UNIVERSITÉ DE POITIERS

UFR Faculté des Sciences

Licence 3 Informatique – Génie Logiciel

Rapport de Projet GL – Réseau de neurones

Présente par

El Mahdi Benfdal

Ewen Croizier

Table des matières

I. Introduction

1. Présentation et explication du projet.

Dans le cadre de ce projet de Génie Logiciel, nous avons conçu une application en C, utilisant la bibliothèque SDL2, permettant de modéliser, entraîner et visualiser un réseau de neurones multicouches. L'objectif principal était de classifier des points répartis en différentes populations, en se basant initialement sur deux spirales distinctes – l'une rouge, l'autre bleue – générées à partir d'équations paramétriques.

Ce projet nous a permis d'explorer les fondements du machine learning supervisé, en mettant en œuvre l'algorithme de rétropropagation du gradient et en développant des structures de données personnalisées pour représenter les neurones, les couches et le réseau dans son ensemble. L'interface graphique réalisée avec SDL2 permet à l'utilisateur d'interagir directement avec les données et d'observer les résultats d'apprentissage en temps réel.

Cependant, les premières versions du système ont mis en évidence plusieurs limites : imprécisions dans la génération des spirales, erreurs dans l'affichage graphique, et comportements inattendus du réseau lors de l'apprentissage. Ces constats nous ont poussés à revoir l'architecture du code, à restructurer certaines parties du réseau neuronal, et à intégrer des outils de sauvegarde, de test et de benchmarking pour fiabiliser l'ensemble.

Ce rapport retrace l'ensemble de notre démarche, de la conception initiale aux optimisations finales, en passant par :

2. la création progressive du système neuronal et son adaptation aux données spirales,
3. les améliorations graphiques et fonctionnelles apportées au dessin et à la visualisation,
4. la gestion des sauvegardes et la modularité du code,
5. les tests unitaires, les logs et les benchmarks mis en place pour valider les performances.

II. Décomposition des étapes

1. Création du système neuronal

La première étape de notre projet a consisté à concevoir la structure interne du réseau de neurones. Celui-ci repose sur une architecture multicouche configurable, permettant à l'utilisateur de définir le nombre de couches cachées et le nombre de neurones dans chacune.

Le réseau est représenté par une structure ReseauNeuronal, contenant :

- le nombre d'entrées initiales (n_entrees),
- le nombre de couches cachées et de sortie (n_couches),
- un tableau dynamique indiquant le nombre de neurones par couche (taille_couches),
- et un tableau contenant chaque couche du réseau, elle-même composée de neurones.

Chaque **neurone** est défini comme un objet Neurone contenant :

- un tableau de poids synaptiques,
- un champ sortie correspondant à la valeur actuelle calculée par le neurone,
- un champ delta utilisé lors de l'apprentissage par rétropropagation.

L'ensemble du réseau est initialisé via la fonction creer_reseau(). Celle-ci :

- alloue dynamiquement les couches et les neurones,
- génère aléatoirement les poids dans l'intervalle [−1,1][-1, 1][−1,1], en utilisant /dev/urandom si disponible (sinon time(NULL)),
- initialise les sorties et deltas à zéro.

Une fois le réseau créé, la **fonction de propagation** (propagation) permet de propager les entrées vers la sortie du réseau, en appliquant la fonction d'activation **tanh** à chaque neurone. Cette fonction simule le comportement d'un neurone biologique en produisant une sortie non linéaire en fonction des entrées pondérées.

Enfin, l'**apprentissage** est assuré par l'algorithme de **rétropropagation du gradient** implémenté dans backpropagation(). Cet algorithme suit trois étapes principales :

1. Calcul du **delta** pour chaque neurone de la couche de sortie, à partir de l'écart entre la sortie réelle et la sortie attendue.
2. Propagation de l'erreur en remontant couche par couche, en utilisant les poids synaptiques pour évaluer l'influence des neurones précédents.
3. Mise à jour des poids de chaque neurone selon la formule :

$$w_{ij} = w_{ij} + \varepsilon \cdot \delta_i \cdot x_j$$

où ε est le taux d'apprentissage, δi le delta du neurone i.i et xj la valeur de l'entrée J.

Grâce à cette structure modulaire, le réseau peut facilement être adapté à différents jeux de données, comme les spirales à deux classes, ou des données personnalisées multi-classes. Cette base solide nous a permis de poursuivre le développement du reste du projet : visualisation graphique, sauvegardes, tests et benchmarks

2. Adaptation du format neuronal sur une donnée

Explication de la méthode utilisée, avec l'exemple des données de la spirale.

