Réseaux de neurones

BAYEMAN LAWIN ET MATTHIEU BIZIEN

22 février 2014

Résumé

Nous avons implémenté un réseau de neurones de type perceptron multi-couches pour la classification. Le perceptron peut avoir un nombre de couches cachées quelconque. Le réseau est optimisé en utilisant la descente de gradient RProp, via l'algorithme de backpropagation. Pour lire les données un parser CSV a été implémenté. Deux jeux de données ont été utilisés pour l'application. Le premier, un jeu de données sur 3 types de vins avec 13 attributs a permis d'observer 29 observations biens classés sur 30 dans l'échantillon test au bout d'une vingtaine d'itérations de l'algorithme de la descente de gradient. Le deuxième, un jeu de données sur la reconnaissance de chiffres manuscrits (0 à 9) avec 784 attributs a permis d'atteindre au bout de 1000 itérations, 86.14% de biens classés dans l'échantillon test de 5000 observations après avoir utilisé un échantillon d'apprentissage de 5000 observations. Pour une configuration donnée (nombre de couches cachées, nombre de neurones par couche), le choix du nombre optimal d'itérations pourrait être fait par la validation croisée.

1 INTRODUCTION

Les réseaux de neurones permettent un apprentissage supervisé puissant. Bien qu'ils soient aient moins de fondements théoriques que d'autres algorithmes, ils fonctionnent bien en pratique et les dernières générations ont des résultats state-of-the-art. Ils ont reçu un regain d'intérêt ces dernières années grâce à de nouveaux algorithmes, la disponibilité de base de données importantes et la puissance de calcul des ordinateurs et des cartes graphiques actuelles.

Le réseau de référence est le perceptron multi-couches. Le perceptron multi-couches est un classifieur linéaire de type réseau neuronal formel organisé en plusieurs couches au sein desquelles une information circule de la couche d'entrée vers la couche de sortie uniquement. La sortie de chaque couche est linéairement proportionelle à la sortie de la nouvelle. Chaque couche est constituée d'un nombre variable de neurones, les neurones de la couche de sortie correspondant toujours aux sorties du système.

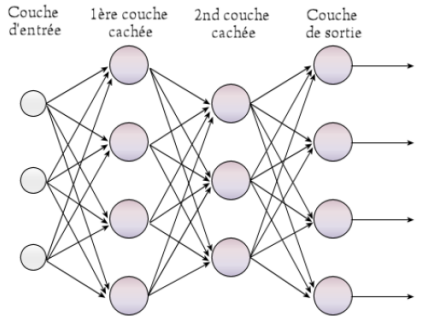


FIGURE 1 – Exemple de perceptron multi-couches avec 4 couches

Nous avons porté un effort important sur la documentation du code. Cela nous permet d'utiliser Doxygen pour générer la documentation pour

notre projet. La description détaillée de notre code a été créée au format pdf automatiquement. Elle est incluse en annexe à la fin du présent rapport.

Nous avons choisis de porter un effort particulier aux tests et à la fiabilité. 500 lignes réparties sur 20 tests assurent la solidité de notre code. Nous avons pour toutes nos opérations utilisées des double pour minimiser le risque d'overflow, au coût d'une performance moindre. Celle-ci est néanmoins très convenable et permet d'utiliser des réseaux de taille respectable.

Tout le code est disponible sur Github, à l'adresse `https://github.com/MatthieuBizien/neural_networks` Il est utilisable aisément grâce à l'utilisation de qmake pour générer les makefiles.

2 Mise en oeuvre du reseau de neurones

2.1 *L'architecture*

Nous avons choisis une architecture simple mais néanmoins flexible. Nous avons décomposé le programme en quatre modules. Le premier est neuralnets, qui contient l'implémentation du perceptron multi-couches. Le second est gradientdescent, où nous avons implémenté la descente de gradient classique et l'algorithme Rprop. Le troisième est parsercsv, qui permet de transformer un fichier CSV - comma separated values - en matrices. Le quatrième et dernier est utils, regroupant différents objets et fonctions utilisés dans le reste du projet.

Cette architecture permet d'ajouter facilement de nouveaux algorithmes de descente de gradients et de nouveaux réseaux, car le format de communication entre les deux est un vecteur de doubles.

Les données sont stockées sous forme matricielle. Elles sont séparées en deux parties, la matrice d'entrée X et la matrice de résultat Y, ayant autant de lignes. Chaque ligne de (X, Y) est une observation. Y est une matrice de variables binaires, où chaque colonne représente une catégorie. Celles-ci peuvent partitionner l'espace d'arrivée ou non, c'est-à-dire que chaque ligne peut être égale à un ou non. Dans le cas de la classification binaire, Y ne comporte qu'une seule colonne. Pour la classification multi-classe, Y doit comporter autant de colonnes qu'il y a de classes. Il faut normaliser les prédictions avec l'algorithme "seul contre tous", où la probabilité d'une prédiction i pour la classe k est $\frac{\hat{Y_{i,k}}}{\sum_{k'=1..K} \hat{Y_{i,k'}}}$. Cela permet que la somme des probabilités soit bien égale à un.

Nous utilisons pour toutes nos opérations une librairie de calcul matriciel, Eigen. Celle-ci permet d'avoir de bonnes performance de calculs. Elle est distributée sous la forme de header seuls, comme notre programme - à l'exception du main. Eigen est également utilisable sur GPU, ce qui permettrait des performances dans l'état de l'art s'il était nécessaire.

2.2 *Le perceptron multi-couches*

La classe principale est MultiLayerPerceptron, qui modélise un perceptron multi couches. Cette classe permet de calculer la sortie du réseau avec une entrée donnée, ainsi que le gradient de backpropagation pour un couple de données d'entrées et de sorties. [6] Elle peut avoir un nombre de couches cachées quelconque. Une absence de couches cachées revient à faire une régression logistique.

