|  |
| --- |
|  |
| **Projet C #4** |
| Algorithmique et Programmation en C |
|  |
| *Meningix : Réseau de Neurones* |
| D:\server\projects\meningix\data\logo.png |
| *Par Loïc Belmonte et Rémi Jarasson* |
| *12/05/2010* |

|  |
| --- |
| Rapport de projet sur la création en langage C d’un programme de modélisation de fonctions (spirale par défaut) effectuée au travers d’un réseau de neurone |

**Table des Matières**

[Introduction 1](#_Toc261391623)

[Préparation du projet 2](#_Toc261391624)

[A. Le Principe du Réseau 2](#_Toc261391625)

[1. Caractéristiques d’un neurone 2](#_Toc261391626)

[2. Caractéristiques d’un réseau 3](#_Toc261391627)

[B. L’Apprentissage 4](#_Toc261391628)

[C. Organisation du Projet 5](#_Toc261391629)

[1. Répartition du Travail 5](#_Toc261391630)

[2. Outils de développement 5](#_Toc261391631)

[Le Réseau 6](#_Toc261391632)

[A. La modélisation 6](#_Toc261391633)

[1. Les données pour le calcul 6](#_Toc261391634)

[2. Les données pour l’apprentissage 7](#_Toc261391635)

[3. Les données de configuration/d’interface 8](#_Toc261391636)

[B. Implémentation 9](#_Toc261391637)

[1. Fonctions de création et de suppression 9](#_Toc261391638)

[2. Fonctions de création des exemples et de suppression 9](#_Toc261391639)

[3. Fonctions de sauvegarde et de chargement 9](#_Toc261391640)

[4. Fonction de calcul et de rétropropagation d’erreur 10](#_Toc261391641)

[5. Fonction d’apprentissage 11](#_Toc261391642)

[6. Fonctions auxiliaires 12](#_Toc261391643)

[L’Interface 13](#_Toc261391644)

[A. Le Système de l’interface 13](#_Toc261391645)

[B. Apprentissage 14](#_Toc261391646)

[1. Configuration 14](#_Toc261391647)

[2. Apprentissage 15](#_Toc261391648)

[C. Fonctionnalités Avancées 16](#_Toc261391649)

[1. Ecran du Résultat 16](#_Toc261391650)

[2. Forme Personnalisée 17](#_Toc261391651)

[Bilan du Projet 18](#_Toc261391652)

[A. Les Connaissances Acquises 18](#_Toc261391653)

[B. Le potentiel des réseaux de neurones 18](#_Toc261391654)

[Conclusion 19](#_Toc261391655)

[Le Site du Projet 19](#_Toc261391656)

Introduction

Pour ce dernier projet de l’année, le sujet propose de nous intéresser aux réseaux de neurones, ce qui nous permet de faire nos premiers pas dans le développement d’une intelligence artificielle, basée sur l’utilisation d’exemples prédéfinis plutôt que par la mise en place d’un système strictement logique/mathématique.

L’exploitation du réseau de neurones en question se fait pour le projet Meningix par la modélisation en 2 dimensions d’une fonction génératrice d’une double spirale d’Archimède bicolore, sans passer bien entendu par une expression mathématique, excepté pour obtenir des exemples. La modélisation de cette fonction passe d’abord par l’apprentissage des exemples au réseau, puis par la modélisation de sa réponse après apprentissage.

En d’autres mots, ce nouveau projet s’annonçait dès le début assez complexe, et nous demandait d’être rigoureux. Cependant, comme à notre habitude, nous ne nous sommes pas limités aux seuls termes du sujet, et avons tenté de développer un maximum de fonctionnalités.

Préparation du projet

La lecture intégrale du sujet nous a laissé tout d’abord perplexe. En effet, une seule lecture était insuffisante pour prendre conscience de l’ampleur du projet, et du travail, mais encore moins de l’utilité, dont nous parlerons en fin de rapport. Mais après quelques lectures, et une petite documentation sur Internet, nous avons pu avoir une meilleure idée de ce qui nous était demandé.

## Le Principe du Réseau

Le principe des réseaux de neurones s’inspire du fonctionnement biologique des neurones, bien qu’il soit exclusivement mathématique. En lui-même le principe est extrêmement simple, et nous le verrons plus tard, n’est pas tellement plus compliqué à mettre sous forme informatique.

Le but du réseau est qu’il sache différencier à quelle forme appartient un point donné, en renvoyant sa réponse sous forme de probabilité.

Coordonnées

**[x,y]**

Probabilités

**[a,b,…]**

**RÉSEAU**

### Caractéristiques d’un neurone

Un neurone artificiel possède exactement la même structure qu’un neurone biologique : il possède des synapses qui constituent les entrées d’un neurone, ainsi qu’un axone qui est la réponse que fournit un neurone aux données récoltées par ses synapses/entrées.

**E2**

**E1**

**E3**

**E4**

**SN**

**N**

Cependant, les neurones diffèrent totalement des neurones biologiques, puisqu’ils effectuent des calculs mathématiques, afin de générer une réponse en Sortie. Afin que chaque neurone ne génère pas tous la même réponse, on attribut des coefficients appelés poids pour chacune de ses synapses, afin que chaque entrée s’exprime différemment, et puisse permettre à chaque neurone de s’exprimer eux aussi différemment. Au départ, on attribut les poids aléatoirement, entre -0.5 et 0.5, valeurs qui peuvent se justifier de manière expérimentale.

Pour ce qui est du calcul, le sujet nous donne un exemple de formule utilisable pour notre cas. La sortie de chaque neurone s’exprime ainsi comme la tangente hyperbolique de la somme pondérée (par les coefficients) de chaque entrée.

avec S la sortie et P le poids des entrées.

D’après cette formule, la réponse d’un neurone se situe entre -1 et 1, que on pourrait l’interpréter basiquement comme la probabilité que le point soit situé sur une forme ou l’autre (-1 pourrait théoriquement être compris comme la probabilité qu’il n’y soit pas). Cependant, nous verrons que l’interprétation peut être un peu plus compliquée.