Une fois le système neuronal en place, il a fallu l'adapter à un jeu de données réel. Pour cela, nous avons utilisé comme jeu de test les spirales rouges et bleues générées mathématiquement à l'aide d'équations paramétriques, représentant deux classes séparables dans un plan.

Le module data.c est chargé de générer ces jeux de données via la fonction generer_spirales_archimede(). Chaque spirale est composée de **NB_POINTS_SPIRALE** points. Pour chaque point, ses coordonnées (x, y) sont calculées à partir du paramètre angulaire t, et normalisées dans l'intervalle [−1,1][-1, 1][−1,1]. Les classes sont encodées au format *one-hot*, par exemple :

- [1.0, 0.0] pour un point bleu (classe 1)
- [0.0, 1.0] pour un point rouge (classe 2)

Le réseau est ensuite entraîné sur ces données en propagant les coordonnées comme entrée (x, y) et en comparant la sortie du réseau au label attendu via la fonction backpropagation().

```
// Génération de la spirale bleue (classe 1)
dataset[index].x = (r * cos(t)) / scale;
dataset[index].y = (r * sin(t)) / scale;
dataset[index].label[0] = 1.0;   // bleu
dataset[index].label[1] = 0.0;
```

Fichier data.c

Dans la boucle d'apprentissage, chaque point est sélectionné aléatoirement dans le dataset, puis envoyé dans le réseau pour une propagation suivie d'une rétropropagation selon son label.

Ce format de données a permis de tester concrètement le comportement du réseau et d'afficher une **frontière de décision** en temps réel, observée via la fonction colorier_ecran().

Visualisation graphique

La sortie du réseau est une paire de valeurs entre −1 et 1, que nous avons convertie en probabilités [0,1][0, 1][0,1] pour afficher un rendu coloré :

```
// On convertit la sortie de tanh (∈[-1,1]) en probabilité ∈[0,1]
double p_bleu  = (s_bleu  + 1.0) / 2.0;
double p_rouge = (s_rouge + 1.0) / 2.0;
```

Ces probabilités sont ensuite envoyées dans la fonction melange_couleurs(), qui calcule une couleur RGB pour chaque pixel du rendu, permettant de visualiser les régions dominées par chaque classe.

3. Spirale normale et modélisation

Sous-partie sur la spirale normale.

Initialement, nous avions généré les spirales à l'aide d'équations paramétriques simples :

- **Spirale bleue :**
  $x = t \times \cos(t)$
  $y = t \times \sin(t)$
- **Spirale rouge (opposée) :**
  $x = -t \times \cos(t)$
  $y = -t \times \sin(t)$

Cette méthode présentait deux problèmes majeurs : une séparation imparfaite des spirales et une convergence difficile lors de l'apprentissage du réseau. Pour y remédier, nous avons introduit la **spirale d'Archimède** définie par :

$$r = a + b \times t, \quad x = r \times \cos(t), \quad y = r \times \sin(t)$$

avec un décalage d'angle de π\piπ pour générer la spirale opposée. Cette modification a permis d'améliorer significativement la distribution des points et la précision de la frontière de décision

Ajout des classes qui modulent les couches neuronales avec un certain nombre d'entrées et de sorties.

En parallèle de l'amélioration de la génération des spirales, nous avons rendu le réseau neuronal plus modulable en permettant la configuration dynamique du nombre d'entrées, de couches cachées et de neurones en sortie.

- Pour la classification binaire (spirales rouge et bleue), le réseau est typiquement configuré avec 2 entrées (pour xxx et yyy), plusieurs couches cachées (par exemple, 10 neurones par couche) et 2 neurones en sortie.

- Pour des scénarios plus complexes (comme un rendu en RGB ou une classification multi-classes), le nombre de sorties est ajusté en conséquence (par exemple, 3 neurones pour un affichage en couleurs).

