

Géométrie de l'information et apprentissage distribué

Clément Dell'Aiera, Clément Prévosteau, David Wahiche

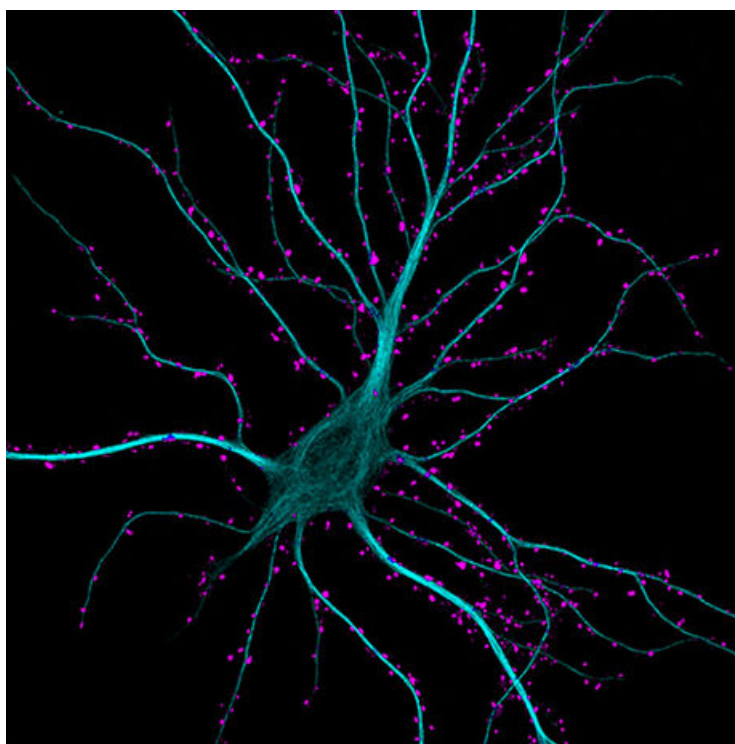


FIGURE 1 – Neurone grossi 63 fois, image de Kieran Boyle

Table des matières

1	Point de vue géométrique en statistique	3
1.1	Géométrie de l'information	3
1.2	Algorithme de descente de gradient naturel	4
2	Réseau de neurones	5
2.1	Présentation	5
2.2	Neurones binaires et algorithme du "perceptron convergence pro- cedure"	7
2.3	Neurones linéaires ou filtres linéaires	7
2.3.1	La surface d'erreur du neurone linéaire	8
2.3.2	Pourquoi plusieurs couches ?	10
2.3.3	Expériences autour du Xor	11
2.4	Neurones logistiques	15
2.5	Algorithme de rétropropagation	15
2.6	16
2.7	Point de vue géométrique	17
2.8	Machines de Boltzmann	17
2.8.1	Introduction	17
2.8.2	Dynamique stochastique d'une machine de Boltzmann	17
2.8.3	Learning dans les machines de Boltzmann	19
2.9	Restricted Boltzmann machines	19
2.9.1	Introduction	19
2.9.2	Deep Learning à l'aide de RBM	20
2.9.3	Précisions sur l'algorithme de Contrastive divergence (noté CD-1)	20
2.10	Multilayer Neural Network	21
2.11	Cas du perceptron	22
3	Application à la reconnaissance de caractères manuscrits	23
3.1	Rappel sur les ondelettes	23
3.2	Description de l'algorithme	23
4	Deep Learning vs Support Vector Machine	24
5	Appendice	24
5.1	Géométrie différentielle	24

1 Point de vue géométrique en statistique

Parmi les premières méthodes d'apprentissage figurent celles utilisant des réseaux neuronaux.

Le but de ce travail est d'étudier ce réseau et d'appliquer les méthodes de géométrie de l'information pour peut-être mieux comprendre les raisons de leur efficacité.

1.1 Géométrie de l'information

La géométrie de l'information propose d'utiliser les méthodes de la géométrie différentielle en statistique afin d'optimiser les algorithmes. Pour le lecteur apeuré par les gros mots mathématiques, qu'il se rassure : il n'est pas question ici d'étudier les propriétés géométriques fines des objets statistiques, mais plutôt d'appliquer des outils à peine plus sophistiqués que ceux d'un cours de calcul différentiel classique dans une optique d'optimisation.

L'idée première est de donner une structure de variété différentielle au modèle statistique étudié : si $\mathcal{S} = \{p_\theta\}_\theta$ est notre honnête famille paramétrique de lois de probabilité, on peut voir $\theta \in \Theta$ comme une coordonnées, ou, pour parler le langage des géomètres, l'application :

$$p_\theta \mapsto \theta$$

fournit une carte locale. Petite remarque : cette application n'est définie que si le modèle est identifiable. On supposera d'ailleurs que le modèle a toutes les propriétés que l'on voudrait.