Reste à savoir maintenant comment fonctionnent ces neurones en réseau.

### Caractéristiques d’un réseau

Un réseau de neurones, comme son nom l’indique, est constitué de plusieurs neurones. Cependant, pour lui permettre de générer des fonctions assez complexes, celui-ci contient un nombre conséquent de neurones, organisés en couches.

Ce système de couches est facile à structurer étant donné celle d’un neurone. Comme pour le cerveau, les neurones sont interconnectés ainsi, il suffit de connecter toutes les sorties des neurones d’une couche aux entrées de chacun des neurones de la couche suivante, ainsi, les réponses des neurones d’une couche seraient modifiées par les neurones d’une couche suivante, et ainsi de suite, afin d’obtenir au final la réponse voulue.

*Remarque : Il existe de nombreuses structures possibles, mais celle-ci est l’une des plus basiques. On l’appelle perceptron multicouches.*

**X**

**Y**

Forme A

Forme B

*Remarque :* *On voit sur ce schéma deux « neurones » X et Y, qui sont en fait les entrées X et Y du réseau, et qui ne sont pas réellement des neurones, ils n’ont pas de synapses. Comme chaque réponse neuronal se situe entre -1 et 1, alors la position [x,y] doit s’y situer également, ou plutôt entre 0 et 1, -1 n’ayant pas de sens pour la position (dans notre cas).*

## L’Apprentissage

Le fonctionnement d’un réseau de neurones est à présent compris, cependant la question qui subsiste est : *comment faire en sorte que la réponse donnée soit celle que l’on attend, à savoir pour le sujet, la fonction double spirale ?*

La réponse réside simplement dans les coefficients synaptiques. En effet, les coefficients sont les seules valeurs sur lesquelles on peut influer. En modulant ces coefficients, on peut faire tendre le réseau vers la fonction voulue. Néanmoins, il faut effectuer le processus permettant de paramétrer ces synapses.

C’est ainsi que l’on va introduire l’**Apprentissage**. Ce processus est l’essentiel du projet, car en lui-même le réseau n’est qu’une simple fonction de calcul. Il prend toute son importance une fois l’Apprentissage effectué.

Le but de ce processus est de faire apprendre au réseau un certain nombre de points d’exemples sur lesquels celui-ci va pouvoir se baser pour répondre le plus judicieusement possible. Cela se fait par un algorithme très ingénieux : on se sert de l’erreur du réseau pour la rectifier au fur et à mesure. Cette rectification s’appelle la **rétropropagation du gradient**.

Synapses

Synapses

**X, Y**

**Aréponse**

**Bréponse**

**Aattendu**

**Battendu**

**ERREUR**

**RECTIFICATION**

Le Processus défini ci-dessus met en image la correction des coefficients pour une valeur, cependant il faut permettre au réseau de répondre à plusieurs valeurs. Ainsi, on ajoute un facteur **epsilon** afin que la correction puisse prendre peu à peu un grand nombre d’exemples. Il faudra alors plusieurs passages par ce cycle pour obtenir du réseau une réponse viable, jusqu’à un certain **seuil**, qui permettra d’arrêter le processus quand la réponse du réseau est à peu près correcte.

Nous avons fait quasiment le tour du principe du réseau de neurones, reste à organiser le travail pour commencer à implémenter celui-ci en informatique.

## Organisation du Projet

### Répartition du Travail

Le sujet indiquait dès le début que le travail devait s’effectuer en trinôme, cependant après nous être entretenu en binôme, nous avons jugé qu’une troisième personne serait inutile, car nous ne pourrions pas nous répartir les tâches de manière correcte.

Nous avons donc préféré adopter la répartition habituelle que nous avions adopté jusqu’à maintenant. Ainsi, l’un des binômes s’est consacré au développement du réseau et ses fonctions associées, et l’autre pouvait commencer à créer l’interface, chacun ayant toujours un œil sur le travail de l’autre.

### Outils de développement

Comme toujours au début d’un projet, nous nous consacrons à étudier les possibilités qui nous sont offertes, et quelles fonctionnalités nous pouvions implémenter. Pour ce qui est de la forme, nous avions prévu de faire une première version en console, utilisant les codes sources fournis, puis d’adapter ce code en graphique.

Comme le précédent projet **Kaaptur** utilisait la SDL, et bénéficiait d’une interface développée spécifiquement pour celui-ci, nous avons choisi de réutiliser la SDL pour ce projet, ainsi que le système d’interface, dont nous parlerons en seconde partie de ce rapport.

Enfin, à propos de nos règles de développement, notre « éthique », nous avons globalement choisi de travailler de la même manière que pour **Kaaptur**. Ainsi, la structure du projet y est très similaire.

*Petite note de bas de page :*

*Comme dit précédemment, une grosse partie du projet sera issue de copier-coller du précédent projet, auquel nous aurons apporté des modifications. Cependant, les commentaires de ces lignes risquent de ne plus être très clairs, et pourraient être incohérents. Cela est dû au rythme qui a accompagné ce projet, et qui nous a poussé à faire des compromis, et donc mettre la priorité sur le code, plutôt que les commentaires.*

Une fois toutes ces formalités abordées, nous avons donc pu commencer par développer Meningix, en commençant par développer le réseau et tout le système d’apprentissage, pour enfin implémenter le tout dans une interface graphique.

Le Réseau

## La modélisation

Nous avons modélisé le réseau de neurones par une structure. Contrairement à ce que le sujet conseillait de faire, nous n’avons pas modélisé chaque couche par une structure. En effet il était inutile de faire un tableau de cette structure pour représenter les différentes couches du réseau : il suffisait de rajouter une dimension aux différents tableaux (comme celui des poids) correspondant au numéro de la couche concernée. De plus cette modélisation permet d’économiser un peu de place en mémoire car certaines données auraient été dupliquées (le nombre d’entrées d’une couche est le nombre de sorties de la couche précédente, etc.). Notre structure se compose ainsi de 3 parties différentes.