Cette modularité a été implémentée via la fonction creer_reseau(), qui permet d'allouer dynamiquement le réseau en fonction des paramètres fournis. Un exemple de création de réseau pour une classification R/B est le suivant :

```
    // 2 entrées => 5 couches => 10 neurones par couche cachée => 2 neurones en sortie
    reseau = creer_reseau(2, 5, (int[]){10, 10, 10, 10, 2});
```

Fichier ui_Manger.c

et pour un modèle RGB :

```
// (2) Créer un réseau 2 entrées (x, y) -> 3 sorties (R, G, B)
//     + 3 ou 4 couches cachées de 10 neurones, au choix.
int taille_couches[] = {10, 10, 10, 10, 3};
ReseauNeuronal *reseau = creer_reseau(2, 5, taille_couches);
```

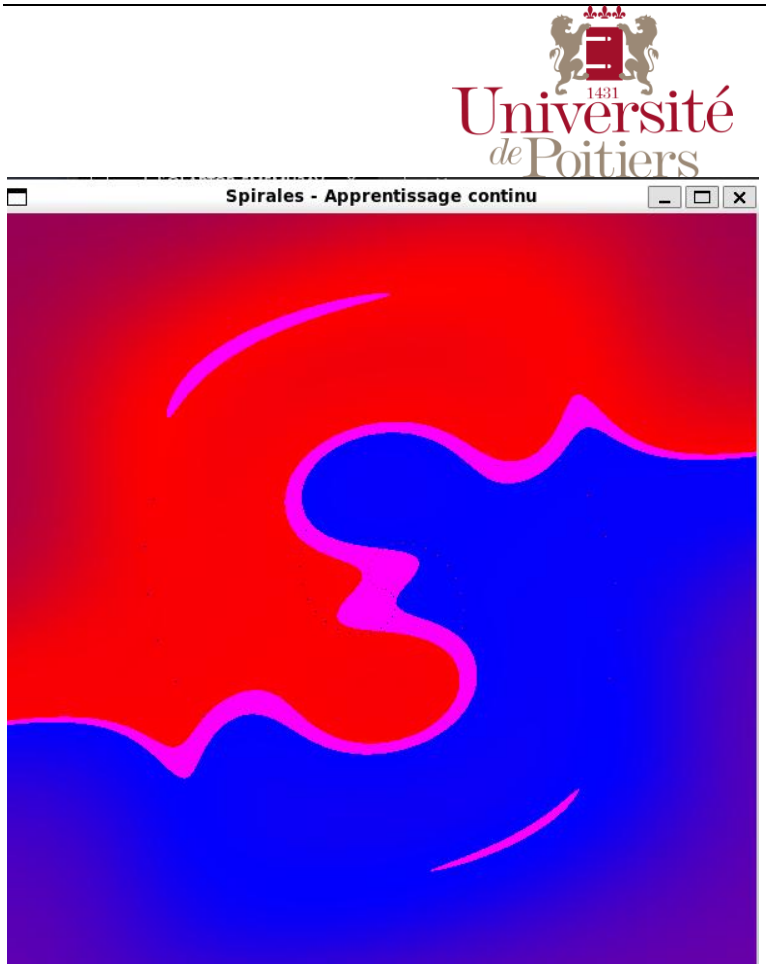
Explication des étapes avancées réalisées

Version 1 :

Initialement, nous avions utilisé les équations paramétriques suivantes pour générer les spirales :
Spirale bleue : x = t × cos(t), y = t × sin(t)
Spirale rouge (opposée) : x = -t × cos(t), y = -t × sin(t)

Cette méthode présentait deux problèmes majeurs : séparation imparfaite des spirales et convergence difficile.

Comme illustré dans l'image ci-dessous correspondant à la version 1



Voir sur git :

https://github.com/kaaix/reseau-neurones-spirales/commit/59c48f5da4e2ea9ce75210b0ee6505057e68f7c3

Version 2 Spirale d'Archimède :

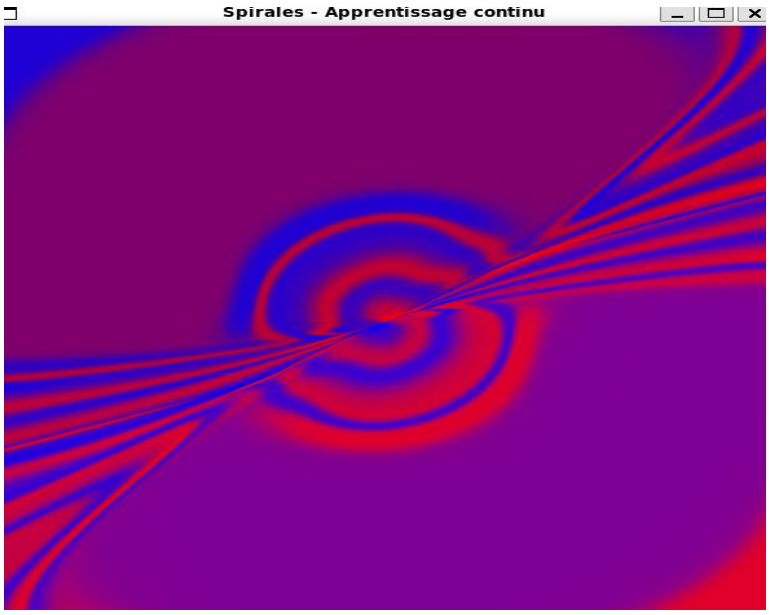
Nous avons introduit la spirale d'Archimède définie par : r = a + b × t, x = r × cos(t), y = r × sin(t), avec un décalage d'angle π pour générer la deuxième spirale. Ces modifications ont amélioré la distribution des points et la précision de la frontière de décision.

