Rapport Projet ZZBrain

ISIMA

Imad ENNEIYMY
enneiymy.i@gmail.com

Yassir Karroum
ukarroum17@gmail.com

6 juin 2017

Table des matières

1 Introduction **3**

2 Réseaux de neurones **3**
2.1 Introduction . . . 3
2.2 Structure du réseau . . . 4
2.2.1 Fonction du cout . . . 6
2.2.2 Back Propagation . . . 7

3 Structures de données **7**

4 ZZBrain **7**
4.1 Intoduction . . . 7
4.2 Les fonctions matricielles . . . 7
4.2.1 Organisation du code source . . . 7
4.2.2 Liste des fonctions . . . 8
4.2.2.1 Multiplication matrice matrice . . . 8
4.2.2.2 Multiplication scalaire matrice . . . 8
4.2.2.3 Multiplication élément par élément . . . 8
4.2.2.4 Somme des deux matrices . . . 8
4.2.2.5 Différence des deux matrices . . . 8
4.2.2.6 Transposée d'une matrice . . . 8
4.2.2.7 Matrice des uns . . . 8
4.2.2.8 Fonction d'affichage . . . 8
4.3 Les fonctions du réseau de neurone . . . 9
4.3.1 Organisation du code source . . . 9
4.3.2 Liste des fonctions . . . 9
4.3.2.1 Le constructeur . . . 9
4.3.2.2 La prediction / propagation . . . 9
4.3.2.3 La Retropropagation . . . 9
4.3.2.4 L'apprentissage . . . 10

5 Quelques Applications **10**

6 Aller plus loin **10**

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 pareil 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.

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 il y a deux semaines 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 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 neurons peut etre representé comme suit :

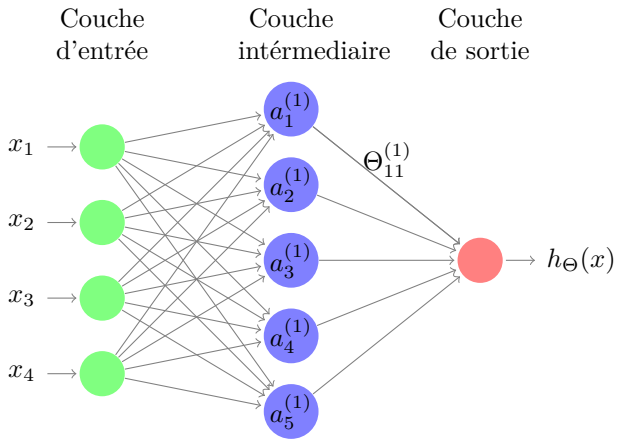

FIGURE 1 – Exemple d'un reseau de neurones

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 :

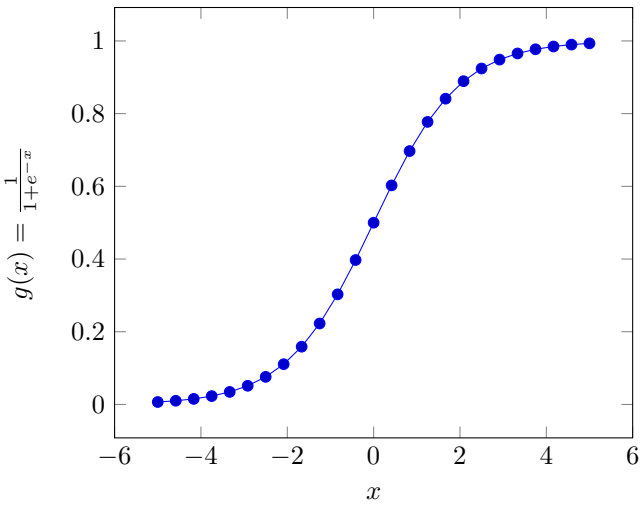

FIGURE 2 – Courbe de la fonction Sigmoid

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 **:** : Parametre fournie 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

4 ZZBrain

4.1 Intoduction

Dans cette section on va présenter les différentes fonctionnalités de la bibliothèque. On a opté pour partager cette dernière en plusieurs parties différentes, la première contient tout ce qui est calculs matriciels mais compatible avec notre structure presente en haut. La deuxième partie contient les fonctions principales du réseau de neurones, et une dernière partie pour tous autres fonctions diverses utilisées dans la bibliothèque.