### Les données pour le calcul

Les premières données que nous avons intégrées à notre structure sont les données indispensables à un réseau : toutes celles qui sont nécessaires pour faire le calcul des sorties correspondant aux entrées données (sans se soucier que ce calcul donne un résultat satisfaisant ou non).

#### Tableau donc taille utile

Il y a d’abord les entiers servant de tailles utiles : le nombre de couches (**nbr\_couches**) et un tableau (**nbr\_neurones**) qui contient le nombre de neurones pour une certaine couche (ex. : nbr\_neurones[2] est le nombre de neurones sur la 3ème couche, la numérotation commençant à 0).

Pour ces entiers, on ne tient pas compte de la différence entre couche d’entrée, couche(s) cachée(s) et couche de sortie : nous verrons plus tard que nous avons essayé de programmer sans se soucier des quelques différences de ces 3 types de couche pour obtenir un code plus clair en effectuant le même traitement lorsque cela était possible.

#### Le plus important : les synapses

Ensuite, les données les plus importantes : les poids représentés par des réels (**poids**). Il y a autant de poids que de synapses c’est-à-dire autant qu’il y a de liens entre les neurones. Les neurones n’étant liés qu’à ceux de la couche précédente (dans ce programme, car il existe des réseaux plus complexes), il était donc pertinent de faire une première dimension pour le tableau des poids qui correspond à la couche considérée.

Pour chaque couche, on va considérer les liens de chaque neurone de cette couche : il faut donc une deuxième dimension qui correspond aux neurones de la couche considérée.

Enfin, chacun de ces neurones a un lien avec un neurone de la couche précédente, ce qui constitue la troisième et dernière dimension du tableau. C’est cette dernière dimension qui justifie le choix de modéliser le réseau complet plutôt que couche par couche : on obtient directement les différentes tailles utiles (poids [*nbr\_couches*] [*nbr\_neurones[i]*] [*nbr\_neurones[i-1]*], en sachant que la couche d’entrée [i=0] n’a pas de poids).

#### Le résultat des calculs

Pour éviter de refaire toutes les allocations dynamiques à chaque calcul du réseau, nous avons rajouté un tableau (**resultat**) à 2 dimensions (couche puis neurone de la couche) qui est utilisé pour stocker tous les calculs intermédiaires. Pour celui-ci la troisième dimension était inutile car on ne considère pas les synapses mais les sorties des neurones et il n’y a qu’une sortie par neurone.

*Remarque : toutes ces données pour le calcul ne peuvent être changées par l’utilisateur entre les différents apprentissages sinon le réseau ne serait plus cohérent.*

### Les données pour l’apprentissage

Si on s’arrête à ce niveau de la modélisation, le réseau ne servira à rien : les poids étant initialisés aléatoirement, la réponse que l’on obtiendrait serait elle aussi aléatoire. Il faut donc rajouter à cette structure ce qui fait toute la puissance et l’intérêt des réseaux de neurones : les données pour pouvoir apprendre.

#### Les paramètres

Le premier paramètre important est la vitesse d’apprentissage **epsilon**. Ce facteur détermine la fraction de l’erreur commise que l’on va prendre en compte pour corriger les poids du réseau. Une valeur trop importante fait apprendre le réseau trop vite et donc oublier les exemples les plus anciens au profit de ceux présentés plus récemment. Une valeur trop faible ralentit le processus d’apprentissage : le réseau n’apprend pas ! Les valeurs qui semblent être un bon compromis se situent au tour de *0,001*.

Le deuxième paramètre important est le seuil de correction (**seuil**) qui détermine la correction des poids minimale à atteindre pour considérer que l’apprentissage est terminé. Plus ce seuil est grand et plus l’apprentissage se terminera vite ce qui peut donc entraîner un apprentissage bâclé. Plus ce seuil est petit et plus l’apprentissage sera long mais précis. Il faut cependant faire attention : avec un seuil trop bas, on diminue la capacité du réseau à généraliser. Les valeurs correctes sont autour de *0,1* (légèrement plus basse en coordonnées polaires).

*Remarque : ces deux paramètres sont les seuls (sans prendre en compte ceux de l’interface que nous verrons plus tard) qui peuvent être changés entre les apprentissages sans perdre la cohérence du réseau.*

Les paramètres suivants ne sont pas des champs de la structure mais des constantes : nous pensons que ces paramètres ne devraient pas être changées par l’utilisateur. Il y a LONGUEUR\_INTERVALE qui détermine l’intervalle dans lequel on génère les poids aléatoirement. Cette longueur vaut 1 : entre -0,5 et 0,5 (comme conseillé sur plusieurs documents dont la correction apportée au sujet). Il y a ensuite PAS\_SPIRALES qui est le pas pour la variable t de la génération des spirales. Ce pas est fixé à 0,2 par le sujet. Enfin, il y a NBR\_PTS\_BOUCLE qui est le nombre de points exemples que l’on présente au réseau avant d’évaluer si la correction moyenne est sous le seuil demandé. Nous l’avons fixé à 100.

#### L’ensemble des points exemples

Appelé trainingset dans les fichiers mis à disposition des élèves avec la correction du sujet, nous avons pensé qu’il serait plus simple de l’intégrer à la structure du réseau d’une part, parce qu’un réseau ne peut pas changer complètement d’ensemble d’exemples sans perdre la cohérence des poids et d’autre part, parce que cela permet de passer un seul argument aux fonctions ce qui simplifie l’écriture du code. Nous en avons aussi profité pour refaire nos propres fonctions pour traiter l’ensemble d’exemples et ainsi corriger certaines erreurs du code original (plantage dû à un dépassement mémoire lorsque l’on définissait la constante du nombre de points à un nombre impair).