Néanmoins, le résultat obtenu demeurait insatisfaisant, l'affichage étant perturbé par un mauvais réglage du paramètre scale, lié à des choix inadaptés concernant la normalisation et le paramétrage angulaire.

```
int index = 0;
double a = 0.0;         // Décalage initial (tu peux ajuster)
double b = 1.0;         // Contrôle l'espacement entre les spires
// Choisis un facteur de normalisation pour ramener les points dans [-1,1]
double scale = NB_POINTS_SPIRALE * 0.1;  // Ajuste cette valeur si besoin
```

La capture ci-contre illustre les paramètres de la fonction void generer_spirales_archimede(), ayant permis d'obtenir les résultats de la version 2. Les ajustements finaux, menant au rendu de la version 3, y seront présentés.

Comme illustré dans l'image ci-dessous correspondant à la version 2:



Voir git :

https://github.com/kaaix/reseau-neurones-spirales/commit/f6542a4615b4962338a63a843532e66d627b8cca

Version 3 Ajustements fins des paramètres :

Dans cette troisième itération, nous avons apporté des ajustements précis aux paramètres clés pour optimiser la visibilité et la séparation des spirales. Plus spécifiquement :

Explication des modifications de la version 3 et rappel des points clés :

Dans cette troisième version, nous avons principalement ajusté deux paramètres importants :

1. **Le facteur de normalisation (scale) :** La valeur choisie pour scale affecte directement la taille des spirales affichées. Nous avons testé diverses valeurs (10.0, 12.0, 15.0, etc.) afin d'obtenir un compromis idéal où les points ne sont ni trop concentrés au centre ni trop éloignés sur les bords, assurant une répartition homogène.

2. **Le paramètre d'espacement (b) :** Dans la spirale d'Archimède définie par r =a+b*t, le paramètre b contrôle l'espacement entre les tours. Nous avons retenu une valeur optimale (b = 0.5) permettant une séparation nette des boucles tout en évitant un écartement trop rapide ou une accumulation excessive des points.

   Comme indiqué précédemment, les paramètres suivants sont ceux qui permettent un fonctionnement correct dans la version 3 :

```
int index = 0;
double a = 0.0;        // Décalage initial (tu peux ajuster)
double b = 0.5;        // Contrôle l'espacement entre les spires
// Choisis un facteur de normalisation pour ramener les points dans [-1,1]
double scale = 10.0 ; // Ajuste cette valeur si besoin
```

Voici un extrait optimisé du code correspondant :

```
double t = i * 0.05;          // Paramètre angulaire
double r = a + b * t;        // Rayon selon la spirale d'Archimède
```

L'objectif principal de ces ajustements fins était d'obtenir des spirales clairement définies et une frontière de décision nette, sans artefacts visuels tels que lignes horizontales ou arcs superposés.

Voir git :

https://github.com/kaaix/reseau-neurones-spirales/commit/86481522d162912ad970767bc5ef929b52bd3972

4. Sauvegardes et gestion des fichiers

Pour permettre la réutilisation et la reprise de l'entraînement du réseau de neurones, nous avons implémenté un système de sauvegarde et de chargement basé sur l'écriture et la lecture de fichiers binaires. Ce système se décline en deux parties principales :

Présentation des fichiers de sauvegarde de la spirale classique et de la spirale avec classes.

La fonction sauvegarder_reseau() permet d'enregistrer la structure complète du réseau dans un fichier binaire. Ce fichier contient :

- Les **métadonnées** du réseau : le nombre d'entrées, le nombre de couches, et la taille de chaque couche.
- Les **poids** de chaque neurone, qui déterminent le comportement du réseau lors de la propagation.