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

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

double * mult(double* M, double cst, int n, int m) :
En utilisant la notion de la surcharge de 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

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 Matrice des uns

double * ones(int n) :
Cette fonction génère une matrice carrée de taille **n** qui contient seulement **des uns** et qui renvoie un pointeur vers cette dernière.

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 .

4.3 Les fonctions du réseau de neurone

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=0.01) : le constructeur du réseau, On lui passe comme parametre :

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

int nbLayers : Le nombre des couche.

int setSize : Le nombre des éléments de l'ensembre d'apprentissage .

double **input : Un tableau de vecteurs d'entrées.

double **output : Un tableau de vecteurs de sorties.

lambda=0.01 : un parametre utilisé dans le processus d'apprentissage. On lui affecte 0.01 par defaut.

Ces 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 marche seulement qu'on utilise la fonction *sigmoid* , ou r est un double qu'on le calcule de la facon suivant :

```
r = 4*sqrt(6.0/(network[i].nbNodesLayer1 + 1 + network[i].nbNodesLayer2));
```

Bien évidemment si on opte à utiliser une fonction autre que *sigmoid* cet intervalle n'est plus utile.

4.3.2.2 La prediction / propagation

double * predict(double *input) :

la fonction du prediction des résultats d'après un ensemble des entrées passées en paramètre :

double *input : Un tableau des entrées.

On obtient à la fin un tableau **a** des sorties prédit par l'algorithme de la propagation. 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** .

```
z = mult(network[i].weights, a, network[i].nbNodesLayer2, network[i].nbNodesLayer1 + 1, 1);
a = sigmoid(z, network[i].nbNodesLayer2, 1);
```

4.3.2.3 La Retropropagation

double ***backPropagation() :

l'implementation de l'algorithme de la retropropagation expliqué en haut, On obtient le gradient un tableau tri-dimensionnel **D** comme sortie par contre elle prend aucune entrée :

— Allocation des différentes tableaux et vecteur utilisés pour les calculs intermédiaire.

— Application de la propagation pour obtenir les vecteurs **a** intermédiaire pour chaque couche.

— Calcul d'erreurs entre les vecteurs **a** obtenus à l'étape précédente et les sorties donnée par l'utilisateur lors de la creation du reseau

```
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;
```

— Calcul du gradient accumulateur grand delta pour chaque couche

```
bigDelta[l][k][j] = bigDelta[l][k][j] + theta_delta[k*(network[l].nbNodesLayer1+1)+j];
```

— Calcul du gradient on utilisant le grand delta calcule a l'etape precedente, la valeur lambda et les poids

```
D[l][i][j]=bigDelta[l][i][j]/setSize+lambda*network[l].weights[i*(network[l].nbNodesLayer1+1)+
```

Note : l est le numero de , i est le numero de et j est le numero de

4.3.2.4 L'apprentissage

void train() : la fonction de l'apprentissage du programme , on reçoit les resultat sur la sortie standard. Pour utiliser les fonctions du minimisation de la bibliothèque *dlib* il faut adapter la structure des vecteurs a celle utilisée dans cette bibliothèque. 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*. On utilise pour la minimisation la methode *Quasi-Newton*, c'est une méthode numérique utilisée pour résoudre des systèmes d'équations non linéaires dont on ne connaît pas forcément l'expression analytique. une deuxieme raison pour laquelle on a opté pour cette methode c'est la conditions d'arrêt utilisé ; et c'est qu'on l'ecart entre deux iterations devient inférieur a 10^{-7} et on arrive toujours cette condition notre l'algorithme utilisé.

5 Quelques Applications

6 Aller plus loin

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.*