractéristiques des vins. En outre, chaque vecteur de données comprend un label (entre 1 et 3) qui indique le cépage (la classe) du vin.

```
// structure pour un vecteur de données
typedef struct {
    float* attributs; // tableau des attributs du vecteur
    int label;   // label associé (la classe du vin)
} VecteurDonnees;
```

Code Listing 3 – Code source du fichier elements.h

L'ensemble des vecteurs du jeu de données (qui représentent les vins étudiés) est encapsulé dans la structure `JeuDeDonnees`. Cette structure contient, outre les vecteurs, des éléments pertinents pour le bon fonctionnement du programme comme le nombre d'exemples contenus dans le jeu de données et la taille des vecteurs (le nombre de caractéristiques de chacun des vins). Par ailleurs, c'est également cette structure qui gère les indices (en ordre séquentiel et mélangé) pour le parcours des données du dataset.

```
// structure pour le jeu de données entier
typedef struct {
    int nb_exemples;      // nombre total d'exemples dans le dataset
    int taille_vecteur;    // nombre d'attributs par vecteur
    VecteurDonnees* exemples;   // tableau des vecteurs de données
    int* sequence_ordre_lignes;    // indices séquentiels des exemples
    int* shuffled_order_lignes; // indices mélangés des exemples
} JeuDeDonnees;
```

Code Listing 4 – Code source du fichier elements.h

Chaque neurone de la carte SOM est représenté par la structure `Neurone`. Cette dernière stocke les poids de chaque neurone ainsi qu'un label qui sera utilisé pour l'évaluation de la classification.

```c
// structure pour un neurone
typedef struct {
    float* poids;  // tableau des poids du neurone
    int label;    // étiquette associée au neurone
} Neurone;
```

Code Listing 5 – Code source du fichier elements.h

La structure `CarteNeurones` correspond à la carte du SOM, c'est-à-dire qu'elle organise l'ensemble des neurones en une grille bidimensionnelle dont les dimensions sont précisées. C'est également cette structure qui gère le traitement des indices séquentiels et mélangés pour les lignes et les colonnes.

```c
// structure pour une carte de neurones (SOM)
typedef struct {
    int lignes;    // nombre de lignes dans la carte
    int colonnes;   // nombre de colonnes dans la carte
    Neurone** neurones;   // grille 2D des neurones
    int* sequentiel_order_lignes;   // indices séquentiels pour les lignes
    int* sequentiel_order_col;    // indices séquentiels pour les colonnes
    int* shuffle_order_lignes; // indices mélangés pour les lignes
    int* shuffle_order_col;    // indices mélangés pour les colonnes
} CarteNeurones;
```

Code Listing 6 – Code source du fichier elements.h

Finalement, la structure `ReseauNeurones` comprend l'ensemble du réseau SOM et inclut, notamment, les paramètres d'apprentissage comme le nombre d'itérations, les taux d'apprentissage initiaux et actuels, les tailles initiales et actuelles du voisinage. Cette structure contient également une `CarteNeurones` et détaille les bornes minimales et maximales utilisées pour la génération aléatoire des poids des neurones.

```c
// structure pour le réseau de neurones
typedef struct {
    int nb_iterations;  // nombre total d'itérations pour l'apprentissage
    float taux_apprentissage_alpha_initial; // taux d'apprentissage initial
    float taux_apprentissage_alpha_actuel; // taux d'apprentissage courant
    float taille_voisinage_init;  // taille initiale du voisinage
    float taille_voisinage_actuelle;    // taille actuelle du voisinage
    CarteNeurones carte;   // la carte de neurones
    float min;  // borne inférieure pour la génération aléatoire
    float max;  // borne supérieure pour la génération aléatoire
    int taille_vecteurs;  // taille des vecteurs du réseau de neurones
} ReseauNeurones;
```

Code Listing 7 – Code source du fichier elements.h

Déclarations des fonctions. Finalement, le fichier `elements.h` contient l'ensemble des déclarations des fonctions utilisées par le programme.

```c
/* déclaration des fonctions */

// initialise les tableaux auxiliaires nécessaires au fonctionnement du programme
void initialisation_tableaux_annexes(JeuDeDonnees* data, ReseauNeurones* reseau);

// génère un float aléatoire entre deux bornes
float float_generator(float min, float max);

// génère la matrice de poids initiale pour le réseau de neurones
void init_carte_neurones(ReseauNeurones* reseau);

// génère un vecteur de floats aléatoires
float* vect_generateur(ReseauNeurones* reseau);

// normalise un vecteur pour qu'il ait une norme de 1
void vect_normalizer(float* vecteur, int taille_vecteur);

```

```c
// calcule la distance euclidienne entre deux vecteurs
float distance_euclidienne(const float* vect_entree, const float* vect_neurone, int
taille_vecteur);

// sélectionne le neurone le plus proche du vecteur d'entrée
Neurone* choix_BMU(float* v_entree, ReseauNeurones* reseau);

// mélange les indices du tableau dans un ordre aléatoire
void melanger_indices(int* indices_a_mixer, int taille_du_tableau);

// fonction principale d'apprentissage
void apprentissage(JeuDeDonnees* data, ReseauNeurones* reseau);

// met à jour les poids du neurone en utilisant la règle d'apprentissage
void regle_apprentissage(const float* v_entree, Neurone* neurone, Neurone* bmu,
                         ReseauNeurones* reseau);

// met à jour le taux d'apprentissage en fonction de l'itération courante
void mettre_a_jour_apprentissage_alpha(float iteration, ReseauNeurones* reseau);

// met à jour le rayon du voisinage en fonction de l'itération courante
void taille_voisinage_actif(float iteration, ReseauNeurones* reseau);

// lit le fichier de données et stocke les exemples et leurs labels
void lire_jeu_de_donnees(JeuDeDonnees* data, const char* nom_fichier);

// normalise chaque attribut des données entre 0 et 1
void normaliser_dataset(JeuDeDonnees* data);

// affiche les labels associés à chaque neurone de la carte
void afficher_resultats(ReseauNeurones* reseau);

// calcule le pourcentage de données correctement classifiées par le SOM
void calculer_pourcentage_justesse(JeuDeDonnees* data, ReseauNeurones* reseau);

// affiche un message d'erreur et termine le programme
void usage(const char* message);

// libère la mémoire du réseau de neurones et des données
void free_memoire(ReseauNeurones* reseau, JeuDeDonnees* data);
```

Code Listing 8 – Code source du fichier elements.h

1.5.2 main.c : Le coeur du programme

Le fichier `main.c` est le coeur de notre programme. C'est le fichier qui contient la fonction `main`. Cette fonction est en charge de l'ensemble du processus de classification.