Nous avons évidemment repris l’utilisation de 2 tableaux (**pts** et **att** ; le premier pour les coordonnées des points et le deuxième pour le sorties correspondantes) à 2 dimensions (la première dimension est le nombre de points exemples et la deuxième est le numéro de la sortie). Avec l’utilisation des tableaux, il nous faut leur taille utile : **nbr\_pts** pour la première dimension. Pour la deuxième dimension, on déduit cette taille avec le nombre de neurones : *nbr\_neurones[0]* pour **pts** car les coordonnées sont en fait des entrées et *nbr\_neurones[nbr\_couches - 1]* pour **att** car les sorties attendues correspondent aux sorties du réseau de neurones.

Si nous n’avions pas rajouté plusieurs fonctionnalités, cela aurait été suffisant. Mais nous avons rajouté la possibilité de changer de système de coordonnées et on ne peut pas demander à un réseau qui a appris en cartésien de donner les mêmes résultats en polaire (perte de cohérence). Il faut donc ajouter le système de coordonnés utilisé : **coord** (de type énuméré, cf. commentaires du code).

De plus, on peut soit tracer les spirales soit une forme quelconque. Cela ne change rien dans le traitement du réseau de neurones mais pour la sauvegarde il faut être capable de restaurer la configuration d’origine : on sauvegarde donc le mode dans **mode** (de type énuméré, cf. commentaires du code).

Enfin dans le mode « spirales », nous avons ajouté quelques paramètres qu’il faut aussi ajouter à la structure : le décalage (**offset**, nombre de points qu’on n’utilise pas car trop près de l’origine) et la **phase** qui fait « tourner » les spirales.

Nous avons aussi ajouté un autre tableau (**delta**) à 2 dimensions pour les mêmes raisons que *resultat* (*cf. c. les résultats des calculs*). Ce tableau contient les facteurs de corrections des poids pour chaque neurone.

### Les données de configuration/d’interface

Nous avons aussi rajouté des données inutiles au fonctionnement du réseau mais qui servent à la présentation des résultats et que nous voulions sauvegarder pour être capable de charger exactement une configuration précédente.

Parmi ces données, il y a les couleurs des courbes (**forme**) au format RGB de la SDL. Dans le sujet nous avions une courbe bleue et une rouge mais nous voulions pouvoir mettre d’autres couleurs comme l’orange et le gris qui se fondent mieux dans notre interface.

Nous avons aussi rajouté le temps d’apprentissage (**temps**), pour avoir une indication bien qu’elle ne soit pas absolument juste puisque le temps d’affichage des aperçus est aussi compté avec le temps réel de l’apprentissage, qui lui-même dépend de la puissance de la machine.

## Implémentation

Maintenant que nous avons une structure qui modélise notre réseau de neurones, il va falloir écrire les fonctions qui permettent de s’en servir.

### Fonctions de création et de suppression

Comme notre structure utilise plusieurs tableaux dynamiques, nous avons implémenté des fonctions qui se chargent de faire les allocations et les libérations nécessaires. Cela permet de créer ou supprimer un réseau proprement et en une seule ligne n’importe où dans le code.

La fonction de création *nouveau\_reseau* permet de faire toutes les allocations nécessaires aux calculs et réalise aussi l’initialisation de certains champs (comme les poids qui sont initialisés aléatoirement dans l’intervalle souhaité).

La fonction « réciproque » de suppression *supprimer\_reseau* se charge de toutes les libérations correspondantes aux allocations de la fonction précédente et elle appelle aussi la fonction de suppression des exemples (dont nous parlerons après) pour libérer si nécessaire les exemples et ainsi effacer un réseau en une seule ligne de code sans se soucier du fait qu’on ait ou non alloués des exemples.

### Fonctions de création des exemples et de suppression

De même, nous avons des fonctions pour créer un ensemble d’exemples dans le cas des spirales et pour la suppression dans tous les cas.

Pour la création *generer\_spirales*, on alloue d’abord la place mémoire nécessaire et on met à jour les champs concernant les exemples (comme le mode, le système de coordonnées, etc.). On génère ensuite les spirales en coordonnées cartésiennes en alternant les courbes en faisant un test sur la parité du compteur. Cette façon de procéder évite le bug du nombre impair de points (dont nous avons parlé précédemment). On convertit si nécessaire les coordonnées dans un autre système de coordonnées (polaires par exemple) à l’aide d’une fonction de conversion que l’on détaillera plus tard. Enfin, on appelle la fonction de normalisation pour rendre ces données utilisables par un réseau de neurones (nous parlerons de cette fonction plus tard aussi).

La fonction de suppression *supprimer\_exemples* est très simple : on se contente de supprimer chaque allocation faite par la fonction précédente (ce qui fonctionne même dans les autres cas que les spirales). Cette fonction permet seulement de mieux structurer le code (et gagner en simplicité pour les modifications ultérieurs).

### Fonctions de sauvegarde et de chargement

Pour la sauvegarde et le chargement, nous voulions améliorer le système précédent (*cf.* ***Kaaptur***) de complétion des noms. Pour ***Kaaptur***, nous ajoutions le dossier de destination des fichiers, et pour ce projet, nous ajoutons également l’extension à la fin du nom : cela rend plus intuitif la saisie du nom de fichier.

Les deux fonctions *sauvegarder\_reseau* et *charger\_reseau* ont un fonctionnement presque parallèle : lorsque la première écrit, l’autre lit. Nous ne détaillerons donc qu’une seule des deux fonctions, à savoir le chargement car légèrement plus complexe.

On charge, pour commencer, les 4 premiers octets qui doivent former la chaîne « MNGX » pour confirmer que le fichier ouvert a bien été généré avec notre programme. Dans le cas où le fichier est valide, on vérifie qu’un réseau n’occupe pas déjà la place mémoire : si c’est le cas on libère avant de continuer. On lit successivement les différents champs indispensables pour connaître la place nécessaire pour charger le réseau. On peut alors faire les allocations nécessaires (en appelant la fonction de création d’un réseau). On écrit alors tous les poids en mémoire. On passe ensuite à l’ensemble d’exemple : on lit les paramètres correspondant. Dans le cas des spirales, cela est suffisant pour générer de nouveau les points. Dans le cas de formes personnalisées, on alloue la mémoire nécessaire et on lit chaque point. Enfin, on lit quelques paramètres nécessaires à l’interface comme le temps d’apprentissage déjà effectué.