Le réseau est optimisé en utilisant la descente de gradient pour minimiser l'erreur $E = \sum_{i=1...n,j=1..k} Y_{i,j} \log(\hat{Y}_{i,j}) + (1 - Y_{i,j}) \log(1 - \hat{Y}_{i,j})$, où $\hat{Y}_{i,j}$ est la probabilité, selon le réseau de neurones, que $Y_{i,j}$ soit égal à un. Cette erreur correspond à l'inverse de la vraisemblance.

Les poids d'un réseau de neurones sont stockés comme une suite de matrice, dans un objet de type MatrixSequence. Cet objet permet de passer de cette représentation à la représentation sous forme de vecteur unidimen-

sionnel, formé de la concaténation des colonnes des matrices. Cela permet de simplifier le lien entre réseau de neurone et les algorithmes de descente de gradient en considérant l'écriture vectorielle.

Nous utilisons l'algorithme de backpropagation, en utilisant l'astuce que la dérivée de la fonction sigmoid est $sig' = sig(1 - sig)$. Nous calculons les sorties des différentes couches, en partant de la première, la plus proche de X, jusqu'à la dernière. Le calcul de la dérivée se fait en sens inverse. Nous stockons les sorties des couches et les dérivées dans un objet MatrixSequence.

2.3 *La descente de gradient*

Deux algorithmes de descente de gradient sont implémentés. Ils sont implémentés dans des classes à part qui dérivent le la classe abstraite ACostFunctionMinimizeur. Le premier est la descente de gradient classique, où chaque poids $W_{i,j}$ du réseau est mis à jour selon la formule

$$W_{i,j}^{n+1} \leftarrow W_{i,j}^{n} - \epsilon \left.\frac{dE}{dW_{i,j}}\right|^{n}$$

où ϵ est un paramètre qui doit être ajusté pour que la convergence se fasse en temps raisonnable sans diverger.

Le deuxième algorithme est Rprop, décrit par Riediller et al. [5]. Il permet d'individualiser le paramètre ϵ pour chaque poids. Le paramètre $\epsilon_{i,j}^{n}$ est mis à jour selon l'évolution du signe du gradient. Si le gradient n'a pas changé de signe, ce paramètre augmente, sinon il diminue. Les hausses et baisses optimales sont, selon les auteurs, de 1.2 et 0.5. Cela permet de favoriser les directions où le gradient diminue constament, et de diminuer le paramètre quand on se rapproche de l'optimum. La mise à jour des poids se fait selon l'algorithme

$$\begin{cases} W_{i,j}^{n+1} & \leftarrow & W_{i,j}^{n} - \epsilon_{i,j}^{n} \left.\frac{dE}{dW_{i,j}}\right|^{n} \\ \epsilon_{i,j}^{n+1} & \leftarrow & \kappa_{i,j}^{n} \epsilon_{i,j}^{n} \\ \kappa_{i,j}^{n} & \leftarrow & 1.2 \text{ si signe inchangé sinon } 0.5 \end{cases}$$

2.4 *Tests*

Nous avons attaché une importance particulière à avoir une application testée. La librairie de tests unitaires utilisée UnitTest++, sous licence MIT. Nous avons créé un fichier de tests par objet. La solidité de notre programme est vérifiée par 20 tests unitaires, s'exécutant en moins de 50 millisecondes.

Le test le plus simple est celui de MatrixSequence. Nos tests vérifient que les dimensions des matrices trouvées sont bonnes, que l'affichage fonctionne, que l'on peut récupérer les données sous forme de vecteurs ou de matrices, que si l'on modifie une matrice ou un vecteur l'objet change de façon cohérente. L'opérateur copie doit fournir une nouvelle séquence de matrices indépendante de l'ancienne. Le fichier de test est long de 113 lignes.

Le perceptron est testé de plusieurs manières. Nous vérifions que différents perceptrons avec des valeurs codées en dur fonctionnent bien, comme un perceptron avec des sorties nulles, un perceptron indépendant de l'entrée et un perceptron implémentant l'opérateur XOR. Nous vérifions l'opérateur copie. Nous vérifions que le gradient est correctement implémenté, en calculant la dérivée marginale empirique selon les différents poids. Nous utilisons pour cela le perceptron XOR, et plusieurs perceptrons aléatoires créés avec des seeds codées en dur, pour la reproductibilité des résultats. Cette vérification nous a permis de déceler plusieurs bugs dans notre implémentation de la backpropagation. Le fichier de test est long de 183 lignes.

Les algorithmes de descente de gradient ont été plus compliqués à tester. Nous avons vérifié qu'ils faisaient converger un perceptron vers des poids approximant correctement une fonction. Nous avons testé la fonction XOR

avec deux et avec trois entrées - noté XOR3 - et des entrées codées en dur. Nos algorithmes convergent en 100 et 700 itérations. Nous avons du faire attention à avoir une convergence rapide, afin de ne pas perdre trop de temps pendant l'exécution des tests. Pour la reproductibilité, nous avons fixé les seeds du générateur de nombres aléatoires. Les fichiers de tests sont longs de 74 et 75 lignes.

3 APPLICATION DU RESEAU DE NEURONES A LA PREDICTION

3.1 *Données wine [1]*

Les données wine portent sur 3 types de vins avec 13 caractéristiques numériques. Nous disposons de 178 observations au total. Nous avons donc créé un échantillon d'apprentissage de 148 observations et un échantillon test de 30 observations.

Nous avons construit 4 différents types de perceptron multi-couches :

— une couche cachée (avec 10 neurones et 20 neurones) Les résultats obtenus dans les deux échantillons sont présentés dans la figure 2.