La fonction `main` ne reçoit pas d'arguments. Une vérification a été mise en place afin de s'assurer qu'aucun argument n'est passé à la fonction.

```c
/*# nom ......... : main.c
 * rôle........ : Self Organizing Map (SOM)
 *                Dataset utilisé : wines
 *                Fonction principale (main)
 * auteur ...... : Avrile Floro
 * version ..... : v1.2 du 26/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage : pour compiler : gcc -Wall -Wextra main.c utils.c resultats.c
 *                          apprentissage_SOM.c neural_net.c dataset.c -o SOM
 *         pour compiler : make
 *         pour exécuter : ./SOM (compris dans make)
 */

#include "elements.h" // structures utilisées et définitions des fonctions


int main(int argc, char *argv[]) {

```

```c
    (void)argv; // pas d'argument - élimination de l'avertissement

    // vérifie qu'aucun argument n'est passé au programme
    if (argc > 1) {
        usage("Ce programme ne prend pas d'arguments.\n");
    }
```

Code Listing 9 – Code source du fichier main.c

La fonction `main` initalise la graine du générateur de `srandom` avec `time(NULL)` (source : https://stackoverflow.com/questions/1108780). La fonction `srandom()`, disponible nativement sous Unix, est préférée à la fonction `srand`, qui est moins efficace lorsqu'il s'agit de proposer des nombres effectivement aléatoires (source : https://stackoverflow.com/questions/18726102). En l'espèce, la fonction `random` sera utilisée à double titre : lors du mélange des indices d'après l'algorithme de Fisher-Yates et lors de la génération de flottants aléatoires conduisant à la création des neurones.

```c
    // initialise la graine du générateur aléatoire avec le PID du processus
    srandom(time(NULL));
```

Code Listing 10 – Code source du fichier main.c

La fonction `main` initialise les structures principales du programme, à savoir `data`, notre `JeuDeDonnees`, et `reseau` qui est un `ReseauNeurones`. Ces structures ont été présentées précédemment.

Suite à l'initialisation des structures de données, la fonction `main` définit les paramètres du réseau et des données. Sont déterminés, notamment, la taille des vecteurs (le nombre d'attributs pour chacun des vins), les bornes lors de la génération des flottants dans le cadre de la création des neurones, le nombre d'itérations (Epochs), la taille du voisinage initial et le taux d'apprentissage initial.

En outre, la carte SOM est définie d'après une dimension de 10 * 10.

```c
    // déclarations des structures de données
    JeuDeDonnees data;
    ReseauNeurones reseau;

    // initialise les paramètres du réseau et des données
    data.taille_vecteur = 13; // nombre d'attributs dans le jeu de données des vins
    reseau.taille_vecteurs = data.taille_vecteur; // utilisation pour le réseau
    reseau.min = 0.1f; // bornes pour la génération des vecteurs aléatoires
    reseau.max = 0.5f;
    reseau.nb_iterations = 1500; // nombre total d'itérations pour l'apprentissage
    reseau.taille_voisinage_init = 5.7f; // taille initiale du voisinage
    reseau.taux_apprentissage_alpha_initial = 0.865f; // taux d'apprentissage initial

    // taille de la carte SOM (10x10)
    reseau.carte.lignes = 10;
    reseau.carte.colonnes = 10;
```

Code Listing 11 – Code source du fichier main.c

Enfin, la fonction `main` va orchestrer l'ensemble du programme en permettant :

- La lecture et la normalisation du jeu de données à partir du fichier `wine.data`.
- L'initialisation de la matrice de poids des neurones.
- L'initialisation des tableaux auxiliaires permettant le fonctionnement du réseau.
- L'exécution de la phase d'apprentissage du réseau SOM.
- L'affichage des résultats obtenus après l'apprentissage.
- Le calcul et l'affichage du pourcentage de classification correcte des données.
- La libération de la mémoire avant de terminer le programme.

```
    // lit le jeu de données à partir du fichier et détermine le nombre d'exemples
    lire_jeu_de_donnees(&data, "wine.data");

    // normalise le jeu de données pour que chaque attribut soit entre 0 et 1
    normaliser_dataset(&data);

    // crée la matrice de poids initiale pour le réseau de neurones
    init_carte_neurones(&reseau);

    // initialise les tableaux auxiliaires
    initialisation_tableaux_annexes(&data, &reseau);

    // effectue l'apprentissage du réseau
    apprentissage(&data, &reseau);

    // affiche les résultats après l'apprentissage
    afficher_resultats(&reseau);

    // calcule et affiche le pourcentage de données correctement classifiées
    calculer_pourcentage_justesse(&data, &reseau);

    // libère la mémoire du réseau de neurones et des données
    free_memoire(&reseau, &data);

    return 0;
}
```

Code Listing 12 – Code source du fichier main.c

1.5.3 dataset.c : La gestion du jeu de données

Le fichier `dataset.c` gère la lecture et la préparation du jeu de données qui sera utilisé lors de l'entraînement du réseau de neurones.