### Fonction de calcul et de rétropropagation d’erreur

Nous arrivons maintenant à la partie la plus importante de ce projet et qui tient pourtant sur seulement une soixantaine de lignes de code. Cette partie a été codée 2 fois à cause des erreurs contenues dans la première version du sujet. Nous avions pourtant pensé à la normalisation des entrées mais cela ne marchait pas. Nous avons ensuite tenté de remplacer la dérivée de la sigmoïde par celle de la tangente hyperbolique dans la formule de rétropropagation et, même si les résultats furent meilleurs, le réseau n’était pas stable. C’est à ce moment que nous avons décidé de recommencer. Il fallait en fait associer ces deux corrections pour obtenir un réseau fonctionnel. Bien que frustrant, ce premier échec nous a permis de mesurer toute l’importance de la normalisation et de comprendre que la formule de rétropropagation n’a pas été choisie arbitrairement mais calculée par d’excellents mathématiciens (que nous ne sommes pas…).

Ces deux aspects, calcul et correction du réseau de neurones, peuvent sembler indépendants à priori mais en fait la rétropropagation du gradient d’erreur ne peut s’effectuer sans un calcul préalable des sorties du réseau. C’est pourquoi nous avons implémenté le calcul et la rétropropagation dans une seule et même fonction. Cette fonction peut être utilisée pour le simple calcul : il suffit de lui donner un pointeur NULL à la place des sorties attendues.

#### Calcul des sorties

La fonction commence par deux boucles POUR imbriquées : la première pour parcourir les couches et la seconde pour parcourir les neurones de cette couches. On calcule ensuite la somme pondérée des entrées comme nous calculions les sommes lors du premier projet de cette année (c’est-à-dire une autre boucle POUR) ! Enfin on effectue le calcul de la fonction de sortie qui, dans notre programme, est la fonction tangente hyperbolique (on utilise la fonction de la librairie *math*). Dans le cadre d’un simple calcul, la fonction se termine ici.

#### Correction des poids

Si le réseau est encore en apprentissage alors, on a des valeurs attendues qui ont été passées en arguments. On va alors faire « remonter » (partir de la sortie pour revenir jusqu’à l’entrée) l’erreur commise en cherchant le(s) neurone(s) responsable(s) ! On part donc de la couche de sortie pour laquelle le calcul est particulier. Ensuite on parcourt le réseau en sens inverse à l’aide d’une boucle POUR dont le compteur est décrémenté. Puisqu’il faut traiter chaque neurone, chaque couche rentre dans une autre boucle POUR qui, elle, parcourt les neurones (dans le bon sens puisqu’ici cela n’a pas d’importance). Pour chaque neurone, le calcul est la somme pondérée de l’erreur (utilisation de la dérivée) que sa sortie implique sur les neurones de la couche suivante. C’est de cette façon que l’on désigne un neurone responsable !

Il va maintenant falloir appliquer une sanction : la correction des poids ! On va donc explorer l’ensemble des synapses (3 boucles POUR imbriquées) pour les corriger en fonction de la responsabilité des neurones concernés et du degré de sévérité (i.e. facteur d’apprentissage).

*Remarque : On comprend donc la complexité de l’algorithme qui demande 3 fois 3 boucles POUR imbriquées pour 1 unique exemple et la correction correspondante (sans compter les calculs de tangente hyperbolique, etc.).*

### Fonction d’apprentissage

Maintenant que nous avons tous les outils pour faire un réseau de neurones, il va falloir organiser l’apprentissage. De plus, on sait que l’apprentissage va être long donc on ne va pas pouvoir faire uniquement les calculs : il va falloir faire une mise à jour de l’interface régulièrement pour que l’utilisateur sache que le programme n’est pas planté. On fait alors une boucle qui va continuer tant que l’apprentissage n’est pas terminé et que l’utilisateur ne souhaite pas arrêter le processus.

#### Les calculs

On arrête l’apprentissage lorsque la correction devient « petite » (sous le seuil demandé). Or la suite des corrections n’est pas strictement décroissante et peut se révéler plus ou moins aléatoire. Nous avons pensé au conseil que nous a donné *M. Flasque* en cours, à savoir évaluer ce test sur plusieurs exemples (nombre à trouver) de suite et n’arrêter que si c’est bon pour tous. Mais nous pensons avoir trouvé une solution encore meilleure : faire la moyenne des erreurs et tester si cette moyenne est sous le seuil. Cette méthode est plus sûre (car si cela peut-être aléatoire pour un exemple, ça le reste pour plusieurs même si la probabilité diminue) et plus facile à implémenter. Nous faisons donc une boucle sur un certain nombre d’exemples (100 par défaut) avant de passer à la suite.

#### L’interface

Pour le code de base, nous avons utilisé les sources fournies aux élèves, afin de pouvoir visualiser les résultats de nos apprentissages. Pour une version console, ces aides nous suffisaient pour développer notre système de base.

Afin de voir en quasi temps réel l’avancement de l’apprentissage, nous avons eu l’idée de faire un rafraichissement du résultat à intervalle régulier, car les deltas ne nous donnaient pas des informations suffisamment précises, tandis que l’aspect visuel nous était plus utile.

### Fonctions auxiliaires

Nous avons codés des fonctions supplémentaires que nous avons utilisées dans les algorithmes précédents. C’est fonction étant utilisées plusieurs fois, il était nécessaire de les coder en tant que fonction et pas directement dans les autres algorithmes.