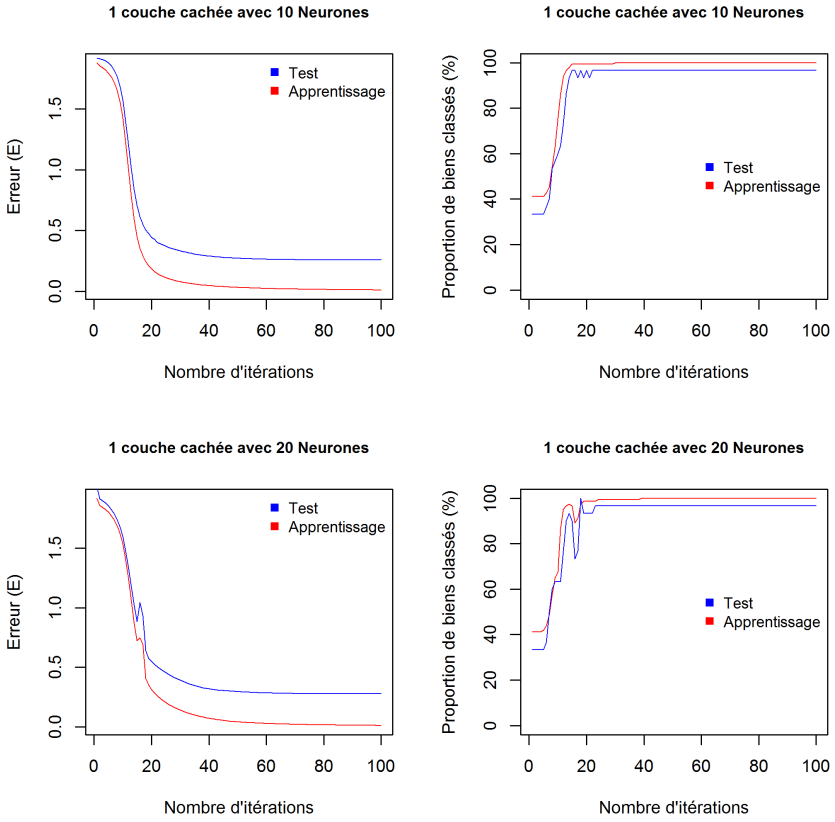


FIGURE 2 – Evolution de l'erreur (E) et de la proportion de bien classés dans les deux échantillons (données wine) (une couche cachée)

— 2 couches cachées (avec 5 neurones chacunes et 10 neurones chacunes) Les résultats obtenus dans les deux échantillons sont sont présentés dans la figure 3.

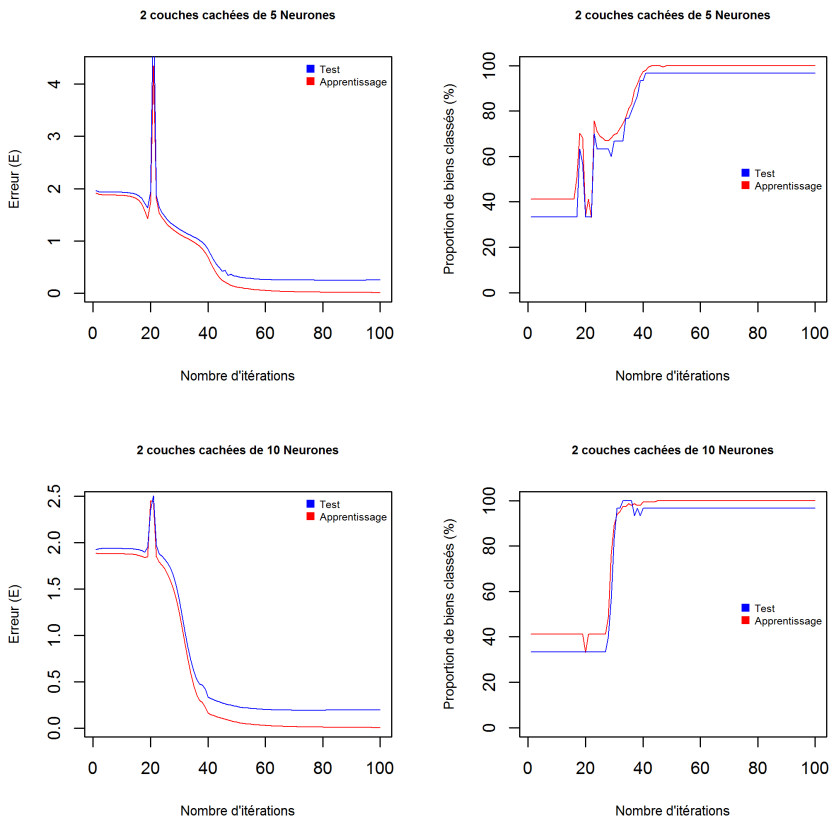


FIGURE 3 – Evolution de l'erreur (E) et de la proportion de bien classés dans les deux échantillons (données wine) (2 couches cachées)

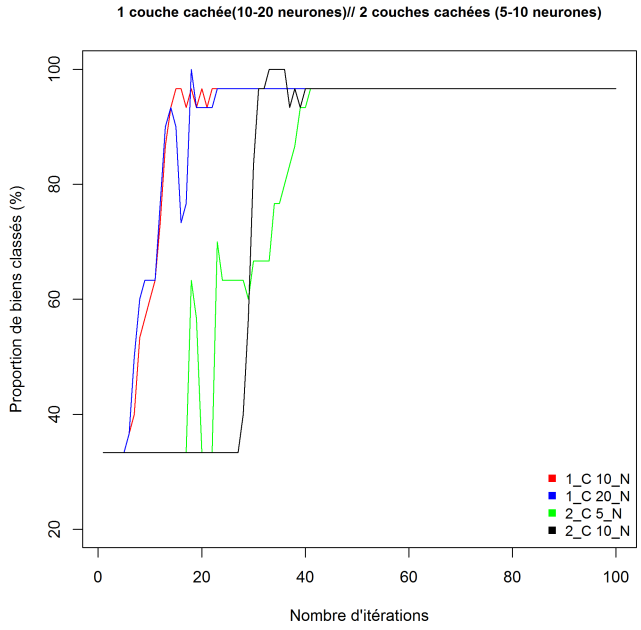


FIGURE 4 – Evolution de la proportion de bien classés dans l'échantillon test (données wine (une couche cachée / 2 couches cachées))

La figure 4 présente l'évolution de la proportion de bien classés dans l'échantillon test pour les différentes configurations. Dans les différents cas, les réseaux arrivent à bien classer 29 observations sur 30 au bout des 100 itérations (Ce nombre de bien classés est atteint généralement avant) sur l'échantillon test. En particulier, sur le réseau à une couche cachée

avec 20 neurones, à la 18 ième itération, le réseau classe bien 100% des observations ; il en est de même pour le réseau à deux couches cahées avec 10 neurones par couche à la 33 ième itération. Cela suggère, que pour obtenir le nombre optimal d'itérations, il faudrait préalablement faire de la validation croisée. Enfin, nous avons essayé une configuration de réseau avec 2 couches cachées de 100 neurones chacunes. On obtient les même résultat en termes d'obsersations bien classées, en particulier 29 dans l'échantillon test. On peut donc que remarquer qu'avec un nombre de neurones relativement grand, la performance n'est pas dégradée d'une part, et qu'on arrive pas à 100% de bien classés d'autre part. Toutefois, ceci est à prendre avec précaution, la base de données wine étant de taille petite.