L'extrait suivant illustre l'écriture des métadonnées et des poids dans le fichier :

```
// Sauvegarde des métadonnées
fwrite(&reseau->n_entrees, sizeof(int), 1, f);
fwrite(&reseau->n_couches, sizeof(int), 1, f);
fwrite(reseau->taille_couches, sizeof(int), reseau->n_couches, f);

// Sauvegarde des poids et des biais
for (int i = 0; i < reseau->n_couches; i++) {
    for (int j = 0; j < reseau->taille_couches[i]; j++) {
        Neurone *neurone = &reseau->couches[i][j];
        fwrite(neurone->poids, sizeof(double), neurone->n_entrees, f);
    }
}
```

La fonction charger_reseau() lit les métadonnées sauvegardées pour reconstituer dynamiquement la structure du réseau. Après avoir alloué la mémoire nécessaire (grâce à la fonction creer_reseau()), elle lit les poids enregistrés pour remettre le réseau dans l'état exact de la sauvegarde.

Voici un extrait illustrant la lecture des métadonnées :

```c
// Lecture des métadonnées
int n_entrees, n_couches;
if (fread(&n_entrees, sizeof(int), 1, f) != 1 ||
    fread(&n_couches, sizeof(int), 1, f) != 1) {
    perror("Erreur lors de la lecture des métadonnées");
    fclose(f);
    log_message(LOG_LEVEL_INFO, "Réseau sauvegardé avec succès dans '%s'.", nom_fichier);
    return NULL;
}

int *taille_couches = malloc(n_couches * sizeof(int));
if (fread(taille_couches, sizeof(int), n_couches, f) != (size_t)n_couches) {
    perror("Erreur lors de la lecture des tailles des couches");
    free(taille_couches);
    fclose(f);
    log_message(LOG_LEVEL_INFO, "Réseau sauvegardé avec succès dans '%s'.", nom_fichier);
    return NULL;
}

// Création du réseau
```

Explication des problèmes rencontrés avec les sauvegardes personnalisées, non résolus en raison d'un manque de personnel dans le groupe (l'essentiel a été réalisé).

Bien que le mécanisme de sauvegarde et de chargement fonctionne correctement pour le cas standard (spirale classique ou modèle binaire R/B) – avec un chargement par défaut opérationnel – certaines fonctionnalités avancées rencontrent encore des difficultés. En particulier :

- **Chargement en mode "Custom"** :
  Alors que le chargement par défaut (load default) fonctionne comme prévu, le mode "Custom" pour gérer un nombre variable de classes ou des paramètres personnalisés ne permet pas toujours de reconstituer correctement la configuration du réseau. Cela peut entraîner des erreurs lors de la reprise de l'entraînement en mode personnalisé.

- **Sauvegarde du dessin** :
  La fonctionnalité permettant de sauvegarder le dessin réalisé par l'utilisateur n'est pas encore opérationnelle. Bien que l'interface de dessin soit fonctionnelle et permette de créer un jeu de données personnalisé, la persistance de ces données via un fichier de sauvegarde n'a pas pu être finalisée en raison de contraintes de temps et de ressources.

Néanmoins, l'essentiel du système – c'est-à-dire la sauvegarde et le chargement du réseau neuronal principal via les modes par défaut ainsi que les fonctions d'écriture et de lecture d'entiers pour stocker des paramètres simples – est opérationnel. Ces limitations sont bien documentées dans notre historique Git et pourront être améliorées lors d'évolutions futures du projet.

5. Dessin graphique

Explication de la refonte du dessin graphique.

Pour améliorer l'expérience utilisateur et offrir une visualisation claire des résultats, nous avons refondu le module d'affichage graphique en utilisant la bibliothèque SDL2. La nouvelle implémentation offre une interface plus fluide et interactive pour :

- **Afficher les spirales de référence** : La fonction afficher_spirales() dessine les spirales bleue et rouge en utilisant des équations paramétriques. Cela permet à l'utilisateur de comparer visuellement le dataset généré avec la frontière de décision du réseau.
- **Afficher la frontière de décision** : En évaluant pixel par pixel la sortie du réseau neuronal, la fonction colorier_ecran() colore l'écran selon les probabilités pour les classes (rouge et bleu). La refonte a permis d'assurer une meilleure qualité graphique et une mise à jour en temps réel lors de l'apprentissage.
- **Interface de dessin personnalisé** : Un module dédié permet à l'utilisateur de dessiner ses propres points via une interface graphique intuitive. Les zones de dessin, les curseurs de réglage de couleur, et les boutons interactifs (ex. « Effacer » et « Valider ») ont été repensés pour une utilisation plus ergonomique.