Traitement du jeu de données. La fonction `lire_jeu_de_donnees` va lire les données à partir d'un fichier, stocker les exemples et leurs labels dans la structure `JeuDeDonnees`. Une fois le comptage terminé, la fonction repositionne le pointeur de lecture au début du fichier avec `rewind` pour permettre le traitement des données (source : https://www.tutorialspoint.com/c_standard_library/c_function_rewind.htm).

```
/*# nom ........ : dataset.c
 * rôle......... : Module en charge de la gestion du jeu de données (dataset: Wine)
 * auteur ...... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make  */

#include "elements.h"


/* lit le fichier de données et stocke les exemples et leurs labels */
void lire_jeu_de_donnees(JeuDeDonnees* data, const char* nom_fichier) {

    // ouvre le fichier en mode lecture
    FILE* fichier = fopen(nom_fichier, "r");
    if (!fichier) {
        usage("Erreur lors de l'ouverture du fichier");
    }

    // initialise un buffer pour la lecture des lignes
    char buffer[1024];

    // initialise le compteur de lignes à 0
    data->nb_exemples = 0;

    // compte le nombre de lignes dans le fichier (chaque ligne est un exemple)
    while (fgets(buffer, sizeof(buffer), fichier)) {
        data->nb_exemples++;
    }

```

```
    // revient au début du fichier pour la lecture des données
    rewind(fichier);
```

Code Listing 13 – Code source du fichier dataset.c

La fonction alloue d'abord de la mémoire pour stocker tous les exemples de données. Ensuite, pour chaque ligne du fichier, elle suit ces étapes :

- **Allocation mémoire** : Elle alloue de la mémoire pour les attributs de chaque exemple.
- **Extraction du label** : Elle extrait le label (première colonne) et le convertit en entier.
- **Traitement des attributs** : Elle récupère les attributs suivants, les convertit en flottants et, s'ils sont manquants, les initialise à 0.0.
- **Incrémentation** : Une fois les attributs extraits, elle passe à l'exemple suivant.

Une fois que tous les exemples ont été traités et stockés dans la structure `JeuDeDonnees`, la fonction ferme le fichier.

```
    // alloue de la mémoire pour stocker tous les exemples
    data->exemples = malloc(data->nb_exemples * sizeof(VecteurDonnees));
    if (!data->exemples) {
        usage("Erreur d'allocation mémoire pour les exemples");
    }

    int indice_ligne = 0;

    // lit chaque ligne du fichier
    while (fgets(buffer, sizeof(buffer), fichier)) {
        // récupère l'exemple correspondant à l'indice actuel dans le dataset (adresse)
        VecteurDonnees* exemple = &data->exemples[indice_ligne];

        // alloue de la mémoire pour les attributs de l'exemple
        exemple->attributs = malloc(data->taille_vecteur * sizeof(float));
        if (!exemple->attributs) {
            usage("Erreur d'allocation pour les attributs");
        }

        // d'abord, extrait le label (classe) de l'exemple [1ère colonne]
        char* token = strtok(buffer, ",");
        exemple->label = atoi(token); // convertit le token en int

        // extrait chaque attribut de l'exemple [à partir de la 2ème colonne]
        for (int i = 0; i < data->taille_vecteur; i++) {
            token = strtok(NULL, ",");
            if (token != NULL) {
                exemple->attributs[i] = atof(token); // convertit le token en float
            } else {
                exemple->attributs[i] = 0.0f; // si le token est NULL, attribut = 0.0
            }
        }
        indice_ligne++; // passe à l'exemple suivant
    }
    // ferme le fichier
    fclose(fichier);
}
```

Code Listing 14 – Code source du fichier dataset.c

Il convient de noter que le programme a été conçu pour analyser notre jeu de données. En effet, sans une connaissance préalable de la structure du dataset, il serait impossible de déterminer la position du label, ce qui influencerait également le nombre d'attributs. Puisque cette fonction a été développée sur mesure pour notre jeu de données, nous avons programmé en tenant compte de son organisation spécifique, où la catégorie se trouve en premier, suivie de treize attributs. Bien que le nombre total d'exemples ait été calculé dynamiquement par la fonction, il aurait également été possible de le définir de manière statique.

Normalisation des données du dataset. La fonction `normaliser_data` a pour but de normaliser les différents attributs des exemples du jeu de données entre 0 et 1. Pour ce faire, nous avons utilisé la normalisation `min-max`. Cette méthode de normalisation des attributs permet que toutes les valeurs se trouvent entre 0 et 1. Cela évite que les attributs ayant une valeur plus importante ne dominent le calcul des distances.

La fonction commence par allouer deux tableaux, `min` et `max`. Ils vont permettre le stockage des valeurs minimales et maximales de chaque attribut. Nous initialisons `min` avec `FLT_MAX` et `max` avec `-FLT_MAX`. Cela assure que les valeurs des attributs seront inférieures à `FLT_MAX` et supérieures à `-FLT_MAX`.

La fonction parcourt tous les exemples (lignes) du dataset pour trouver les valeurs minimales et maximales de chaque attribut. Lorsque ces valeurs ont été trouvées, la normalisation `min-max` est effectuée d'après la formule : $\text{attribut\_normalisé} = \frac{\text{attribut}-\text{min}}{\text{max}-\text{min}}$.
Nous évitons une division par 0 en définissant l'attribut à `0.0` lorsque la différence entre `max[j]` et `min[j]` est nulle.

```c
/* normalise chaque attribut des données entre 0 et 1 */
void normaliser_dataset(JeuDeDonnees* data) {

    // initialise les tableaux pour stocker les valeurs min et max de chaque attribut
    float* min = malloc(data->taille_vecteur * sizeof(float));
    float* max = malloc(data->taille_vecteur * sizeof(float));


    // initialise les min à FLT_MAX et les max à -FLT_MAX
    for (int i = 0; i < data->taille_vecteur; i++) {
        min[i] = FLT_MAX;
        max[i] = -FLT_MAX;
    }

    // parcourt tous les exemples pour trouver les min et max de chaque attribut
    for (int i = 0; i < data->nb_exemples; i++) {
        // isole l'exemple (avec son adresse)
        VecteurDonnees* exemple = &data->exemples[i];
        // parcourt chaque attribut de l'exemple
        for (int j = 0; j < data->taille_vecteur; j++) {
            if (exemple->attributs[j] < min[j]) {
                min[j] = exemple->attributs[j]; // met à jour le min si nécessaire
            }
            if (exemple->attributs[j] > max[j]) {
                max[j] = exemple->attributs[j]; // met à jour le max si nécessaire
            }
        }
    }

    // normalisation min-max des attributs [après avoir identifié min/max pour chaque]
    for (int i = 0; i < data->nb_exemples; i++) {
        // récupère chaque exemple
        VecteurDonnees* exemple = &data->exemples[i];
        // parcourt les attributs de chaque exemple
        for (int j = 0; j < data->taille_vecteur; j++) {
            if (max[j] - min[j] != 0) {
                // applique la formule de normalisation
                exemple->attributs[j] = (exemple->attributs[j] - min[j]) / (max[j] -
                        min[j]);
            } else { // (pour éviter une div par 0)
                exemple->attributs[j] = 0.0f; // si min == max, attribut normalisé à 0.0

            }
        }
    }

    // libération des tableaux utilisés
    free(min);
    free(max);
}
```

