Rapport Projet ZZBrain

ISIMA

Imad ENNEIYMY
enneiymy.i@gmail.com

Yassir KARROUM
ukarroum17@gmail.com

7 juin 2017

Encadrés par :
Violaine ANTOINE
violaine.ANTOINE@uca.fr

Table des matières

1 Introduction

Le présent document vise à décrire le projet ZZBrain, ce projet fut réalisé dans le cadre du module *Projet de deuxième semestre de la première année* à l'ISIMA. Il vise à créer une bibliothèque basique permettant la création et l'entrainement d'une classification basée sur les réseaux de neurones, c'est un projet purement pédagogique, et le lecteur intéressé par une bibliothéque pareille trouvera des alternatives bien plus puissantes telles que : *TensorFlow*[1] ou *DLib*[2].

Les notations, algorithmes et formules utilisés sont fortement basés sur le cours *Machine Learning* de *Andrew Ng*[2] même si ce dernier est principalement proposé en *Octave*[3] alors que ZZBrain est codé en C++[4].

Nous utilisons la biblothéque *Dlib*[1] pour la minimisation de la fonction $J(\Theta)$.

Dans ce qui suit nous présenterons de façon générale l'idée derrière les réseaux de neurones, nous présenterons également des conventions et notations qui seront utilisées un peu plus bas our expliquer les algorithmes et les structures de données utilisés. Nous proposerons également des pistes pour d'éventuelles améliorations de ce projet.

Une multitude d'exemples (implémentés en utilisant ZZBrain), seront également présentés et expliqués un peu plus bas.

A noter que l'intégralité du code (y compris ce rapport et son code source Latex) est disponible en open-source sur GitHub : https ://github.com/ukarroum/ZZBrain. Nous n'avons inlus dans le rapport que les parties que nous avons jugé pertinents.

2 Réseaux de neurones

2.1 Introduction

Les réseaux des neurones représentent une des méthodes les plus utilisées en *Machine Learning*, ils sont utilisés dans de nombreux domaines comme la vision par ordinateur entre autres. Leur popularité vient du fait qu'ils permettent de modéliser des problèmes très complexes. Récemment le champion du monde au jeu de Go vient de subir une défaite le 25 mai par un réseau de neurones[4], dans le jeu que les informaticiens considèrent comme étant l'un des jeux les plus durs à automatiser puisque le nombre de possibilités est énorme. Cette méthode nécessite néanmoins une puissance de calcul et une mémoire assez conséquente, dès que le nombre des noeuds commencent à croitre.

1. https ://www.tensorflow.org/
2. http ://dlib.net/
3. Alternative libre à *Matlab*
4. AlphaGo développé par Google

2.2 Structure du réseau

Un réseau de neurones peut etre representé comme suit :

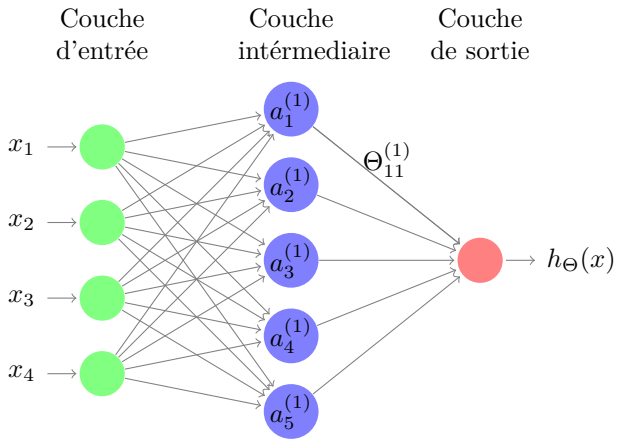

FIGURE 1 – Exemple d'un reseau de neurones

La couche d'entrée est ce qui est fourni au programme. Dans le cas d'une vision par ordinateur cela pourrait être les pixels de l'image, dans le cas d'un filtre à spam, cela pourrait être des booléans représentant l'existence de certains mots-clés.

Les couches intermédiaires permettant de complexifier le réseau et de lui permettre de modéliser des problèmes de plus en plus complexes, le souci est qu'elles augmentent énormément le temps de calcul.

La couche de sortie représente le résultat retourné par le réseau qui est un vecteur de booléans (souvent il est de taille 1) qui nous informe si oui ou non le vecteur d'entrée appartient à une classe donnée (par exemple si une image fournie en entrée est une voiture, ou si un message est un spam).