Ces améliorations ont conduit à une expérience utilisateur améliorée, où l'ensemble des opérations graphiques (création, modification, et visualisation) est plus fluide et mieux intégré à l'apprentissage du réseau.

Comparaison entre les fonctions void colorier_ecran et void colorier_ecran_rgb.

Deux fonctions principales gèrent le rendu des résultats du réseau :

- **colorier_ecran**
  Cette fonction est utilisée pour les cas de classification binaire (deux classes : rouge et bleu).
  - Elle effectue une **propagation** à partir des entrées pixel par pixel pour obtenir deux sorties.

- Les sorties, qui varient entre -1 et 1, sont converties en probabilités (entre 0 et 1) puis mélangées à l'aide de la fonction melange_couleurs().
- Le rendu se fait en attribuant à chaque pixel une couleur basée sur la dominance de l'une ou l'autre classe.
- Cette approche est particulièrement adaptée pour les modèles simples où l'on compare directement deux classes.

- **`colorier_ecran_rgb`**
  Conçue pour les modèles multi-classes (par exemple, en mode RGB), cette fonction :
  - Effectue également une propagation sur chaque pixel.
  - Récupère trois sorties correspondant aux trois canaux (R, G, B).
  - Convertit les sorties (variant de -1 à 1) en valeurs de couleur en utilisant `melange_couleurs_rgb()`, permettant ainsi de visualiser des mélanges de couleurs pour représenter l'appartenance aux différentes classes.
  - Cette fonction est idéale lorsque le réseau doit gérer plus de deux classes et produire un rendu en couleur plus complexe.

En résumé, la fonction `colorier_ecran` est optimisée pour une classification binaire, tandis que `colorier_ecran_rgb` offre une solution plus flexible pour la classification multi-classes en utilisant des mélanges RGB.

Sous-partie sur la sauvegarde du dessin (non réalisée) avec des propositions de solutions.

Bien que l'interface de dessin permette à l'utilisateur de créer un dataset personnalisé en temps réel, la fonctionnalité de **sauvegarde du dessin** n'a pas encore été implémentée. Voici quelques pistes pour intégrer cette fonctionnalité dans de futures itérations :

- **Sauvegarde dans un fichier binaire ou texte** :
  - On pourrait enregistrer le tableau dessin (contenant les points dessinés et leur label) dans un fichier binaire, en utilisant un format similaire à celui utilisé pour la sauvegarde du réseau.
  - Alternativement, un format texte (CSV ou JSON) pourrait être employé pour faciliter la lecture et l'analyse manuelle des données.

Fonction d'exportation dédiée :

- Développer une fonction, par exemple sauvegarder_dessin(), qui itère sur le tableau dessin et écrit les coordonnées et les labels de chaque point dans un fichier.
- Une fonction complémentaire, charger_dessin(), permettrait de recharger ces données pour réafficher le dessin ou pour réutiliser le dataset personnalisé lors d'un entraînement ultérieur.

Interface utilisateur :

- Intégrer dans l'interface de dessin un bouton "Sauvegarder le dessin" qui déclencherait ces fonctions d'exportation.
- Fournir également un moyen de charger un dessin sauvegardé pour permettre une expérimentation continue sans perdre les données créées.

6. Répartition du projet

Description de la répartition des tâches et de l'organisation du projet.

Le projet a été réalisé en binôme, avec une répartition des tâches définie comme suit :

- **El Mahdi Benfdal**
  - Conception et implémentation du module de réseau neuronal (création, propagation, rétropropagation).
  - Gestion des interactions utilisateur dans le module UI (ui_manager.c / ui_manager.h).
  - Gestion des benchmarks (benchmark.c / benchmark.h) pour évaluer la performance des algorithmes.
  - Coordination de l'intégration globale, gestion du Git et développement du fichier main.
- **Ewen Croizier**
  - Conception et intégration des interfaces de dessin personnalisé (sdl_display.c/h, data.c/h).
  - Réalisation des fonctions de sauvegarde et de chargement (save.c / save.h).

- Rédaction de la documentation (docoxigène) et mise en place des fichiers de tests unitaires.

Cette répartition nous a permis de combiner nos compétences pour couvrir l'ensemble des aspects techniques du projet, assurant ainsi une intégration cohérente et une gestion efficace du développement.

7. Fichiers benchmarkés

Présentation des fichiers benchmarks.