Code Listing 15 – Code source du fichier dataset.c

1.5.4 neural_net.c : Initialisation du réseau de neurones

Le module `neural_net.c` gère l'initialisation et la configuration du réseau de neurones.

La fonction `float_generator` utilise `random` afin de générer un flottant aléatoire entre des bornes minimum et maximum.

```c
/*# nom ........ : neural_net.c
 * rôle........ : Module en charge de l'initialisation du réseau de neurones
 * auteur ..... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make        */

#include "elements.h"

/* génère un float aléatoire entre deux bornes */
float float_generator(float min, float max) {
    // génère un float aléatoire entre min et max
    float rd_num = ((float)random() / (float)RAND_MAX) * (max - min) + min;
    return rd_num; // retourne le float généré
}
```

Code Listing 16 – Code source du fichier neural_net.c

La fonction `vect_generateur` génère un vecteur de flottants aléatoires. La fonction commence par allouer de la mémoire selon la taille du vecteur à générer. Ensuite, pour chaque composante du vecteur, un float aléatoire est généré à l'intérieur des bornes qui sont précisées. Le vecteur est retourné.

```c
/* génère un vecteur de floats aléatoires */
float* vect_generateur(ReseauNeurones* reseau) {
    int taille = reseau->taille_vecteurs;

    // alloue de la mémoire pour le vecteur
    float* vecteur = malloc(taille * sizeof(float));
    if (!vecteur) {
        usage("Erreur d'allocation pour le vecteur");
        exit(EXIT_FAILURE);
    }

    // génère des valeurs aléatoires pour chaque composante du vecteur
    for (int i = 0; i < taille; i++) {
        vecteur[i] = float_generator(reseau->min, reseau->max);
    }
    return vecteur;
}
```

Code Listing 17 – Code source du fichier neural_net.c

La fonction `vect_normalizer` calcule la norme d'un vecteur.
Si la norme est non nulle, chaque composante du vecteur est divisée par cette norme.

```c
/* normalise un vecteur pour qu'il ait une norme de 1 */
void vect_normalizer(float* vecteur, int taille_vecteur) {
    float norme = 0;

    // calcule la norme du vecteur
    for (int i = 0; i < taille_vecteur; i++) {
        norme += vecteur[i] * vecteur[i];
    }
    norme = sqrtf(norme);

    if (norme == 0) {
        usage("Erreur lors de la normalisation du vecteur");
    }

    // divise chaque composante du vecteur par la norme
    for (int i = 0; i < taille_vecteur; i++) {
        vecteur[i] /= norme;
```

```
    }
}
```

Code Listing 18 – Code source du fichier neural_net.c

La fonction `init_carte_neurones` initialise la carte de neurones du réseau. Elle alloue d'abord la mémoire pour la matrice de neurones (ici de dimensions 10 * 10).
Pour chaque neurone, la fonction :

- Alloue la mémoire pour le vecteur de poids.
- Génère un vecteur de poids aléatoires en appelant `vect_generateur`.
- Normalise ce vecteur avec `vect_normalizer` pour obtenir une norme de 1.
- Copie le vecteur normalisé dans les poids du neurone.
- Initialise le label du neurone (correspondant à la classe de vin, c'est-à-dire son cépage) à 0.

```
/* génère la matrice de poids initiale pour le réseau de neurones */
void init_carte_neurones(ReseauNeurones* reseau) {
    // récupère les infos sur la carte (nb de lignes, colonnes et taille des vecteurs)
    int nb_lignes = reseau->carte.lignes;
    int nb_colonnes = reseau->carte.colonnes;
    int taille_vecteurs = reseau->taille_vecteurs;

    // alloc. mémoire pour la matrice de neurones - lignes (tableau de pointeurs)
    reseau->carte.neurones = malloc(nb_lignes * sizeof(Neurone*));
    if (!reseau->carte.neurones) {
        usage("Erreur d'allocation pour la carte de neurones");
    }

    // parcourt chaque ligne de la carte (qu'on vient d'allouer)
    for (int i = 0; i < nb_lignes; i++) {
        // alloue de la mémoire pour les neurones de la ligne
        reseau->carte.neurones[i] = malloc(nb_colonnes * sizeof(Neurone));
        if (!reseau->carte.neurones[i]) {
            usage("Erreur d'allocation pour les neurones");
        }

        // parcourt chaque neurone de la ligne (qu'on vient d'allouer)
        for (int j = 0; j < nb_colonnes; j++) {
            // alloue de la mémoire pour les poids du neurone (qui sont des floats)
            reseau->carte.neurones[i][j].poids = malloc(taille_vecteurs * sizeof(float));
            if (!reseau->carte.neurones[i][j].poids) {
                usage("Erreur d'allocation pour les poids du neurone");
            }

            // génère un vecteur de poids aléatoires (random float)
            float* vecteur_random_genere = vect_generateur(reseau);

            // normalise le vecteur de poids pour avoir une norme de 1
            vect_normalizer(vecteur_random_genere, reseau->taille_vecteurs);

            // copie le vecteur normalisé dans les poids du neurone
            memcpy(reseau->carte.neurones[i][j].poids, vecteur_random_genere,
                   taille_vecteurs * sizeof(float));

            // libère la mémoire du vecteur temporaire
            free(vecteur_random_genere);

            // initialise le label du neurone à 0
            reseau->carte.neurones[i][j].label = 0;
        }
    }
}
```

Code Listing 19 – Code source du fichier neural_net.c

1.5.5 apprentissage_SOM.c : Entraînement du SOM

Le fichier `apprentissage_SOM.c` est en charge de l'entraînement de la carte SOM. Ce module définit les fonctions qui permettent de mélanger les données, de calculer les distances entre vecteurs, de sélectionner le neurone gagnant (BMU), de mettre à jour les poids des neurones, d'ajuster le taux d'apprentissage et la taille du voisinage, et de lancer l'algorithme d'apprentissage principal.