Le vecteur $a^{(1)} = \begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \\ a_5^{(1)} \end{pmatrix}$ permets de calculer la sortie $h_\Theta(x)$, en effet on pourrait considérer le vecteur $a^{(1)}$ comme la nouvelle entrée est ainsi de suite. Ce procédé s'appelle la *Forward Propagation*.

Les poids sont stockés dans une matrice tridimensionnelle notée Θ, chaque poids est dénoté par : $\Theta_{ij}^{(k)}$ où k est le numéro de la couche, i le numéro du noeud de la couche 2 et j le numéro du noeud de la couche 1.

On peut alors calculer $h_\Theta(x)$ comme suit :

$$
\begin{aligned}
z_1^{(1)} &= \Theta_{11}^{(0)} x_1 + \Theta_{12}^{(0)} x_2 + \Theta_{13}^{(0)} x_3 + \Theta_{14}^{(0)} x_4 \\
a_1^{(1)} &= g(z_1^{(1)}) \\
z_2^{(1)} &= \Theta_{21}^{(0)} x_1 + \Theta_{22}^{(0)} x_2 + \Theta_{23}^{(0)} x_3 + \Theta_{24}^{(0)} x_4 \\
a_2^{(1)} &= g(z_2^{(1)}) \\
z_3^{(1)} &= \Theta_{31}^{(0)} x_1 + \Theta_{32}^{(0)} x_2 + \Theta_{33}^{(0)} x_3 + \Theta_{34}^{(0)} x_4 \\
a_3^{(1)} &= g(z_3^{(1)}) \\
z_4^{(1)} &= \Theta_{41}^{(0)} x_1 + \Theta_{42}^{(0)} x_2 + \Theta_{43}^{(0)} x_3 + \Theta_{44}^{(0)} x_4 \\
a_4^{(1)} &= g(z_4^{(1)}) \\
z_5^{(1)} &= \Theta_{51}^{(0)} x_1 + \Theta_{52}^{(0)} x_2 + \Theta_{53}^{(0)} x_3 + \Theta_{54}^{(0)} x_4 \\
a_5^{(1)} &= g(z_5^{(1)}) \\
h_\Theta(x) &= g(\Theta_{11}^{(1)} a_1 + \Theta_{12}^{(1)} a_2 + \Theta_{13}^{(1)} a_3 + \Theta_{14}^{(1)} a_4)
\end{aligned}
$$

Ce procédé est communement appelé : *Forward Propagation*.

Ici la fonction $g(x)$ est ce que l'on désigne une fonction d'activation, dans le programme nous avons choisi la fonction de *Sigmoid*[5] (ce choix pourra être changé dans le futur), il existe une multitude de fonctions, notre choix est basé sur celui fait par *Andrew*. La fonction *Sigmoid* est définie par : $g(x) = \frac{1}{1+e^{-x}}$ dont la courbe est :

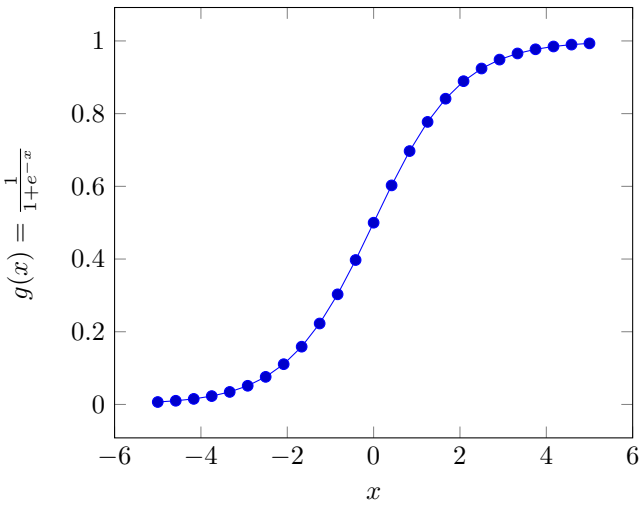

FIGURE 2 – Courbe de la fonction Sigmoid

Cette fonction a la particularité de converger très vite vers 1 ou 0 ce qui s'avère très utile puisque le résultat final doit être un booléan.

5. https ://fr.wikipedia.org/wiki/Sigmo%C3%AFde_%28math%C3%A9matiques%29

2.2.1 Fonction du cout

La fonction du cout permet de mesurer à quel point notre réseau est précis, cette fonction dépend de Θ et la minimiser permettera d'augmenter la précision de notre réseau.