Pour évaluer les performances de notre réseau de neurones et la robustesse de notre implémentation, nous avons réalisé une série de benchmarks en utilisant le framework **ubench.h**. Ces tests mesurent notamment :

- La **propagation** et la **rétropropagation** sur différentes architectures (par exemple, réseaux 2→2, 2→3, 3→4).
- La **génération des données** (spirales d'Archimède et spirales personnalisées).

Un document PDF, intitulé **Benchmark.pdf**, présente une analyse comparative entre deux approches de benchmark il ce situe dans le dossier Bench :

- **Version 1 (UBENCH_F)** qui utilise la gestion des fixtures pour une initialisation et un nettoyage explicite du réseau avant et après chaque test, et
- **Version 2 (UBENCH_EX)** qui repose sur des tests autonomes plus simples.

Points clés tirés du document :

- L'approche avec fixture (V1) offre une structure modulaire et une meilleure gestion des ressources complexes, bien que la syntaxe soit plus verbeuse.
- L'approche simple (V2) est plus rapide à mettre en place et lisible, avec des performances globalement comparables à celles de la version avec fixture.
- Les résultats, exprimés en microsecondes, montrent des écarts négligeables entre les deux approches pour la propagation, la rétropropagation, et la génération des données.

Ces benchmarks confirment que notre implémentation est performante et que les choix architecturaux adoptés permettent une exécution efficace des algorithmes d'apprentissage. Pour plus de détails, veuillez-vous référer au document **Benchmark.pdf** Benchmark, qui fournit l'analyse comparative complète, incluant les captures d'écran et les résultats numériques détaillés.

8. Logs et fichiers testés

Présentation des logs et des fichiers de tests.

Pour faciliter le suivi du fonctionnement de notre système, nous avons intégré un mécanisme de journalisation (logging) centralisé, ainsi que plusieurs fichiers de tests. Ces outils permettent de tracer les événements clés (initialisation, propagation, rétropropagation, sauvegarde/chargement, etc.) et d'identifier rapidement les éventuelles erreurs ou anomalies dans l'exécution.

Les fichiers de logs et de tests ont été documentés de manière détaillée dans un fichier PDF ainsi que dans un document Word (logs.docx et exemple de rapport.docx). Ces documents expliquent :

- **La structure et le fonctionnement du système de log**
  Les fichiers *log.c* et *log.h* regroupent toutes les fonctions de journalisation, avec des niveaux variés (INFO, WARNING, ERROR) et des messages spécifiques pour chaque module (par exemple, génération des spirales, propagation du réseau, opérations de sauvegarde/chargement).

- **L'intégration des logs dans le flux d'exécution**
  Chaque action critique du projet (propagation, rétropropagation, sauvegarde, chargement, etc.) est loguée afin de garantir une traçabilité complète des opérations.

- **Les fichiers de tests et benchmarks**
  Des tests unitaires et des benchmarks ont été développés pour mesurer les performances du réseau de neurones et vérifier la robustesse du système. Ces résultats sont également présentés dans le PDF de benchmark, qui compare différentes approches et fournit des statistiques détaillées.

Ces documents fournissent une vision exhaustive de la qualité et de la performance de notre implémentation, et constituent un support précieux pour la maintenance et l'évolution future du projet.

III. Sources d'information

Recensement de tous les liens et références utilisés pour réaliser le projet.

Pour la réalisation de ce projet, nous avons consulté diverses sources qui nous ont permis de mieux comprendre et implémenter les concepts utilisés. Voici une liste non exhaustive des références et liens :

- **Documentation SDL2**
  La documentation officielle de SDL2 a été essentielle pour la conception de l'interface graphique et la gestion des fenêtres et du rendu.
  Lien : libsdl.org

- **Framework ubench.h**
  Le micro-framework ubench.h a été utilisé pour réaliser les benchmarks. La documentation et les exemples disponibles sur GitHub nous ont guidés pour structurer nos tests de performance.
  Lien : github.com/sheredom/ubench.h

- **Tutoriels et articles sur les réseaux de neurones en C**
  Plusieurs tutoriels et articles en ligne, notamment sur Developpez.com, nous ont aidés à comprendre et implémenter les algorithmes de propagation et de rétropropagation.
  Exemple : Développez.com – Réseaux de neurones

- **Historique Git et dépôt du projet**
  L'historique Git a permis de suivre l'évolution du code et de documenter les différentes itérations du projet.