3.2 *Données MNIST [4]*

La base de données MNIST porte sur des chiffres manuscrits. Le but est de reconnaitre les chiffres (0 à 9) sur une image en noir et blanc. Nous disposons d'une base de 70000 observations avec 785 features (dont la variable d'intérêt Y). Pour faire nos essais, nous avons tiré des échantillons de taille plus petite de façon aléatoire :

— Un (unique) échantillon test de 5000 observations.
— Quatre échantillons d'apprentissage de 1000, 2000, 5000 et 10000 observations.

Les configurations utilisées pour le perceptron multi-couches sont les suivantes :

— une couche cachée de 100 neurones sur les différents échantillons d'apprentissage.
— une couche cachée de 200 neurones sur les différents échantillons d'apprentissage.
— 2 couches cachées de 100 neurones chacune sur les différents échantillons d'apprentissage.

Nous avons choisi d'utiliser l'algorithme RProp sur 1000 itérations. Notre algorithme effectue une itération en une seconde sur nos ordinateurs, pour un réseau de 100 neurones cachés et un échantillon d'apprentissage de taille 5000, calcul du score de test inclus. Cette bonne performance est permise grâce à l'utilisation d'une librairie de calculs linéaires optimisée et au faible surcou t de notre architecture.

3.2.1 *Une couche cachée de 100 neurones*

Les résultats obtenus pour les échantillons d'apprentissage de tailles 1000 et 2000 sont présentés dans la figure 5 ; ceux obtenus pour les échantillons d'apprentissage de tailles 5000 et 10000 sont présentés dans la figure 6.

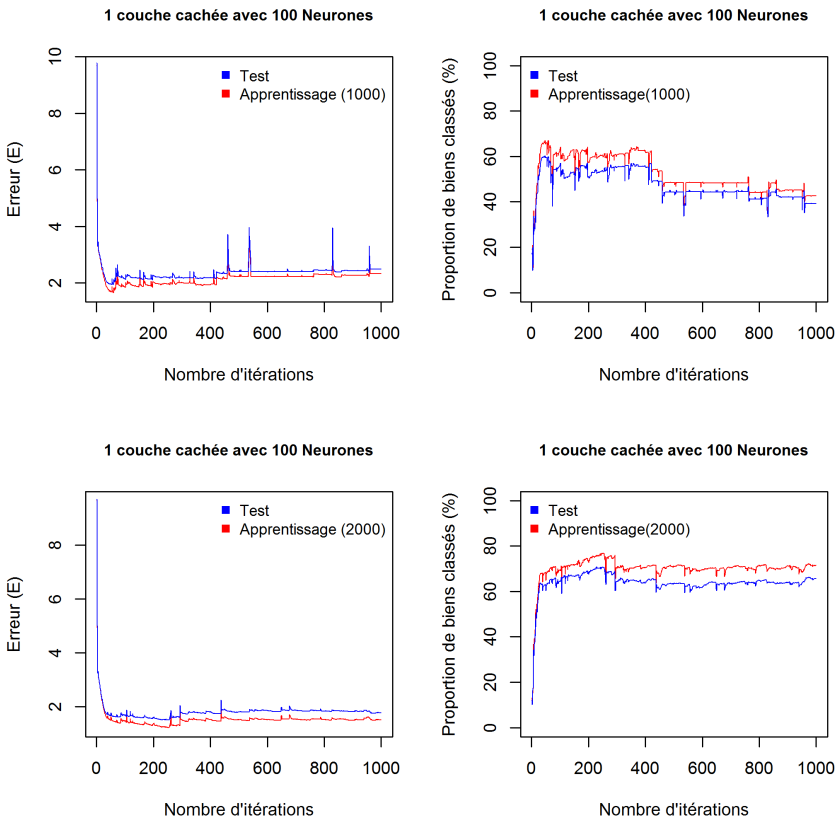


FIGURE 5 – Evolution de l'erreur (E) et de la proportion de bien classés dans les deux échantillons (données mnist (1000 et 2000 obs.) (une couche cachée))

La figure 7 présente l'évolution de la proportion de bien classés dans l'échantillon test pour les différents échantillons d'apprentissage. Au bout de 1000 itérations, cette proportion a les valeurs suivantes :

— échantillon d'apprentissage de 1000 observations : 39,22%
— échantillon d'apprentissage de 2000 observations : 65,68%
— échantillon d'apprentissage de 5000 observations : 76,8%
— échantillon d'apprentissage de 10000 observations : 79,28%