Cette fonction est définie par :

$$J(\Theta) = -\frac{1}{m}[\sum_{i=0}^{m}\sum_{k=0}^{K} y_k^{(i)} log(h_\theta(x^{(i)})_k) + (1 - y_k^{(i)}) log(1 - h_\theta(x^{(i)})_k)] + \frac{\lambda}{2m}\sum_{l=0}^{L-1}\sum_{i=0}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ij}^{(l)})^2 \qquad (1)$$

Avec :

- m **:** Taille du dataset donnée en entrée (pour l'entrainement du classificateur).
- K **:** Taille du vecteur de sortie.
- y **:** Réponse à une donnée particulière par exemple 1 si l'image en entrée est une voiture.
- $h_\theta(x^{(i)})$ **:** Réponse donnée par le réseau de neurons (n'est pas nécéssairement égale à y).
- λ **:** : Paramètre fourni par l'utilisateur qui permet d'éviter *l'overfitting*[6].
- L **:** Nombre de couches.
- s_l **:** Nombre de noeuds dans la couche l.

Entrainer le réseau des neurones revient donc à trouver le Θ qui minimisera la fonction, cela se fera grace à des fonctions pré fournies par la bibliothèque *Dlib*. Il faut tout de même noter qu'en général cette fonction n'est pas convexe[7], l'algorithme de minimisation ne garantit donc en rien qu'on trouvera un minima global, en fait trouver un minimum global serai un problème NP-difficile[8] mais en pratique les minima locals peuvent donner des résultas satisfaisants si le réseau possède assez de couches intermédiaires[9].

Pour minimiser cette fonction nous utilisons le *BFGS*, un algorithme quasi-newtonien[3]. Cet algorithme est déja implémenté dans Dlib, et prends deux paramétres : une fonction et son gradient dans un point donné.

Pour calculer le gradient nous utiliserons un procédé nommé la *Back Propagation*. Il faut noter qu'il est également possible d'approcher ce gradient par des méthodes numérique, cette dernière à l'avantage d'etre plus facile et plus rapide à implémenter mais elle est beaucoup plus couteuse et ne sera donc pas utilisée, ces méthodes numériques permettent tout de même de vérifier si une implémentation de la *Back Propagation* est correcte, mais ne devra être utilisé que pour des fins de déboguage.

6. L'overfitting est le fait d'avoir un réseau qui fournit d'excellents réseultats sur le dataset d'entrainements mais a des performences médiocres sur des données qu'il n'a jamais vu
7. https ://stats.stackexchange.com/questions/106334/cost-function-of-neural-network-is-non-convex
8. Reference : Avrim Blum and Ronald L. Rivest, "Training a Three-Neuron Neural Net is NP- Complete", Neural Networks 5(1) January 1992. [https ://people.csail.mit.edu/rivest/pubs/BR93.pdf]
9. https ://stats.stackexchange.com/questions/203288/understanding-almost-all-local-minimum-have-very-similar-function-value-to-the

2.2.2 Back Propagation

La *Back Propagation* nous permettera de calculer les gradients de la fonction du cout.

Algorithme 1 : Back Propagation

Procedure *backPropagation()*
  Initialiser $\Delta_{ij}^{(l)} = 0$ pour tout l, i, j ;
  pour i *allant 1 à* m **faire**
    Calculer les différents $a^{(l)}$ par la forward propagation ;
    Calculer $\delta^{(L)} = a^{(L)} - y^{(i)}$;
    pour l *allant de L - 1 à 2* **faire**
      $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* {}^a g'(z^{(l)})$
    fin pour
    $\Delta_{kj}^{(l)} = \Delta_{kj}^{(l)} + a_j^{(l)} \delta_k^{(l+1)}$
  fin pour
  $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ (si $j \neq 0$)
  $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{ij}^{(l)}$ (si $j = 0$)
Fin

a. Multiplication élément par élément

Bien entendu la *Back Propagation* ne permet de calculer les gradients que pour un point donnée, elle sera donc appelé plusieurs fois, c'est pour ca qu'opter pour un algorithme utilisant un nombre d'itérations faibles (contrairement à un gradient descendant par exemple) peut considérablement réduire le temps du calcul.

Une fois la fonction $J(\theta)$ minimisée, le réseau est, en théorie, capable de prédire des résultats juste via une simple forward propagation pour sur une entrée donnée. Il faut tout de meme garder en tête que minimiser la fonction $J(\theta)$ peut causer un problème d'overfitting, ce qui sera résolu par l'essaie de plusieurs valeurs de λ (prametre optionnel du constructeur) pour le moment ZZBrain ne calcule pas le λ optimal, ce qui signifie que l'utilisateur devra lui même choisir des valeurs et les tester.