  - Commit V1

- Commit V2
- Commit V3

Si vous souhaitez consulter l'ensemble des commits, veuillez consulter le dépôt complet : github.com/kaaix/reseau-neurones-spirales

- **Autres ressources techniques**
  Divers articles et tutoriels sur la programmation en C, la gestion de la mémoire et la conception d'algorithmes ont également été consultés pour renforcer notre approche.

Ces références, combinées aux retours d'expérience obtenus lors du développement, ont permis d'assurer la robustesse et la performance de notre implémentation.

IV. Conclusion

Ce projet a constitué une opportunité d'explorer en profondeur la conception, l'implémentation et l'optimisation d'un réseau de neurones en langage C, en intégrant à la fois des algorithmes d'apprentissage supervisé et une interface graphique interactive grâce à SDL2. Nous avons pu développer un système modulaire capable de traiter des données complexes, illustré ici par la classification de points répartis en spirales, et démontrer que même avec des outils de bas niveau, il est possible de réaliser des applications performantes en intelligence artificielle.

La démarche adoptée s'est articulée autour de plusieurs étapes clés :

- **Conception du réseau neuronal** : Nous avons d'abord établi une architecture multicouche flexible, permettant de configurer dynamiquement le nombre de couches cachées, ainsi que le nombre de neurones par couche. L'implémentation des fonctions de propagation et de rétropropagation a permis d'appréhender concrètement les mécanismes de l'apprentissage supervisé, tout en intégrant une initialisation aléatoire basée sur une graine issue de /dev/urandom pour améliorer la diversité des poids initiaux.
- **Adaptation aux données** : La génération des spirales à l'aide d'équations paramétriques classiques a servi de point de départ, avant de passer à la spirale

d'Archimède, qui a offert une meilleure répartition des points et, par conséquent, une frontière de décision plus nette. Les ajustements fins apportés aux paramètres de normalisation (scale) et d'espacement (b) ont été essentiels pour optimiser la qualité visuelle du dataset et la précision de la classification.

- **Interface graphique et interactions utilisateur** : L'utilisation de SDL2 a permis d'offrir une interface en temps réel, où l'utilisateur peut non seulement visualiser les résultats de l'apprentissage, mais aussi interagir avec le système via des interfaces de dessin personnalisé. La refonte de l'affichage graphique a abouti à deux approches complémentaires : l'une pour une classification binaire (avec la fonction colorier_ecran), et l'autre pour des scénarios multi-classes (via colorier_ecran_rgb), chacune adaptée aux besoins spécifiques du modèle testé.

- **Sauvegarde, chargement et benchmarks** : La mise en place d'un système de sauvegarde et de chargement, basé sur l'écriture et la lecture de fichiers binaires, a permis de pérenniser l'état du réseau neuronal, assurant ainsi une reprise d'entraînement ou une utilisation ultérieure sans perte des informations cruciales. Parallèlement, les benchmarks réalisés avec le framework ubench.h ont confirmé la performance et la robustesse de notre implémentation, en comparant différentes approches et configurations (par exemple, réseaux 2→2, 2→3, 3→4). Ces tests, documentés dans un PDF dédié, montrent des écarts de performance très faibles entre les diverses méthodes testées, attestant de la stabilité du système.

- **Suivi et documentation** : La gestion rigoureuse des versions via Git a été un atout majeur, nous permettant de suivre l'évolution du projet à travers plusieurs itérations (Commit V1, V2, V3) et d'apporter les ajustements nécessaires à chaque étape du développement. De plus, la documentation fournie, tant sous forme de fichiers PDF que de documents Word, a permis de détailler chaque aspect du projet, facilitant ainsi la compréhension et la maintenance du code.

En conclusion, ce projet a non seulement permis de développer un réseau de neurones opérationnel en C, capable de réaliser des classifications non linéaires sur des datasets complexes, mais il a également démontré l'importance d'une approche intégrée combinant performance algorithmique, interface utilisateur ergonomique et rigueur documentaire. Les résultats obtenus constituent une base solide pour des évolutions futures, notamment en ce qui concerne la sauvegarde du dessin personnalisé et l'extension du modèle à des cas d'utilisation plus variés. Ce travail ouvre également la voie à l'exploration de nouvelles architectures de réseaux et à l'optimisation des performances dans un contexte de

développement en C, tout en s'appuyant sur des outils de benchmarking et de journalisation efficaces pour garantir la qualité du code.

Projet encadré par M. Nicolas Courilleau
Réalisé par El Mahdi Benfdal et Ewen Croizier
Licence 3 Informatique – Génie Logiciel
Année universitaire 2024-2025