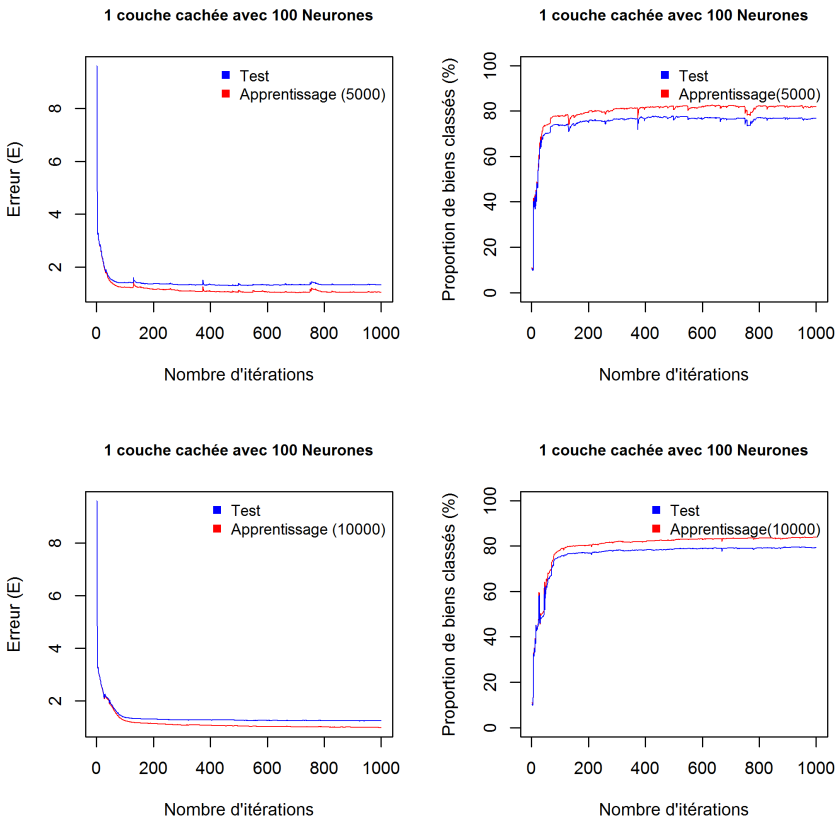


FIGURE 6 – Evolution de l'erreur (E) et de la proportion de bien classés dans les deux échantillons (données mnist (5000 et 10000 obs.) (une couche cachée))

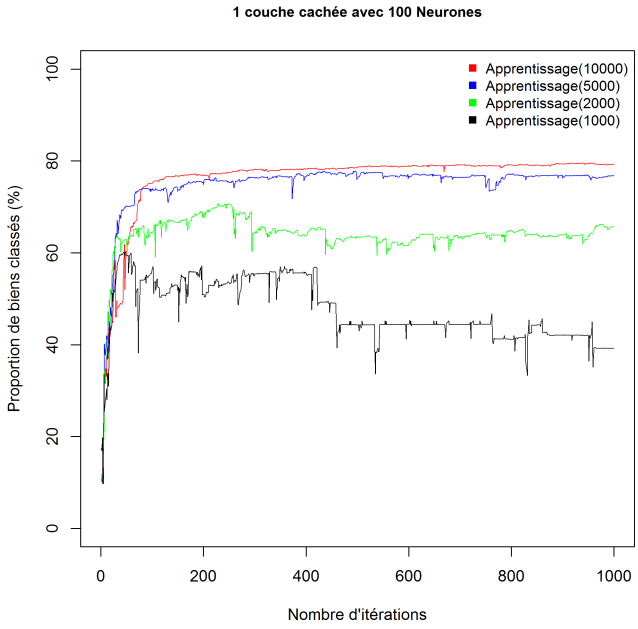


FIGURE 7 – Evolution de la proportion de bien classés dans l'échantillon test (données mnist (1000, 2000, 5000 et 10000 obs.) (une couche cachée))

3.2.2 *Une couche cachée de 200 neurones*

Les résultats obtenus dans les deux échantillons sont présentés dans la figure 8. On constate que cette configuration de 200 neurones pour l'unique

couche cachée fait mieux que la configuration à 100 neurones à nombre égal d'itérations. Au bout des 1000 itérations, ce réseau arrive à 86,14% de bien classés sur l'échantillon test. Précisons que selon les taux d'apprentissage fixés, la convergence peut ne pas être régulière car le pas de l'algorithme RProp est probablement trop important. Néanmoins celui-ci arrive à limiter la divergence pour arriver à une solution bien meilleure qu'un tirage aléatoire.

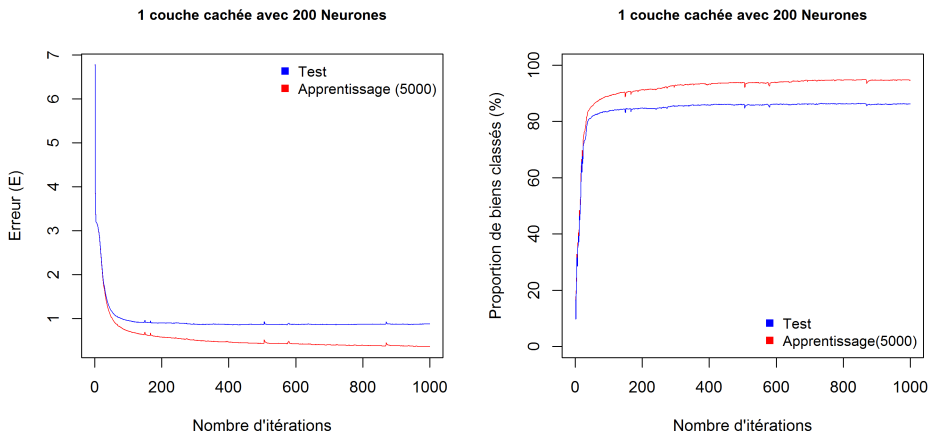


FIGURE 8 – Evolution de la proportion de bien classés dans l'échantillon test (données mnist une couche cachée de 200 neurones))

3.2.3 *Deux couches cachées de 100 neurones chacunes*

Notre programme prend tout son sens lorsque l'on considère plusieurs couches. Nous avons testé pour les échantillons d'apprentissage de 5000 et de 10000 observations. Pour ce dernier, nous ne sommes pas allé jusqu'au bout des 1000 itérations à cause du manque de temps avant la deadline du projet. Les performances sont ici supérieures à celles que nous avions eu précédement, avec une performance finale de 79.87% et 85.94%.