3 Structures de données

Stocker le réseau de neurones en mémoire revient à stocker les poids en mémoire, nous devons donc trouver une structure optimale pour stocker ces dernièrs, le temps de calcul n'étant vraiment nécéssaires que lors de la phase du training est peut donc être fait en cloud ou sur un cluster.

L'idée initiale était d'utiliser une matrice tridimensionnelle ayant des dimensions fixes. Mais celà poserai un problème de gachis puisque le nombre des poids différe d'une couche à une autre. Nous avons donc finalement choisi la strucure suivante.

La strucure défini ci-dessous représente l'implémentation de l'exemple en figure 1.

Nous devons noter toute meme que si le nombre des noeuds dans la première couche est s_0 le nombre des poids est $(s_0 + 1)s_1$ puisqu'on ajoute le noeud biais (qui n'est pas inclus dans nbNodesLayer1 ni dans le schéma de la figure 1)

Pour le moment les poids ne sont représenté que par un tableau statique, nous envisageons de changer celà, en effet Dlib offre un support assez complet des matrices, nous pourrons utiliser le type *matrix¡double, 0, 1¿*. Qui améliorera de facon significative la lisibilité et la perfermance du code (cf. Plus loin).

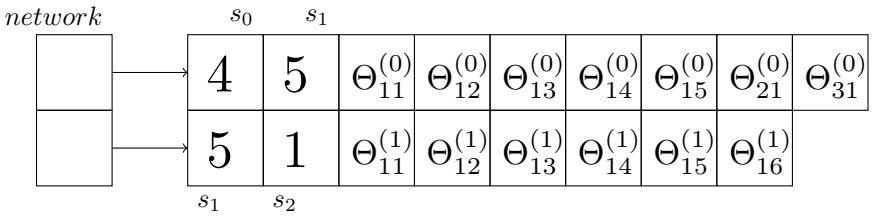

FIGURE 3 – Structure de donnée utilisé

4 ZZBrain

4.1 Intoduction

Dans cette section on va présenter les différentes fonctionnalités de la bibliothèque. On a décidé de partager cette dernière en plusieurs parties différentes, la première contient tout ce qui est calculs matriciels mais adapté à notre structure presentée ci-dessus. La deuxième partie contient les fonctions principales du réseau de neurones, et une dernière partie pour d'autres fonctions comme la Sigmoid.

4.2 Les fonctions matricielles

4.2.1 Organisation du code source

matrix.cpp : La bibliothèque des fonctions matricielles.

matrix.h : Fichier contenant les prototypes des fonctions de la bibliothèque.

4.2.2 Liste des fonctions

4.2.2.1 Multiplication matrice matrice

```
1 double * mult(double* M1, double* M2, int n, int m, int k)
```

Cette fonction fait la multiplication d'une matrice **M1(n x m)** par une deuxième matrice **M2(m x k)** et qui renvoie le résultat comme pointeur vers une troisième matrice allouée par la fonction.

4.2.2.2 Multiplication scalaire matrice

```
1 double * mult(double* M, double cst, int n, int m)
```

En utilisant la notion de la surcharge en C++ on a fait une deuxième fonction pour la multiplication mais cette fois-ci d'une matrice **M(n x m)** par une constante **cst** et qui renvoie le résultat comme pointeur vers une troisième matrice allouée par la fonction.

4.2.2.3 Multiplication élément par élément

```
1 double * eltMult(double * M1, double * M2, int n, int m)
```

Cette fonction fait la multiplication élément par élément d'une matrice **M1(n x m)** par une deuxième matrice **M2(n x m)** et qui renvoie le résultat comme pointeur vers une troisième matrice allouée par la fonction.

4.2.2.4 Somme des deux matrices

```
double * sum(double* M1, double* M2, int n, int m)
```

Cette fonction fait la somme des deux matrice **M1(n x m)** et **M2(n x m)** et qui renvoie le résultat comme pointeur vers une matrice allouée par la fonction.

4.2.2.5 Différence des deux matrices

```
double * diff(double * M1, double *M2, int n, int m)
```

Cette fonction fait la différence entre des deux matrice **M1(n x m)** et **M2(n x m)** et qui renvoie le résultat comme pointeur vers une matrice allouée par la fonction.