La fonction `melanger_indices` permet de mélanger aléatoirement les indices d'un tableau en utilisant l'algorithme de Fisher-Yates. Pour ce faire, la fonction va parcourir le tableau de la fin au début. Chaque élément (en commençant par le dernier) est échangé avec un élément précédent ou lui-même (source : https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle).

```c
/*# nom ........ : apprentissage_SOM.c
 * rôle......... : Module en charge de l'entraînement du SOM
 * auteur ...... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make        */

#include "elements.h"

/* mélange les indices du tableau dans un ordre aléatoire */
void melanger_indices(int* indices_a_mixer, int taille_du_tableau) {
    // algorithme de Fisher-Yates pour mélanger le tableau
    for (int i = taille_du_tableau - 1; i > 0; i--) {
        int j = random() % (i + 1);
        // échange les éléments i et j
        int temp = indices_a_mixer[i];
        indices_a_mixer[i] = indices_a_mixer[j];
        indices_a_mixer[j] = temp;
    }
}
```

Code Listing 20 – Code source du fichier apprentissage_SOM.c

La fonction `distance_euclidienne` permet de calculer la distance euclidienne entre deux vecteurs de même dimension. Le calcul de la distance euclidienne correspond à la racine carrée de la somme des carrés des différences entre les éléments correspondants des deux vecteurs.

```c
/* calcule la distance euclidienne entre deux vecteurs */
float distance_euclidienne(const float* vect_entree, const float* vect_neurone,
                           int taille_vecteur) {
    float somme = 0; // initialisation de la somme

    // calcule la somme des carrés des différences
    for (int i = 0; i < taille_vecteur; i++) {
        float diff = vect_entree[i] - vect_neurone[i];
        somme += diff * diff;
    }
    // retourne la racine carrée de la somme = la distance euclidienne
    return sqrtf(somme);
}
```

Code Listing 21 – Code source du fichier apprentissage_SOM.c

La fonction `choix_BMU` est centrale dans le processus d'apprentissage. Elle permet de sélectionner sur la carte SOM le neurone le plus proche du vecteur d'entrée, appelé BMU (pour Best Matching Unit).
Afin de garantir le traitement aléatoire des indices, la fonction commence par copier les indices séquentiels des lignes et des colonnes dans des tableaux distincts grâce à `memcpy`. Ensuite, la fonction `melanger_indices` est utilisée pour mélanger ces indices. Cette étape assure que les lignes et colonnes de la carte SOM seront parcourues de façon aléatoire, réduisant ainsi les risques de biais dans la sélection du premier BMU lorsque plusieurs BMU sont candidats.
La fonction parcourt tous les neurones du réseau pour trouver celui dont les poids minimisent la distance euclidienne avec le vecteur d'entrée. C'est-à-dire que le BMU sera le neurone dont la distance euclidienne avec le vecteur d'entrée du dataset est la plus faible. Le neurone sélectionné en tant que BMU sera renvoyé par la fonction.

```c
/* sélectionne le neurone le plus proche du vecteur d'entrée (= le BMU) */
Neurone* choix_BMU(float* v_entree, ReseauNeurones* reseau) {
    float distance_min = INFINITY;
    Neurone* bmu = NULL;

    // mélange les indices des lignes et colonnes de la carte
    // permet d'assurer que le BMU est choisi aléatoirement (si plusieurs)

    // initialise shuffle_order d'après seq_order avec memcpy(source, dest, taille)
    memcpy(reseau->carte.shuffle_order_lignes, reseau->carte.sequentiel_order_lignes,
           reseau->carte.lignes * sizeof(int));
    // mélange les indices de shuffle_order d'après l'algo de Fisher-Yates
    melanger_indices(reseau->carte.shuffle_order_lignes, reseau->carte.lignes);

    memcpy(reseau->carte.shuffle_order_col, reseau->carte.sequentiel_order_col,
           reseau->carte.colonnes * sizeof(int));
    melanger_indices(reseau->carte.shuffle_order_col, reseau->carte.colonnes);

    // parcourt tous les neurones de la carte pour déterminer le BMU
    for (int i = 0; i < reseau->carte.lignes; i++) {
        int idx_ligne = reseau->carte.shuffle_order_lignes[i];
        for (int j = 0; j < reseau->carte.colonnes; j++) {
            int idx_col = reseau->carte.shuffle_order_col[j];
            // récupère l'adresse du neurone
            Neurone* neurone = &reseau->carte.neurones[idx_ligne][idx_col];

            // calcule distance entre le vecteur d'entrée (dataset) et le neurone évalué
            float distance = distance_euclidienne(v_entree, neurone->poids, reseau->taille_
    vecteurs);

            // met à jour le BMU si une distance euclidienne plus petite est trouvée
            if (distance < distance_min) {
                distance_min = distance;
                bmu = neurone;
            }
        }
    }
    return bmu; // renvoie le neurone BMU
}
```

Code Listing 22 – Code source du fichier apprentissage_SOM.c

La fonction `regle_apprentissage` met à jour les poids des neurones dans le voisinage du BMU en fonction du taux d'apprentissage courant.
La fonction reçoit notamment parmi ses arguments le vecteur d'entrée du dataset, un neurone dont le poids doit peut-être être modifié et le BMU. Dans un premier temps, la fonction parcourt l'ensemble de la carte des neurones pour déterminer l'adresse du neurone qu'elle a reçu en argument et l'adresse du BMU. Étant donné que la carte SOM est une carte bidimensionnelle, l'adresse de ces deux neurones correspond à un point d'abscisse et à un point d'ordonnée sur la grille.