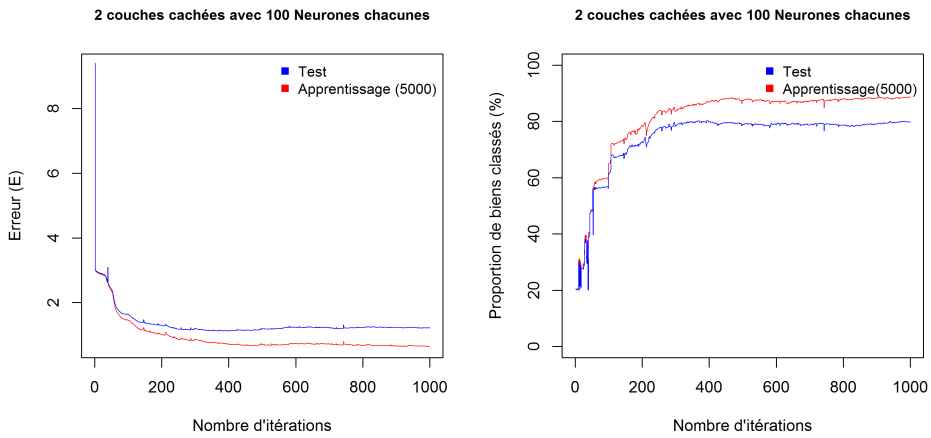


FIGURE 9 – Evolution de la proportion de bien classés dans l'échantillon test (données mnist deux couches cachées de 100 neurones))

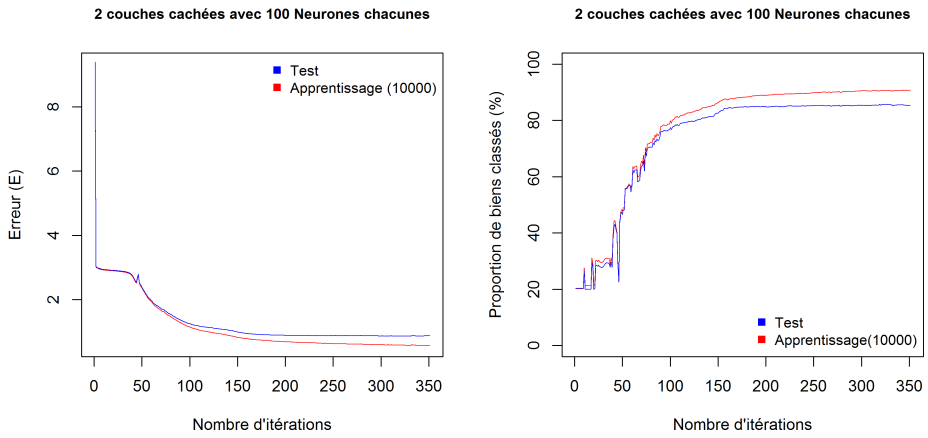


FIGURE 10 – Evolution de la proportion de bien classés dans l'échantillon test (données mnist deux couches cachées de 100 neurones))

4 POSSIBLES DEVELOPPEMENTS

Nous aurions pu, dans le cadre d'un projet plus large, implémenter d'autres aspects des réseaux neuronaux.

Nous aurions pu implémenter une descente de gradient stochastique, qui est plus adaptée pour les grands jeux de données.

L'architecture aurait pu être plus flexible. La couche sigmoid aurait pu être définie dans une classe séparée. Cela aurait permis de créer de nouveaux types de couches plus facilement, comme celles que nous allons présenter.

La régularisation permet de limiter l'overfitting en limitant la taille des poids. Elle est souvent implémentée par une contrainte de type L2 ou L1. La contrainte L1 permet d'obtenir des structures sparses. Le choix du coefficient idéal se fait par cross-validation.

Les réseaux de type convolutionnel sont particulièrement adaptés aux problèmes dimensionnels, comme le son, les images ou les vidéos. Ces résseaux utilisent la topologie de l'espace de départ pour apppliquer un masque local sur les différents zones.

Les dernières générations de réseaux neuronaux utilisent une technique nommée Dropout [3] avec des unités linéaires rectifiées, de la forme $\max(0, x)$. Le Dropout consiste à ne considérer chaque neurone qu'avec une probabilité p. Il serait idéal de prendre p=50% pour des neurones des couches cachées, et p=80% pour les couches d'entrée. Le dropout permet d'éviter la co-adaptation des neurones et de moyenner entre la totalité des sous-réseaux possibles. Dernièrement une extension des unités linéaires rectifiées, nommée Maxout [2] a permis d'améliorer la performance des réseaux, en restant avec des unités pseudo-linéaires de la forme $\max_{j\in[1..k]}(x_j)$.

[1] M Forina, C Armanino, M Castino, and M Ubigli. Multivariate data analysis as a discriminating method of the origin of wines. *Vitis*, 25(3) :189–201, 1986.

[2] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv :1302.4389*, 2013.

[3] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv :1207.0580*, 2012.

[4] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. 1998.

[5] Martin Riedmiller and Heinrich Braun. Rprop - a fast adaptive learning algorithm. Technical report, Proc. of ISCIS VII), Universitat, 1992.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088) :533–536, 1986.

ANNEXE : DOCUMENTATION DOXYGEN

Neural networks

Généré par Doxygen 1.7.6.1

Vendredi Février 21 2014 16 :08 :49

Table des matières

Chapitre 1

Index des classes

1.1 Hiérarchie des classes

Cette liste d'héritage est classée approximativement par ordre alphabétique :

Chapitre 2

Index des classes

2.1 Liste des classes

Liste des classes, structures, unions et interfaces avec une brève description :

Chapitre 3

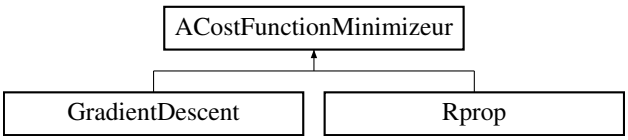
Documentation des classes

3.1 Référence de la classe ACostFunctionMinimizeur

An abstract class for all kind of minimization of cost functions.

`#include <acostfunctionminimizeur.h>`

Graphe d'héritage de ACostFunctionMinimizeur :



Fonctions membres publiques

- void doIteration ()

  doIteration Compute a new iteration of the minimizeur and update the weights.
- const std : :vector< float > getErrors () const

  getErrors Return the history of the errors of the network on the training set.
- virtual float computeError (const Matrix &X, const Matrix &Y) const =0
- virtual MultiLayerPerceptron **getPerceptron** ()=0

Fonctions membres protégées

- virtual float doIteration_ ()=0

  doIteration_ Do an iteration of your optimizeur procedure and return the error. Override it for your own iterative procedure.