#### Affichage du rendu

Pour afficher le rendu d’un réseau, on balaye toute une surface en demandant au réseau les sorties pour chaque coordonnées (qui doivent être converties dans le système du réseau et normalisées). On fait ensuite une moyenne pondérée des couleurs associées aux sorties que l’on affiche. Cela se fait évidement pour obtenir une couleur RGB, utilisable seulement en graphique.

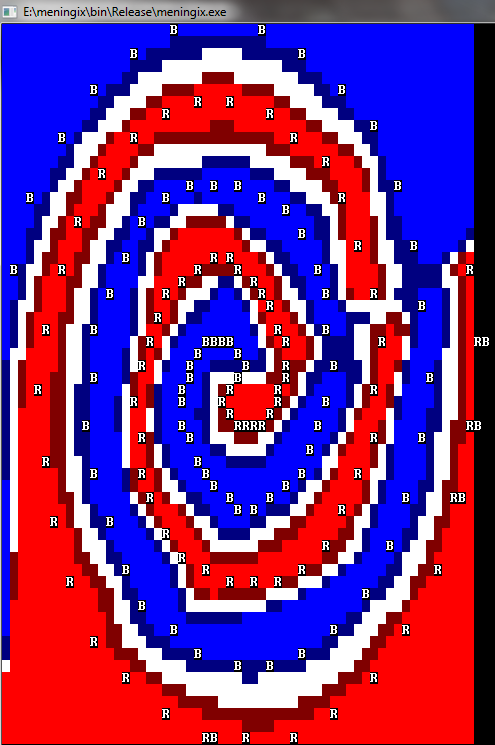
Pour pouvoir visualiser ce que le réseau a essayé d’apprendre, on affiche ensuite chaque point de l’ensemble d’exemples en blanc par-dessus les couleurs précédentes.

#### Conversion cartésiennes vers polaires

Cette fonction fait la simple conversion mathématique d’un système de coordonnées vers un autre. Pour simplifier la normalisation, on contraint l’argument à être dans l’intervalle [0 ; 2∏].

#### Normalisation

Pour rendre les coordonnées utilisables par le réseau de neurones, il faut les normaliser. On effectue donc une normalisation simple (entre 0 et 1) des entrées. On traite tout de même particulièrement certains cas : on conserve l’orthonormalité dans le cas des coordonnées cartésiennes et on divise l’argument par 2∏ (sans se soucier de couvrir entièrement l’intervalle [0 ; 1]) dans le cas des coordonnées polaires. Ces cas particuliers permettent une meilleure visualisation du résultat.



Capture d’écran de Meningix en console

L’Interface

À ce moment du projet, l’essentiel des fonctionnalités requises étaient codées, et affichables en console comme il l’était prévu par le sujet et les fichiers mis à disposition des élèves. Cependant, nous nous somme vite rendu compte qu’il était extrêmement difficile de travailler en console, notamment pour permettre la saisie des paramètres du réseau et de l’apprentissage, d’autant plus que la complexité du sujet nous demandait de fournir un minimum d’explications sur l’utilisation du programme. C’est alors que l’on s’est donné pour but d’implémenter le réseau et toutes ses fonctions dans une interface graphique travaillée, et agréable.



## Le Système de l’interface

Pour créer l’interface de Meningix, nous avons décidé de réutiliser le travail effectué pour le jeu ***Kaaptur***, nous avons donc repris notre code pour construire l’interface grâce à la SDL. Bien que celle-ci ne soit pas très efficace en terme de performances, le coté statique du programme nous posait moins de problèmes. De toute manière, nous n’avions pas le temps d’apprendre à utiliser une autre librairie graphique.

Globalement, le système de l’interface est le même, cependant, nous avons amélioré quelque peu ses fonctionnalités, et complété les données de l’interface pour qu’elles répondent parfaitement aux besoins du programme.

* Ajout du groupe

L’interface de Meningix nécessitait plusieurs menus, afin de pouvoir d’une part charger un réseau, et d’autre part lancer un apprentissage. Cependant, cela nécessitait de rajouter une donnée dans la structure d’un élément.

Le groupe est stocké dans un type énuméré associé à chaque groupe. A cela s’ajoute des fonctions permettant de masquer et afficher chaque groupe, et éventuellement masquer tous les autres si besoin est.

* Ajout du style

Pour des raisons esthétiques, il était nécessaire d’ajouter également un flag binaire (dont les valeurs peuvent être combinées), afin de rendre certains textes plus gros, ou coloré…

Là encore, on a rajouté un type énuméré, configuré de cette manière :

* STYLE\_GROS = 1 : Affiche le texte avec une police de caractère grossie
* STYLE\_NOIR = 2 : Rend le texte Noir
* STYLE\_ORANGE = 4 : Rend le texte Orange/Coloré
* STYLE\_CENTRE = 8 : Centre l’élément sur la position donnée
* Ajout d’une info-bulle

Afin de nous permettre de donner des informations explicatives sur le fonctionnement du réseau, nous avons ajouté également une chaine de caractère pour chaque élément, permettant de stocker la description de son utilité.

La création d’une info-bulle flottante étant peu aisée, et ferait trop contraste avec le reste de l’interface, nous avons jugé utile de réutiliser l’idée de barre de message que nous avions eu lors de notre projet n°2, ***ReversiX***. Cette barre se situe en bas de la fenêtre, et permet d’afficher les messages d’erreur (colorés), mais surtout les info-bulles lors de leur survol.

* Optimisations

La réutilisation de l’interface a également conduit à l’optimisation de nombreux éléments.

La première amélioration concernait le chargement des images. En effet, celui-ci s’effectuait à chaque initialisation d’un élément qui utilise une image, or plusieurs éléments peuvent utiliser la même image (par exemple les boutons). On a donc rajouté une liste de textures, contenant toutes les images nécessaires, et dont les pointeurs peuvent être aisément récupérés pour chaque élément. On obtient donc un gain en mémoire.