Ensuite, la fonction va calculer la distance physique entre chaque neurone et le BMU sur la carte SOM. La distance physique est mesurée dans l'espace de la carte, elle détermine à quel point un neurone est proche du BMU dans la grille. Nous avons adopté la simplification proposée par Kohonen et envisagé ici la distance physique de façon binaire. Si la distance physique est inférieure à la taille actuelle du voisinage (calculée de façon linéaire décroissante), alors les poids du neurone vont être modifiés pour se rapprocher du vecteur d'entrée en fonction, notamment, du taux d'apprentissage alpha. En revanche, si le neurone n'appartient pas au voisinage du BMU, alors ses poids ne seront pas modifiés.

```c
/* met à jour les poids du neurone en utilisant la règle d'apprentissage */
void regle_apprentissage(const float* v_entree, Neurone* neurone, Neurone* bmu,
                         ReseauNeurones* reseau) {
    // variables pour stocker les positions du neurone et du BMU
    int x_neurone = 0, y_neurone = 0, x_bmu = 0, y_bmu = 0;

    // trouve les positions du neurone courant et du BMU dans la carte
    for (int i = 0; i < reseau->carte.lignes; i++) {
        for (int j = 0; j < reseau->carte.colonnes; j++) {
            // compare les adresses (et non les valeurs)
            if (&reseau->carte.neurones[i][j] == neurone) {
                x_neurone = i;
```

```
                y_neurone = j;
            }
            if (&reseau->carte.neurones[i][j] == bmu) {
                x_bmu = i;
                y_bmu = j;
            }
        }
    }

    // calcule la distance PHYSIQUE entre le neurone et le BMU sur la carte
    float distance_physique = sqrt(pow(x_neurone - x_bmu, 2) + pow(y_neurone - y_bmu, 2));

    // vérifie si le neurone est dans le voisinage actif du BMU
    if (distance_physique < reseau->taille_voisinage_actuelle) {
        // met à jour les poids du neurone si c'est le cas (reste à l'identique sinon)
        for (int i = 0; i < reseau->taille_vecteurs; i++) {
            // ajuste poids vers valeurs du vecteur d'entrée selon taux d'apprentissage
            neurone->poids[i] += reseau->taux_apprentissage_alpha_actuel * (v_entree[i]
                    - neurone->poids[i]);
        }
    }
}
```

Code Listing 23 – Code source du fichier apprentissage_SOM.c

La fonction `mettre_a_jour_apprentissage_alpha` ajuste le taux d'apprentissage au fil des itérations. Là encore, nous avons suivi les recommandations de Kohonen et nous avons proposé deux phases d'apprentissage.
La diminution du taux d'apprentissage alpha est linéaire mais divisée en deux phases. Il y a tout d'abord une première phase de décroissance rapide. Le taux d'apprentissage va être divisé par 10 en seulement 20% des itérations totales. Enfin, il y a une seconde phase de diminution plus lente du taux d'apprentissage qui va, à partir du taux d'apprentissage obtenu à la fin de la première phase, de nouveau être divisé par 10 au cours des 80% d'itérations restantes.

```
/* fonction de réduction du taux d'apprentissage avec deux phases distinctes */
void mettre_a_jour_apprentissage_alpha(float iteration, ReseauNeurones* reseau) {
    // taux d'apprentissage alpha initial conservé dans une variable
    float alpha_initial = reseau->taux_apprentissage_alpha_initial;

    // définition des phases de réduction du taux d'apprentissage
    // phase 1 : 20% des itérations
    float nb_iter_phase_1 = (float)reseau->nb_iterations * 0.2f;
    // phase 2 : 80% des itérations
    float nb_iter_phase_2 = (float)reseau->nb_iterations - nb_iter_phase_1;

    // phase 1 : réduction linéaire rapide d'alpha > alpha_initial / 10
    if (iteration < nb_iter_phase_1) {
        reseau->taux_apprentissage_alpha_actuel = alpha_initial * (1.0f - (iteration /
                nb_iter_phase_1) * 0.9f);

        // phase 2 : réduction linéaire lente d'alpha_initial / 10 > alpha_initial / 100
    } else {
        reseau->taux_apprentissage_alpha_actuel = (alpha_initial / 10.0f) * (1.0f - (
                (iteration - nb_iter_phase_1) / nb_iter_phase_2) * 0.9f);
    }
}
```

Code Listing 24 – Code source du fichier apprentissage_SOM.c

La fonction `taille_voisinage_actif` réduit linéairement la taille du voisinage actif au fur et à mesure de l'apprentissage pour affiner la classification.
En outre, nous veillons à ce que la taille du voisinage ne descende jamais en dessous de 1 (ce qui correspond aux voisins directs du BMU sur la grille).

```
/* met à jour la taille du voisinage en fonction de l'itération courante */
void taille_voisinage_actif(float iteration, ReseauNeurones* reseau) {
    float voisinage_initial = reseau->taille_voisinage_init;
    float total_iter = (float)reseau->nb_iterations;

    // réduction linéaire de la taille du voisinage
    reseau->taille_voisinage_actuelle = voisinage_initial * (1 - (iteration /
```

```
            total_iter));

    // la taille du voisinage n'est jamais inférieure à 1.0
    if (reseau->taille_voisinage_actuelle < 1.0f) {
        reseau->taille_voisinage_actuelle = 1.0f;
    }
}
```

Code Listing 25 – Code source du fichier apprentissage_SOM.c

Finalement, la fonction `apprentissage` est la fonction principale qui gère l'ensemble du processus d'apprentissage sur le nombre d'itérations (Epochs) donné.
À chaque itération, les exemples du jeu de données sont mélangés. Ainsi, ils sont présentés de façon aléatoire au réseau de neurones. En outre, le taux d'apprentissage et la taille du voisinage actif sont ajustés lors de chaque itération. Nous avons vu précédemment que ces taux déclinent de façon linéaire (avec deux phases pour le taux d'apprentissage).