4.2.2.6 Transposée d'une matrice

```
double * trans(double* M, int n, int m)
```

Cette fonction fait le transpose d'une matrice **M(n x m)** et qui renvoie le résultat comme pointeur vers une matrice allouée par la fonction.

4.2.2.7 Vecteur des uns

```
double * ones(int n)
```

Cette fonction génère un vecteur de taille **n** qui ne contient que **des uns** et qui renvoie un pointeur vers ce dernier.

4.2.2.8 Fonction d'affichage

```
void print(double * M, int n, int m)
```

Cette fonction permet d'afficher une matrice **M(n x m)** à la sortie standard. Elle est principalement utilisé lors du débogage.

4.3 Les fonctions du réseau de neurones

4.3.1 Organisation du code source

ZZNetwok.cpp : La bibliothèque des fonctions principales du réseau.

ZZNetwok.h : Fichier contenant les prototypes des fonctions de la bibliothèque.

4.3.2 Liste des fonctions

4.3.2.1 Le constructeur

```
ZZNetwork(int sizes[],int nbLayers, int setSize, double **input, double **output, double lambda=0)
```

Le constructeur du réseau, On lui passe comme parametres :

int sizes[] : Un tableau des entiers qui contient la taille (nombre des noeuds) de chaque couche.

int nbLayers : Le nombre des couches.

int setSize : Le nombre des éléments de la dataset utilisé pour l'apprentissage .

double **input : La dataset utilisé pour l'apprentissage.

double **output : Les réponses (correctes) liée au dataset **input*, permet de valider les choix fait par le réseau.

lambda=0 : Le parametre de la régularisation (utilisé pour éviter l'overfitting) par défaut il a une valeur nulle (pas de régularisation).

Ses fonctionnalites :

- Initialisation des différentes variables du réseau par les valeurs passées en paramètre.
- Allocation des différentes parties de la structure du réseau.
- Initialisation des poids dans un intervalle **[-r,r]** bien precis . cet intervalle n'est adapté que lorsqu'on utilise la fonction *sigmoid* , où r est un nombre réel qu'on le calcule de la facon suivant [10] :

$$r = 4\sqrt{\frac{6}{network[i].nbNodesLayer1 + 1 + network[i].nbNodesLayer2}} \tag{2}$$

Si on décide par la suite d'utiliser une autre fonction cet intervalle devra être repensé.

4.3.2.2 La prediction / forward propagation

```
double * predict(double *input)
```

double *input : Un tableau de réels (idéalement normalisés).

La fonction renvoie un vecteur contenant si oui ou non la donnée en entrée appartient à la classe k. Pour celà nous implémentons la forward propagation comme suit :

Pour chaque couche du réseau on calcule un vecteur z et on applique la fonction sigmoid sur ce dernier pour avoir le vecteur final a .

```
memcpy(a, input, network[0].nbNodesLayer1 * sizeof(double));
for (int i = 0; i < nbLayers - 1; i++) {
        /* Add the bias node */
        memmove(a + 1, a, network[i].nbNodesLayer1 * sizeof(double));
        a[0] = 1.0;
        z = mult(network[i].weights, a, network[i].nbNodesLayer2, network[i].nbNodesLayer1 + 1, 1);
        a = sigmoid(z, network[i].nbNodesLayer2, 1);
        delete[] z;
}
```

4.3.2.3 La Retropropagation

```
double ***backPropagation()
```

L'implementation de l'algorithme de la retropropagation expliqué en haut, On obtient le gradient sous forme d'un tableau tri-dimensionnel **D** comme sortie. Cette fonction n'as besoin d'aucune entrée puisque qu'elle utilise les attributs du réseaux.

- Allocation des différentes tableaux et vecteur utilisés pour les calculs intermédiaire.
- Application de la forward propagation pour obtenir les vecteurs **a** intermédiaire pour chaque couche.

```
memcpy(a[0], input[i], network[0].nbNodesLayer1 * sizeof(double));
for (int l = 1; l < nbLayers; l++) {
    memmove(a[l - 1] + 1, a[l - 1], network[l - 1].nbNodesLayer1 * sizeof(double));
    a[l - 1][0] = 1;
    z[l] = mult(network[l - 1].weights, a[l - 1], network[l - 1].nbNodesLayer2,
                network[l - 1].nbNodesLayer1 + 1, 1);
```

10. https ://stats.stackexchange.com/questions/47590/what-are-good-initial-weights-in-a-neural-network

```
    a[l] = sigmoid(z[l], network[l - 1].nbNodesLayer2, 1);
}
```