La seconde optimisation concerne là encore les textures. En effet, les gros ralentissements étaient dus à la différence entre les formats des images blitées et celui de la fenêtre. Cette différence entre les 2 formats conduit à un temps de calcul pour adapter chacune image lors de chaque blit. Une fonction très simple permet de convertir l’image dans le bon format, une seule fois au début du programme, pour accélérer tous les affichages par la suite. On a donc effectué cette conversion dans tous les cas où cela était possible. On obtient alors un gain important en vitesse pour l’affichage (4 fois plus rapide !).

## Apprentissage

Le but du projet est de lancer un apprentissage, et de voir le résultat de celui-ci. Il faut donc que l’interface permette de fournir les outils pour configurer, lancer et afficher le résultat d’un apprentissage.

### Configuration

Pour configurer un réseau, un simple formulaire permet de passer des données en argument aux fonctions développées. Cela se fait facilement en créant les éléments de saisie et les boutons nécessaires.

Comme pour ***Kaaptur***, on a crée des fonctions associées à des Flags, et qui permettent de générer des fonctions de l’interface, comme afficher un menu/écran, changer d’écran, ou lancer un apprentissage.

Chacun des champs du formulaire est accompagné par une description permettant d’indiquer à l’utilisateur comment profiter du programme. Lorsque l’on valide les champs, et qu’on tente de lancer un apprentissage, une fonction *configurer\_apprentissage()* va vérifier s’ils sont corrects, et enfin initialiser le réseau et lancer l’apprentissage. Si les champs sont incorrects, un message d’erreur sera affiché, et le réseau ne s’initialisera pas.

Ce formulaire possède également un champ de saisie spécifique (Nom du Réseau), ainsi que le bouton Charger qu’il lui est associé. Ce bouton permet de charger un réseau précédemment sauvegardé. Voir plus loin les fonctionnalités additionnelles.



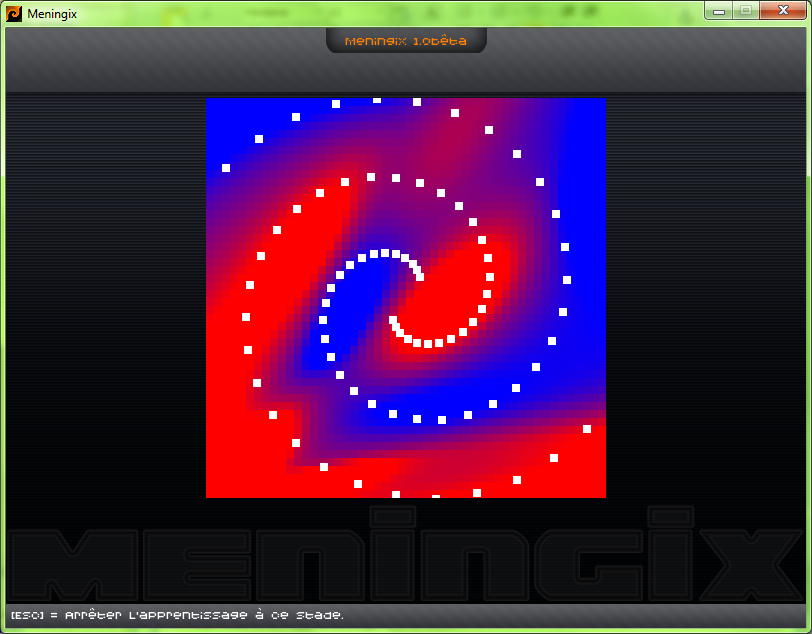
Ecran de Création de Réseau

### Apprentissage

Comme nous l’avions fait en console, nous avons permis à l’utilisateur de constater l’avancement sur l’apprentissage. On a donc un élément, permettant d’afficher la surface d’affichage du résultat/aperçu. Cet aperçu a une définition moindre (par défaut 1 point sur 8 pixels contre 1 pour 1 pour le rendu final), pour pouvoir le calculer fréquemment sans perdre trop de temps. Par défaut, on fait cette actualisation toutes les demi-secondes.

Enfin, on s’occupe des évènements extérieurs (par exemple le clavier : [ESC] pour arrêter) et ainsi l’utilisateur peut toujours interagir avec le programme pendant l’apprentissage. On fait alors seulement le test pour finir la boucle, et passer à un affichage final du résultat.

Pendant cet aperçu, le cycle d’apprentissage se déroulera tout comme dans la version console.



Ecran d’aperçu de l’Apprentissage

## Fonctionnalités Avancées

Pour dépasser les termes du sujet, nous avons pensé un maximum de fonctionnalités, afin de nous permettre de tester au maximum les possibilités des réseaux de neurones.

### Ecran du Résultat

Le résultat final s’affiche dans un écran spécifique, constitué d’un aperçu, de champs de saisie permettant de modifier en temps réel les valeurs du réseau (epsilon et seuil), ainsi que de statistiques sur celui-ci et l’apprentissage qu’il a subit :

* Configuration des couches
* Paramètres de l’Apprentissage
* Paramètres de la Forme
* Temps d’Apprentissage

Cet écran permet notamment de retracer un réseau chargé, mais également de continuer un apprentissage. C’est également via cet écran que l’utilisateur peut sauvegarder un réseau crée, en entrant le nom du réseau, et en appuyant sur Sauvegarder.

* La Sauvegarde

La sauvegarde était demandée par le sujet, cependant, elle n’était pas extrêmement précise. Nous avons choisi de pouvoir retenir l’intégralité du réseau, et pas seulement les poids. Cela nous permettait de réutiliser un fichier de sauvegarde également pour pré-remplir les formulaires.

On a donc également crée toutes les fonctions d’interface pour utiliser les sauvegardes, et les charger. Ces sauvegardes portent l’extension « .mgx » et se trouvent dans le répertoire neu/.



Ecran récapitulatif d’aperçu du Réseau crée

### Forme Personnalisée

L’une de nos principales fonctionnalités ajoutées était la possibilité de dessiner soi-même les formes à apprendre au réseau. Pour cela, nous avons crée un écran spécifique permettant de placer des points sur une surface de dessin (de la dimension de celle du résultat), en récupérant à intervalle de temps régulier la position de la souris.