La fonction va parcourir chacun des exemples du jeu de données et en isoler le vecteur. Cela va permettre de sélectionner le neurone gagnant (BMU) qui en est le plus proche. Ainsi, un BMU est choisi pour chaque vecteur d'entrée. À chaque sélection d'un BMU, l'ensemble des neurones va être parcouru afin de voir leurs poids modifiés vers le vecteur d'entrée s'ils se trouvent effectivement dans le voisinage actif du BMU.

```
/* fonction principale d'apprentissage */
void apprentissage(JeuDeDonnees* data, ReseauNeurones* reseau) {

    // boucle principale sur le nombre d'itérations (EPOCHS)
    for (int iter = 0; iter < reseau->nb_iterations; iter++) {

        // mélange les exemples du dataset pour une présentation aléatoire
        // init. shuffle_order d'après seq_order grâce à memcpy(dest, source, taille)
        // puis mélange les indices du shuffle_order grâce à l'algorithme de Fisher-Yates
        memcpy(data->shuffled_order_lignes, data->sequence_ordre_lignes,
               data->nb_exemples * sizeof(int));
        melanger_indices(data->shuffled_order_lignes, data->nb_exemples);

        // met à jour le taux d'apprentissage alpha et la taille du voisinage
        // alpha: deux phases de baisse linéaire (rapide puis lente)
        mettre_a_jour_apprentissage_alpha((float)iter, reseau);
        taille_voisinage_actif((float)iter, reseau); // linéaire

        // parcourt chaque exemple du dataset
        for (int i = 0; i < data->nb_exemples; i++) {
                    // isole l'indice de l'exemple du dataset (numéro ligne)
            int indice_dataset = data->shuffled_order_lignes[i];
                    // récupère le vecteur associé à l'indice (adresse)
            VecteurDonnees* exemple = &data->exemples[indice_dataset];
            // récupère les attributs du vecteur exemple (et pas le label)
            float* entree = exemple->attributs;

            // trouve le BMU (neurone gagnant) pour l'exemple en cours du dataset
            Neurone* bmu = choix_BMU(entree, reseau);

            // associe le label de l'exemple du dataset courant au BMU
            bmu->label = exemple->label;

            // parcourt tous les neurones de la carte pour mettre à jour leurs poids
            for (int k = 0; k < reseau->carte.lignes; k++) {
                for (int l = 0; l < reseau->carte.colonnes; l++) {
                    // isole chaque neurone de la carte (grâce à son adresse)
                    Neurone* neurone = &reseau->carte.neurones[k][l];
                    // modification du poids du neurone
                    // SI distance physique avec BMU < taille du voisinage actuel
                    regle_apprentissage(entree, neurone, bmu, reseau);
                }
            }
        }
    }
}
```

Code Listing 26 – Code source du fichier apprentissage_SOM.c

1.5.6 resultats.c : Vérification et affichage des résultats

Le module `resultats.c` gère l'affichage des résultats et l'évaluation de la performance du réseau après l'apprentissage.

La fonction `afficher_resultats` affiche les labels associés à chaque neurone de la carte.
La fonction affiche une légende des classes (il y a trois cépages de vin). Elle parcourt la matrice de neurones et affiche, pour chaque neurone, son label dans une grille qui correspond à la disposition de la carte. Le label du neurone correspond au label du dernier vecteur d'entrée (exemple du dataset) pour lequel il a été désigné comme BMU.

```c
/*# nom ......... : resultats.c
 * rôle........ : Module en charge des résultats et de l'évaluation
 * auteur ..... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make        */

#include "elements.h"

/* affiche les labels associés à chaque neurone de la carte */
void afficher_resultats(ReseauNeurones* reseau) {
    // affiche la légende des classes (ici des cépages)
    printf("[1] = Cépage 1\n[2] = Cépage 2\n[3] = Cépage 3\n\n");

    // parcourt chaque neurone de la carte et affiche son label
    for (int i = 0; i < reseau->carte.lignes; i++) {
        for (int j = 0; j < reseau->carte.colonnes; j++) {
            printf(" [%d] ", reseau->carte.neurones[i][j].label);
        }
        printf("\n"); // nouvelle ligne après chaque ligne de la carte
    }
}
```

Code Listing 27 – Code source du fichier resultats.c

La fonction `calculer_pourcentage_justesse` permet de calculer le pourcentage d'exemples du dataset qui sont classifiés correctement par le SOM au cours d'une itération. Pour chaque exemple du jeu de données, la fonction :

- Trouve le neurone le plus proche (BMU) en utilisant la fonction `choix_BMU`.
- Compare le label de l'exemple avec celui du BMU.
- Incrémente un compteur si les labels correspondent.

Le pourcentage de justesse est ensuite calculé en divisant le nombre de correspondances correctes par le nombre total d'exemples.

```c
/* calcule le pourcentage de données correctement classifiées par le SOM */
void calculer_pourcentage_justesse(JeuDeDonnees* data, ReseauNeurones* reseau) {
    int correct_matches = 0;

    // parcourt chaque exemple du dataset
    for (int i = 0; i < data->nb_exemples; i++) {
        VecteurDonnees* exemple = &data->exemples[i];

        // trouve le BMU pour l'exemple courant
        Neurone* bmu = choix_BMU(exemple->attributs, reseau);

        // compare le label de l'exemple avec celui du BMU
        if (exemple->label == bmu->label) {
            correct_matches++; // incrémente le compteur si les labels correspondent
        }
    }

    // calcule le pourcentage de justesse
    float pourcentage = ((float)correct_matches / (float)data->nb_exemples) * 100.0f;
    printf("\nPourcentage de justesse des catégories des vins: %.2f%%\n", pourcentage);
```

```
}
```

Code Listing 28 – Code source du fichier resultats.c

1.5.7 utils.c : Les fonctions utilitaires

Le module `utils.c` comprend des fonctions utilitaires pour l'initialisation des tableaux, la gestion des erreurs et la libération de la mémoire.