Attributs protégés

- std : :shared_ptr< tuple< Matrix, Matrix > > **validationSet_**

Amis

– std : :ostream & **operator**<< (std : :ostream &os, const ACostFunctionMinimizeur &sequence)

3.1.1 Description détaillée

An abstract class for all kind of minimization of cost functions.

3.1.2 Documentation des fonctions membres

3.1.2.1 virtual float ACostFunctionMinimizeur : :computeError (const Matrix & *X*, const Matrix & *Y*) const [pure virtual]

Compute the error using the final network.

Implémenté dans Rprop, et GradientDescent.

La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/acostfunctionminimizeur.h

3.2 Référence de la classe CSVRow

for parsing csv file Create a class representing a row

`#include <CSVRow.h>`

Fonctions membres publiques

– std : :string const & **operator[]** (std : :size_t index) const
– std : :size_t **size** () const
– void **readNextRow** (std : :istream &str)

3.2.1 Description détaillée

for parsing csv file Create a class representing a row

La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/parsercsv/CSVRow.h

3.3 Référence du modèle de la classe DimensionalException< T >

An exception throwed when the dimensions of the objects are not compatible.

`#include <dimensionalexception.h>`

Fonctions membres publiques

– **DimensionalException** (T expected, T found, std : :string dimension) throw ()
– virtual const char ∗ **what** () const throw ()

3.3.1 Description détaillée

template<class T>class DimensionalException< T >

An exception throwed when the dimensions of the objects are not compatible.

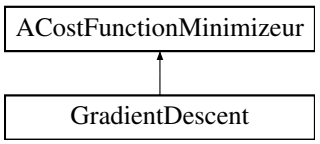
La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/utils/dimensionalexception.h

3.4 Référence de la classe GradientDescent

Plain, normal, gradient descent. Do just that but do it well.

`#include <gradientdescent.h>`

Graphe d'héritage de GradientDescent :



Fonctions membres publiques

– GradientDescent (const Matrix &X, const Matrix &Y, const vector< int > &dimensions, float learningRate)
  GradientDescent Plain, normal, gradient descent. Do just that but do it well.
– float computeError (const Matrix &Xval, const Matrix &Yval) const
– virtual MultiLayerPerceptron **getPerceptron** ()

3.4.1 Description détaillée

Plain, normal, gradient descent. Do just that but do it well.

3.4.2 Documentation des constructeurs et destructeur

3.4.2.1 GradientDescent : :GradientDescent (const Matrix & *X,* const Matrix & *Y,* const vector< int > & *dimensions,* float *learningRate*) [inline]

GradientDescent Plain, normal, gradient descent. Do just that but do it well.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and as many columns as the last layer of the network.
dimensions	The dimensions of the perceptron. The first item is the size of the input, the last one the size of the output.
learningRate	The global learning rate of the descent of the gradient.

3.4.3 Documentation des fonctions membres

3.4.3.1 float GradientDescent : :computeError (const Matrix & *X*, const Matrix & *Y*) const [inline, virtual]

Compute the error using the final network.

Implémente ACostFunctionMinimizeur.

La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/gradientdescent/gradientdescent.h

3.5 Référence de la classe MatrixSequence

This class allows you to manage a sequence of hetegoreneous matrix, and to switch from the matrix view to the array view, where the array is a flatten version of the sequence.

`#include <matrixsequence.h>`

Fonctions membres publiques

– MatrixSequence (vector< tuple< int, int > > shapes)
  MatrixSequence Create a sequence of matrix.
– Map< Matrix > matrix (unsigned int i)
  matrix Get a reference to the matrix i. This operation doesn't construct a new matrix.
– const Map< const Matrix > **matrix** (unsigned int i) const
– Map< ArrayX > data ()
  data Get a reference to the flatten version of the sequence. This operation doesn't construct a new array, so you can modify the object by modifying the Map<ArrayX>.
– const Map< const ArrayX > **data** () const
– unsigned int size ()
  getNbrMatrix Get the number of elements in the sequence.
– unsigned int **size** () const
– Map< Matrix > last ()
  last Get the last matrix of the sequence.
– const Map< const Matrix > **last** () const

Amis

– std : :ostream & **operator**<< (std : :ostream &outputStream, const MatrixSequence &sequence)

3.5.1 Description détaillée

This class allows you to manage a sequence of hetegoreneous matrix, and to switch from the matrix view to the array view, where the array is a flatten version of the sequence.

3.5.2 Documentation des constructeurs et destructeur

3.5.2.1 MatrixSequence : :MatrixSequence (vector< tuple< int, int > > *shapes*) [inline]

MatrixSequence Create a sequence of matrix.

Paramètres

shapes	The tuple of shapes of the matrix. Each matrix can have a different shape.

3.5.3 Documentation des fonctions membres

3.5.3.1 Map<ArrayX> MatrixSequence : :data ( ) [inline]

data Get a reference to the flatten version of the sequence. This operation doesn't construct a new array, so you can modify the object by modifying the Map<ArrayX>.

Renvoie

A reference to the array. WARNING : it is your responsability to check the shape is right for an assignement. WARNING : if you want to assign some data to the array, you either have to directly do obj.array() = somearray, or to store obj.array() in a Map<ArrayXd> object or to use auto. DO NOT STORE obj.array() IN AN ARRAY FOR ASSIGNEMENT !

3.5.3.2 Map<Matrix> MatrixSequence : :last ( ) [inline]

last Get the last matrix of the sequence.

Renvoie

3.5.3.3 Map<Matrix> MatrixSequence : :matrix (unsigned int *i*) [inline]

matrix Get a reference to the matrix i. This operation doesn't construct a new matrix.

Paramètres

i	The index of the matrix to get.