On a également ajouté des éléments d’interface pour sélectionner la forme à dessiner. Deux boutons permettent d’effacer les points ajoutés, ou bien d’annuler le dernier placement

* Structure du dessin

Le dessin est stocké dans la structure de l’interface, par un tableau de structures *t\_point*, qui retient la position X et Y du point, ainsi que l’indice de la forme à laquelle le point appartient. Ce tableau est aisément manipulable grâce à la fonction realloc(). On peut agrandir ou rétrécir le tableau, ce qui nous a permis de créer les fonctions effacer et annuler.

Au final, ce projet Meningix nous permet de faire apprendre à peu près n’importe quelle forme en 2D à un réseau de neurones.

Néanmoins, ce n’est qu’une petite partie du potentiel des réseaux de neurones, et c’est en faisant le *Bilan du Projet* que nous pourrons voir en quoi ce projet nous permet de d’améliorer énormément nos techniques de programmation.

Bilan du Projet

Ce dernier projet de notre première année à l’EFREI n’a pas été de tout repos, étant donné la complexité relative et l’interface travaillé que nous avons développé. Cependant, nous avions bien deviné que l’idéal de ce projet était de travailler en binôme, car un groupe plus grand aurait fracturé la cohésion de groupe, et n’aurait pas été nécessaire.

À présent, nous pouvons faire le bilan de nos acquis et analyser le potentiel des réseaux de neurones.

## Les Connaissances Acquises

Ce projet avait l’avantage d’être complexe, et nous a permis d’étudier une notion avancée en programmation : les réseaux de neurones artificiels. On a donc acquis énormément par ce projet.

Pour ce qui est du travail en groupe, nous avons préféré continuer en binôme en répartissant le travail comme l’était ce rapport (principalement réseau/interface). Ce choix s’est fait après lecture et compréhension du sujet : nous ne ressentions pas la nécessité d’une troisième personne pour achever ce projet (contrairement au projet précédent, ***Kaaptur***). La suite nous a donné raison : en 4 heures de travail (sans compter la lecture et la compréhension préalables du sujet) et environ 500 lignes de code, le programme était parfaitement fonctionnel en mode console.

## Le potentiel des réseaux de neurones

Les réseaux de neurones permettent pour ce projet de modéliser des fonctions, spirales notamment, qui sont considérées comme inconnues. En étendant son application, on pourrait utiliser ce type de neurones pour approcher des fonctions plus compliquées, telles que l’établissement de modèles physiques complexes.

Mais en approfondissant beaucoup plus les applications, on peut vite s’apercevoir que les réseaux de neurones artificiels peuvent permettre de nombreuses applications, et même très farfelues. En fait, dès que l’on a besoin d’une réponse approximative, en se basant sur un nombre limité d’exemples, un réseau de neurone peut être une solution envisageable:

* On peut les utiliser pour recréer une intelligence artificielle comportementale : En faisant apprendre au réseau des réponses à des situations prédéfinies, on peut réussir à le faire prendre des « décisions » pour des situations inédites grâce à leur réponse.
* En prenant comme entrée des pixels, et en sortie des caractères, nous pouvons également créer un système de reconnaissance de caractère (OCR) qui donne une probabilité que les pixels forment telle ou telle lettre.
* On peut faire des conjectures sur des évolutions, basées sur des données financières par exemple, ou pour n’importe quel autre domaine.

Ainsi, l’application des réseaux de neurones n’est limitée que par notre imagination, et peut résoudre un tas de situation (mais pas forcément de la meilleure façon).

Conclusion

Dernier projet de l’année, et nouveau pas en avant dans notre cursus d’ingénieur en informatique. Meningix nous a permis de nous familiariser avec une notion importante de l’intelligence artificielle, qui repose sur l’utilisation d’exemples concrets et sur la mise en place d’un système comportemental qui se rapproche de la réalité. Contrairement à une étude purement logique ou mathématique d’un problème, les réseaux de neurones permettent de résoudre approximativement, mais facilement, des problèmes complexes notamment ceux concernant l’intelligence, et qui prendront probablement une grande place dans les projets futurs, personnels ou scolaires.

Mais ce projet est également crucial, car il clôt notre première année à l’EFREI, et doit nous permettre d’enchainer avec de nouveaux projets bien plus complexes. Ainsi, il est une fenêtre ouverte sur ce que nous réserve nos prochaines années, mais nous permet grâce à de nouvelles connaissances, de mieux appréhender certains problèmes, et même faire émerger de nouvelles idées en nous.

Bien qu’au départ son utilité semblait moindre, l’aboutissement du développement nous a conduit à mieux analyser l’outil que nous avions manipulé, et à présent, nous voyons clairement toute l’importance de celui-ci. Et nous en avons parlé en binôme, ce projet nous mènera à coup sûr, à nous ré-intéresser aux réseaux de neurones, peut-être en développant cet été un programme de reconnaissance de caractère, ou de formes, ou bien un site internet permettant de trouver des affinités, ou donnant des conseils via une analyse comportementale des visiteurs, ou tout simplement en rajoutant une nouvelle dimension à Meningix pour pouvoir modéliser des systèmes 3D, ou plus…

Ce qui est sûr, c’est que comme toujours, ce projet en annonce d’autres, qui seront toujours plus intéressants les uns que les autres, et que nous développerons avec autant de plaisir.

*Loïc Belmonte et Rémi Jarasson*

Le Site du Projet

Afin de faire partager notre travail, nous avons publié le programme et son code source sur un site internet, ouvert à tous. Les éventuelles mises à jour seront donc publiées à cette adresse, aussi longtemps que possible :

***http://projects.artistix.fr/meningix/***

*Remarque : d’anciens ainsi que d’éventuels futurs projets seront également disponibles sur ce domaine. Le projet Kaaptur notamment s’y trouvera, dont le rapport est joint avec les sources.*