La fonction `initialisation_tableaux_annexes` initialise les tableaux auxiliaires utiles pour le fonctionnement du programme.
Elle alloue et initialise :

- Les tableaux d'indices séquentiels et mélangés pour les lignes et colonnes de la carte SOM : `sequentiel_order_lignes`, `shuffle_order_lignes`, `sequentiel_order_col`, `shuffle_order_col`.
- Les tableaux d'indices séquentiels et mélangés pour les exemples du jeu de données : `sequence_ordre_lignes`, `shuffled_order_lignes`.

Ces tableaux sont utilisés pour parcourir la carte et le jeu de données de manière séquentielle ou aléatoire.

```c
/*# nom ......... : utils.c
 * rôle........ : Fonctions utilitaires (initialisation mémoire, usage et free_memoire)
 * auteur ...... : Avrile Floro
 * version ..... : v1 du 27/09/2024
 * licence ..... : réalisé dans le cadre du cours d'IA
 * usage ....... :  make        */


#include "elements.h"


/* initialise les tableaux auxiliaires nécessaires au fonctionnement du programme */
void initialisation_tableaux_annexes(JeuDeDonnees* data, ReseauNeurones* reseau) {

    // alloc. mémoire: tab. indices séquentiels et mélangés des lignes (carte SOM)
    reseau->carte.sequentiel_order_lignes = malloc(reseau->carte.lignes * sizeof(int));
    reseau->carte.shuffle_order_lignes = malloc(reseau->carte.lignes * sizeof(int));

    // alloc. mémoire: tab. indices séquentiels et mélangés des colonnes (carte SOM)
    reseau->carte.sequentiel_order_col = malloc(reseau->carte.colonnes * sizeof(int));
    reseau->carte.shuffle_order_col = malloc(reseau->carte.colonnes * sizeof(int));

    // vérifie si les allocations ont réussi
    if (!reseau->carte.sequentiel_order_lignes || !reseau->carte.sequentiel_order_col ||
        !reseau->carte.shuffle_order_lignes || !reseau->carte.shuffle_order_col) {
        usage("Erreur d'allocation pour les indices de la carte");
    }

    // initialise les indices séquentiels pour les lignes (carte SOM)
    for (int i = 0; i < reseau->carte.lignes; i++) {
        reseau->carte.sequentiel_order_lignes[i] = i;
    }

    // initialise les indices séquentiels pour les colonnes (carte SOM)
    for (int i = 0; i < reseau->carte.colonnes; i++) {
        reseau->carte.sequentiel_order_col[i] = i;
    }

    // alloc. mémoire pour les indices séquentiels et mélangés (exemples du dataset)
    data->sequence_ordre_lignes = malloc(data->nb_exemples * sizeof(int));
    data->shuffled_order_lignes = malloc(data->nb_exemples * sizeof(int));

    // vérifie si les allocations ont réussi (exemples du dataset)
    if (!data->sequence_ordre_lignes || !data->shuffled_order_lignes) {
        usage("Erreur d'allocation pour les indices du dataset");
    }

    // initialise les indices séquentiels (exemples du dataset)
```

```
    for (int i = 0; i < data->nb_exemples; i++) {
        data->sequence_ordre_lignes[i] = i;
    }
}
```

Code Listing 29 – Code source du fichier utils.c

La fonction `usage` affiche un message d'erreur sur la sortie d'erreur et termine le programme avec une erreur.

```
/* affiche un message d'erreur et termine le programme */
void usage(const char* message) {
    fprintf(stderr, "Erreur : %s\n", message);
    exit(EXIT_FAILURE);
}
```

Code Listing 30 – Code source du fichier utils.c

La fonction `libere_memoire` libère la mémoire allouée pour le réseau de neurones et le jeu de données. Pour les structures à plusieurs dimensions, la fonction parcourt chaque niveau.

```
/* libère la mémoire du réseau de neurones et des données */
void free_memoire(ReseauNeurones* reseau, JeuDeDonnees* data) {
    // libère la mémoire de la carte de neurones
    for (int i = 0; i < reseau->carte.lignes; i++) {
        for (int j = 0; j < reseau->carte.colonnes; j++) {
            free(reseau->carte.neurones[i][j].poids); // libère les poids du neurone
        }
        free(reseau->carte.neurones[i]); // libère la ligne de neurones
    }
    free(reseau->carte.neurones); // libère le tableau de lignes
    free(reseau->carte.sequentiel_order_lignes);
    free(reseau->carte.sequentiel_order_col);
    free(reseau->carte.shuffle_order_lignes);
    free(reseau->carte.shuffle_order_col);

    // libère la mémoire des exemples du dataset
    for (int i = 0; i < data->nb_exemples; i++) {
        free(data->exemples[i].attributs); // libère les attributs de l'exemple
    }
    free(data->exemples); // libère le tableau d'exemples (de lignes)
    free(data->sequence_ordre_lignes);
    free(data->shuffled_order_lignes);
}
```

Code Listing 31 – Code source du fichier utils.c

1.6 Conclusion

Dans le cadre de ce projet, nous avons implémenté et exécuté un algorithme de carte auto-organisée (SOM) en C afin de classer les vins du jeu de données `Wine Dataset`. Nous avons présenté les étapes principales du programme, allant de la lecture et de la normalisation des données à l'entraînement du réseau de neurones et à l'évaluation des résultats.

Nous avons choisi une approche modulaire avec des fonctions pour chaque tâche (lecture, normalisation, apprentissage et affichage des résultats). Cela a permis au programme de rester clair et accessible.

L'utilisation de l'algorithme SOM nous a permis de représenter les données multidimensionnelles (13 attributs) de manière bidimensionnelle sur la grille, tout en atteignant un taux de classification correct d'environ 97,7%, en ligne avec les résultats annoncés pour ce jeu de données.