— Calcul des differents δ :

```
err[nbLayers - 1] = diff(a[nbLayers - 1], output[i], network[nbLayers - 2].nbNodesLayer2, 1);

for(int l = nbLayers - 2; l > 0 ; l--){

    tr = trans(network[l].weights, network[l].nbNodesLayer2,
                network[l].nbNodesLayer1 + 1);
    theta_delta = mult(tr, err[l + 1], network[l].nbNodesLayer1 + 1,
                network[l].nbNodesLayer2, 1);
    one = ones(network[l].nbNodesLayer1 + 1);
    memmove(z[l] + 1, z[l], network[l].nbNodesLayer1 * sizeof(double));
    z[l][0] = 1.0;
    dif = diff(one, sigmoid(z[l], network[l].nbNodesLayer1 + 1, 1),
                network[l].nbNodesLayer1 + 1, 1);
    eltm1 = eltMult(sigmoid(z[l], network[l].nbNodesLayer1 + 1, 1), dif,
                network[l].nbNodesLayer1 + 1, 1);
    err[l] = eltMult(theta_delta, eltm1, network[l].nbNodesLayer1 + 1, 1) + 1;

    delete[] tr;
    delete[] theta_delta;
    delete[] dif;
    delete[] one;
    delete[] eltm1;
}
```

— Calcul des Δ :

```
for(int l = 0; l < nbLayers - 1; l++){
    tr = trans(a[l],network[l].nbNodesLayer1 + 1, 1);
    theta_delta = mult(err[l+1], tr, network[l].nbNodesLayer2, 1,
                    network[l].nbNodesLayer1 + 1);

    for(int k = 0; k < network[l].nbNodesLayer2; k++)
        for(int j = 0; j < network[l].nbNodesLayer1 + 1; j++)
            bigDelta[l][k][j] = bigDelta[l][k][j] + theta_delta[k *
                                        (network[l].nbNodesLayer1 + 1) + j];
}
```

— Calcul du gradient on utilisant Δ calculé à l'etape precedente, la valeur λ et les poids Θ.

```
for(int l=0; l < nbLayers - 1; l++){
   for(int i=0; i < network[l].nbNodesLayer2; i++) {
         D[l][i][0] = bigDelta[l][i][0]/setSize;
         for(int j=1; j < network[l].nbNodesLayer1 + 1; j++)
             D[l][i][j] = bigDelta[l][i][j]/setSize +
             lambda*network[l].weights[i * (network[l].nbNodesLayer1 + 1) + j];
    }
}
```

Note : l est le numero de l'intercouche , i est le numero du noeud de la première couche et j est le numero du noeud de la deuxième couche.

4.3.2.4 L'apprentissage

```
void train()
```

La fonction de l'apprentissage du programme. Pour utiliser les fonctions du minimisation de la bibliothèque *dlib* il faut adapter la structure des vecteurs a celle utilisée dans cette bibliothèque. Cette transformation est trés couteuse elle n'est que temporaire, remplacer les tableau double par des *matrix¡double, 0, 1¿* est nécéssaire. Pour faire ca on a créé une fonction qui transforme un vecteur ou matrice représentée sous notre structure à une structure compatible avec les fonctions de *dlib*.

```
find_min(bfgs_search_strategy(), objective_delta_stop_strategy(1e-7),
    [this](column_vector const&v) -> double { return costFunction(v); },
    [this](column_vector const&v) -> column_vector { return costFunctionDerivative(v); },
    init_theta, -1);
```

On utilise pour la minimisation la methode *BFGS*, c'est une méthode numérique utilisée pour résoudre des systèmes d'équations non linéaires dont on ne connaît pas forcément l'expression analytique. Concernant la condition d'arret nous avions le choix entre une difference de la valeur de la fonction objective entre deux itérations (option choisise), et celle de choisir une valeur minimal du gradient. Le problème avec la deuxième option c'est qu'il est difficile de garantir à l'avance que la condition d'arret sera réalisé.

Nous avons aussi utilisé des fonctions anonymes pour contourner le fait que nos fonctions sont des méthodes (et donc prennent le parametre this) alors que la fonction *find_min* attend des fonctions statiques.

5 Quelques Applications

Nous présenterons dans cette section des exemples concrets illustrant l'utilisaton de ZZBrain, ce ne sont que des fonctions logiques, ils n'ont donc aucun intérét pratique puisqu'ils peuvent être codé plus aisement de façon directe.