Renvoie

A reference to the matrix. WARNING : it is your responsability to check the shape is right for an assignement. WARNING : if you want to assign some data to the matrix, you either have to directly do obj.matrix() = somematrix, or to store obj.matrix() in a Map<Matrix> object or to use auto. DO NOT STORE obj.matrix() IN A MATRIX FOR ASSIGNEMENT !

3.5.3.4 unsigned int MatrixSequence : :size ( ) [inline]

getNbrMatrix Get the number of elements in the sequence.

Renvoie

La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/utils/matrixsequence.h

3.6 Référence de la classe MultiLayerPerceptron

A logistic perceptron with a configurable number of layers.

`#include <multilayerperceptron.h>`

Fonctions membres publiques

– MultiLayerPerceptron (const vector< int > &dimensions_layers)
  MultiLayerPerceptron Create a new logistic perceptron with a configurable number of hidden layers.
– Matrix compute (const Matrix &X) const
  compute Calculate the expected output of the network.
– double computeError (const Matrix &X, const Matrix &Y) const
  compute_output_layers Compute the normalized error term of the network, ie the inverse of the likelihood.
– int computeClassificationScoreBinary (const Matrix &X, const Matrix &Y) const
  computeClassificationScoreBinary Compute the number of correct predicted observations for a binary variable.
– int computeClassificationScoreMulti (const Matrix &X, const Matrix &Y) const
  computeClassificationScoreMulti Compute the number of correct predicted observations for a multi classes variable. It uses one versus all probabilities.
– tuple< double, ArrayX > computeGradient (const Matrix &X, const Matrix &Y) const

computeGradient Compute the error term and the gradient of the network, compared to the expected value.

– MatrixSequence & getWeights ()
getWeights Give the current weights used by the network.
– const MatrixSequence & **getWeights** () const

3.6.1 Description détaillée

A logistic perceptron with a configurable number of layers.

3.6.2 Documentation des constructeurs et destructeur

3.6.2.1 MultiLayerPerceptron : :MultiLayerPerceptron (const vector< int > & *dimensions_layers*) [inline]

MultiLayerPerceptron Create a new logistic perceptron with a configurable number of hidden layers.

Paramètres

dimensions	The dimension of the network. The first item is the size of the input and the last one is the size of the output. The other ones are the hidden layers.

3.6.3 Documentation des fonctions membres

3.6.3.1 Matrix MultiLayerPerceptron : :compute (const Matrix & *X*) const [inline]

compute Calculate the expected output of the network.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.

3.6.3.2 int MultiLayerPerceptron : :computeClassificationScoreBinary (const Matrix & *X*, const Matrix & *Y*) const [inline]

computeClassificationScoreBinary Compute the number of correct predicted observations for a binary variable.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and one column

3.6.3.3 int MultiLayerPerceptron : :computeClassificationScoreMulti (const Matrix & *X,* const Matrix & *Y*) const [inline]

computeClassificationScoreMulti Compute the number of correct predicted observations for a multi classes variable. It uses one versus all probabilities.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and as many columns as the last layer of the network.

3.6.3.4 double MultiLayerPerceptron : :computeError (const Matrix & *X,* const Matrix & *Y*) const [inline]

compute_output_layers Compute the normalized error term of the network, ie the inverse of the likelihood.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and as many columns as the last layer of the network.

Renvoie

The mean error term.

3.6.3.5 tuple<double, ArrayX> MultiLayerPerceptron : :computeGradient (const Matrix & *X,* const Matrix & *Y*) const [inline]

computeGradient Compute the error term and the gradient of the network, compared to the expected value.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and as many columns as the last layer of the network.

Renvoie

The first term of the tuple is the mean error term. The secund argument is the gradient of the error term.

3.6.3.6 MatrixSequence& MultiLayerPerceptron : :getWeights ( ) [inline]

getWeights Give the current weights used by the network.

Renvoie

A reference to the weights.

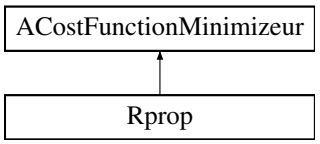
La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/neuralnets/multilayerperceptron.h

3.7 Référence de la classe Rprop

Use the RProp algorithm for gradient descent. See http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.1417 for more informations about the algorithm.

`#include <rprop.h>`

Graphe d'héritage de Rprop :



Fonctions membres publiques

– Rprop (const Matrix &X, const Matrix &Y, const vector< int > &dimensions, float maxLearningRate, float initialLearningRate)
Rprop Use the RProp algorithm for gradient descent.
– float computeError (const Matrix &Xval, const Matrix &Yval) const
– virtual MultiLayerPerceptron **getPerceptron** ()

Attributs publics

– Matrix X_
X_ The input of the stored data.
– Matrix Y_
Y_ The output of the stored data.
– ArrayX lastGradient
lastGradient The gradient of the last iteration. Used for sign comparison.
– ArrayX **individualRates**
– float **maxLearningRate_**
– MultiLayerPerceptron **perceptron_**

3.7.1 Description détaillée

Use the RProp algorithm for gradient descent. See http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.1417 for more informations about the algorithm.

3.7.2 Documentation des constructeurs et destructeur

3.7.2.1 Rprop : :Rprop (const Matrix & *X,* const Matrix & *Y,* const vector< int > & *dimensions,* float *maxLearningRate,* float *initialLearningRate*) [inline]

Rprop Use the RProp algorithm for gradient descent.

Paramètres

X	The input of the network. It must have one row per observation and as many columns as the first layer of the network.
Y	The expected output. It must have as many rows as X and as many columns as the last layer of the network.
dimensions	The dimensions of the perceptron. The first item is the size of the input, the last one the size of the output.
maxLearningRate	The maximum learning rate of the descent of the individual weiths' gradient.
initialLearningRate	The initial learning rate of the descent.

3.7.3 Documentation des fonctions membres

3.7.3.1 float Rprop : :computeError (const Matrix & *X,* const Matrix & *Y*) const [inline, virtual]

Compute the error using the final network.

Implémente ACostFunctionMinimizeur.

La documentation de cette classe a été générée à partir du fichier suivant :
– src/app/gradientdescent/rprop.h