Nous tenons à préciser un point, il peut sembler évident qu'une dataset contenant tous les cas possibles (2 pour une fonction unaire, 4 pour une fonction binaire) sera largement suffisante, mais apparement (résultats expérimentaux) il s'avére que le réseau est beaucoup plus performent si on lui fournit une dataset plus large (par exemple de taille 1000), même si celle-ci ne contient en réalité que 2 entrés répétés plusieurs fois. Pour être tout à fait honnete, nous ne savons toujours pas pourquoi c'est le cas.

5.1 Réseaux sans couches intérmédiaires

5.1.1 Non logique

```
#include <iostream>
#include "lib/ZZNetwork.h"

using namespace std;

int main() {

    int sizes[] = {1, 1};
    int nbLayers = 2;

    double **X = new double*[1000];
    double **Y = new double*[1000];


```

```
    for(int i = 0; i < 1000; i++)
    {
        X[i] = new double[1];
        Y[i] = new double[1];
    }


    for(int i = 0; i < 500; i++)
    {
        X[i][0] = 0.0;
        Y[i][0] = 1.0;
    }
    for(int i = 500; i < 1000; i++)
    {
        X[i][0] = 1.0;
        Y[i][0] = 0.0;
    }

    ZZNetwork notNet(sizes, nbLayers, 1000, X, Y);
    notNet.train();

    cout << "not(0) = " << notNet.predict(X[0])[0] << endl;
    cout << "not(1) = " << notNet.predict(X[999])[0] << endl;

    return 0;

}
```

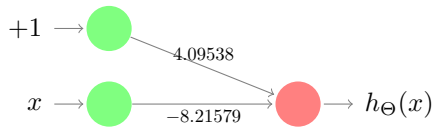

FIGURE 4 – Réseau Non logique généré par ZZBrain

5.1.2 And logique

```
#include <iostream>
#include "lib/ZZNetwork.h"

using namespace std;

int main() {

    int sizes[] = {2, 1};
    int nbLayers = 2;

    double **X = new double*[1000];
    double **Y = new double*[1000];


    for(int i = 0; i < 1000; i++)
    {
        X[i] = new double[2];
```

```
18          Y[i] = new double[2];
19      }
20
21
22      for(int i = 0; i < 250; i++)
23      {
24          X[i][0] = 0.0;
25          X[i][1] = 0.0;
26          Y[i][0] = 0.0;
27      }
28      for(int i = 250; i < 500; i++)
29      {
30          X[i][0] = 1.0;
31          X[i][1] = 0.0;
32          Y[i][0] = 0.0;
33      }
34      for(int i = 500; i < 750; i++)
35      {
36          X[i][0] = 0.0;
37          X[i][1] = 1.0;
38          Y[i][0] = 0.0;
39      }
40      for(int i = 750; i < 1000; i++)
41      {
42          X[i][0] = 1.0;
43          X[i][1] = 1.0;
44          Y[i][0] = 1.0;
45      }
46
47
48
49      ZZNetwork notNet(sizes, nbLayers, 1000, X, Y);
50      notNet.train();
51
52      cout << "and(0, 0) = " << notNet.predict(X[0])[0] << endl;
53      cout << "and(1, 0) = " << notNet.predict(X[300])[0] << endl;
54      cout << "and(0, 1) = " << notNet.predict(X[600])[0] << endl;
55      cout << "and(1, 1) = " << notNet.predict(X[900])[0] << endl;
56
57      return 0;
58
59  }
```

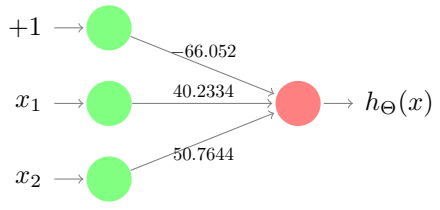

FIGURE 5 – Réseau And logique généré par ZZBrain

5.2 Réseaux avec couches intérmédiaires

5.2.1 XNOR logique

```cpp
#include <iostream>
#include "lib/ZZNetwork.h"

using namespace std;

int main() {

    int sizes[] = {2, 4, 1};
    int nbLayers = 3;

    double **X = new double*[1000];
    double **Y = new double*[1000];


    for(int i = 0; i < 1000; i++)
    {
        X[i] = new double[2];
        Y[i] = new double[2];
    }


    for(int i = 0; i < 250; i++)
    {
        X[i][0] = 0.0;
        X[i][1] = 0.0;
        Y[i][0] = 1.0;
    }
    for(int i = 250; i < 500; i++)
    {
        X[i][0] = 1.0;
        X[i][1] = 0.0;
        Y[i][0] = 0.0;
    }
    for(int i = 500; i < 750; i++)
    {
        X[i][0] = 0.0;
        X[i][1] = 1.0;
        Y[i][0] = 0.0;
    }
    for(int i = 750; i < 1000; i++)
    {
        X[i][0] = 1.0;
        X[i][1] = 1.0;
        Y[i][0] = 1.0;
    }



    ZZNetwork notNet(sizes, nbLayers, 1000, X, Y);
    notNet.train();

    cout << "xnor(0, 0) = " << notNet.predict(X[0])[0] << endl;
```

```
53      cout << "xnor(1, 0) = " << notNet.predict(X[300])[0] << endl;
54      cout << "xnor(0, 1) = " << notNet.predict(X[600])[0] << endl;
55      cout << "xnor(1, 1) = " << notNet.predict(X[900])[0] << endl;
56
57      return 0;
58
59  }
```

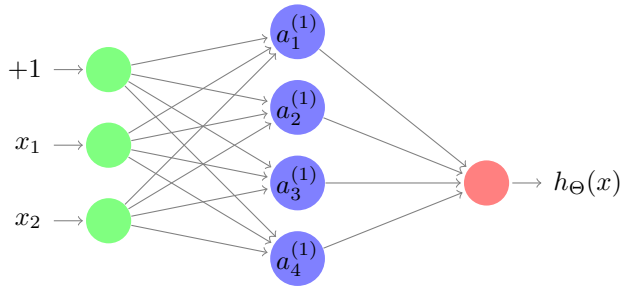

FIGURE 6 – Réseau XNOR généré par ZZBrain

Les poids générés sont données dans la matrice suivante :

$$\Theta^{(0)} = \begin{pmatrix} -1.73473 & -1.75034 & -5.93952 & -5.48223 \\ 33.4988 & -46.5189 & -1.83867 & -19.2062 \\ 16.4799 & 29.3001 & -0.18966 & -37.6205 \end{pmatrix}$$

$$\Theta^{(1)} = \begin{pmatrix} -43.7497 & -85.383 & 3.75306 \end{pmatrix}$$

6 Aller plus loin

Il y a plusieurs améliorations qui peuvent être étudiés, afin d'améliorer le projet.

Utiliser Dlib de façon plus intensive permettera d'éviter un trés grand nombre de calculs inutiles, et permettera d'avoir un code plus lisible, par exemple pour additioner deux matrices il suffira de faire A + B, au lieu d'appeler une matrice avec plein de parametres pour faire celà.

Donner la possibilité à l'utilisateur de choisir la fonction d'activation qui lui convient, cette amélioration néamoins obligera l'utilisateur à coder sa propre fonction (ainsi que son dérivé correspadente) ce qui pourra allourdir l'utilisation de la bibliothéque, elle causera également un problème au niveau des parametres initiales, vu que le domaine choisi est étroitement liée à la fonction d'activation utilisé.

Régler le problème des minimas locals, en lançant plusieurs fois la fonction de minimisation (avec des thetas initials différentes) et de prendre la fonction la plus minimale (cette méthode présente néanmoins un cout de calcul énorme). Il existe aussi des méthodes que nous n'avons pas eu l'occasion de creuser plus pronfondément dont les auteurs prétendent qu'ils permetteront de contourner les problèmes des minimas locaux.

Ajouter un module permettant de tester le réseau (aprés la phase de l'entrainement) sur un autre dataset pour voir à quel point le réseau est précis sur des données qu'il n a jamais étudié.

Automatiser le calcul de λ (dépendera de la proposition précédante).

Ecrire une partie de la fonction *train* en *CUDA*, qui permettera (si l'utilisateur le souhaite) de profiter de la puissance des GPU récents (dont certains sont optimisés pour les opérations en virgule flotantes et les calculs matricielles) au lieu d'effectuer les calculs sur le processeur.

Ressources utilisées

[1] *Documentation DLib (partie optimization).* URL : `http://dlib.net/optimization.html`.

[2] *Machine Learning By Andrew Ng.* URL : `https://www.coursera.org/learn/machine-learning`.

[3] Jonas Koko Philipe Maey Vincent Barra. *Cours analyse numérique.*

[4] Barbara E.Moo Stanley B.Lippman Josée Lajoie. *C++ Primer.*