La deuxième étape est de donner une structure de variété riemannienne au modèle \mathcal{S} . Sans rentrer dans les détails, l'important est de savoir que l'on peut définir une métrique grâce à la log-vraisemblance l du modèle par :

$$g_{ij}(\theta) = \mathbb{E}_\theta[\partial_i l(X, \theta) \partial_j l(X, \theta)]$$

En pratique, cela signifie que l'on peut mesurer des distances et des angles sur l'espace tangent à \mathcal{S} :

$$\forall w_1, w_2 \in T_\theta \mathcal{S}, \langle w_1, w_2 \rangle = w_1^T G w_2$$

où $G(\theta) = (g_{i,j}(\theta))_{i,j}$.

Donnons un exemple important : celui des familles exponentielles. Le modèle est donné par :

$$\mathcal{Exp}(C, F, \psi) = \{p(x, \theta) = \exp[C(x) + \sum_{i=1}^n \theta^i F_i(x) - \psi(\theta)] \quad : \quad \theta \in \mathbb{R}^n\}$$

où $F = (F_1, \dots, F_n)$ est une famille linéairement indépendante de fonctions C^∞ .
Un simple calcul donne :

$$\partial_i \partial_j l(x, \theta) = -\partial_i \partial_j \psi(\theta)$$

Donc :

$$g_{ij}(\theta) = \mathbb{E}[\partial_i l(x, \theta) \partial_j l(x, \theta)] = -\mathbb{E}[\partial_i \partial_j l(x, \theta)] = \partial_i \partial_j \psi(\theta)$$

La matrice G est donc la hessienne de ψ et on a calculé la métrique riemannienne :

$$ds^2 = \sum_{i,j} \partial_i \partial_j \psi \, dx^i dx^j$$

1.2 Algorithme de descente de gradient naturel

Amari décrit dans ses différents articles une méthode de descente de gradient adapté au cadre riemannien : on va corriger le pas par un facteur qui va se révéler être l'inverse de la matrice de Fisher. **idée intuitive avec les ellipses, peut-être une illustration ?**

On se donne une fonction de perte $L : \mathcal{S} \rightarrow \mathbb{R}$ que l'on évalue en w , et on veut minimiser $L(w + dw)$ où la norme $|dw| = \epsilon$ est fixée.

Théorème 1. *La direction optimale de descente dw^* est donnée par :*

$$dw^* = -\hat{\nabla} L(w) = -G^{-1}(w) \nabla L(w)$$

Donnons un sketch de preuve. Posons $dw = \epsilon a$, le problème de minimisation devient alors :

$$\min L(w + \epsilon a) \quad s.c. \quad |a|^2 = \sum_{i,j} g_{ij}(w) a_i a_j = 1$$

Mais $L(w + \epsilon a) - L(w) - \epsilon (\nabla L(w))^T a$ est un petit o de ϵ et minimiser $(\nabla L(w))^T a$ sous la contrainte $a^T G a = 1$ est simple avec les conditions de Lagrange qui donnent :

$$\frac{\partial}{\partial a_j} ((\nabla L(w))^T a - \lambda a^T G a) = 0$$

Finalement : $\nabla L(w) = 2\lambda G a$ et donc

$$a = \frac{1}{2\lambda} G^{-1} \nabla L(w)$$

□

L'algorithme de pas d'apprentissage ϵ_n qui en découle est décrit par la formule de récurrence suivante :

$$w_{n+1} = w_n - \epsilon_n \hat{\nabla} L(w_n)$$

2 Réseau de neurones

2.1 Présentation

Les réseaux de neurones ont initialement été développés comme des algorithmes, initialement dans le domaine de *l'intelligence artificielle*, ce qui en fait naturellement une discipline du *machine learning*. L'idée première était de s'inspirer du fonctionnement du cerveau humain pour créer des algorithmes adaptatifs. Bien que ce rôle fondateur de modèle pour le cerveau humain n'ait pas été un franc succès, les réseaux neuronaux ont donné des modèles statistiques intéressants. Ce changement d'interprétation est reflété dans le nom de réseau de neurones *artificiels* que certains auteurs utilisent désormais.

L'apprentissage est classiquement divisé en plusieurs domaines :

- apprentissage supervisé : apprendre à prévoir un output lorsque l'on donne un input, divisé en 2 catégories :
 - régression
 - classification
- Non-supervisé : découvrir une bonne représentation de l'input.
- Apprentissage par renforcement : sélection des actions en maximisant le gain.

Les réseaux de neurones font partie de l'apprentissage supervisé : on se donne une liste d'exemple, le *training set*, et l'on se sert de la différence entre la réponse attendue et la réponse donnée pour corriger l'algorithme.

Historiquement, on peut faire remonter l'origine des réseaux de neurones à McCulloch et Pitts en 1943 : ils proposent de modéliser formellement les neurones par des unités traitant un signal d'entrée, renvoyant en sortie un signal binaire correspondant à 1 si le signal d'entrée dépasse un certain seuil, ce qu'ils nomment un *binary threshold neurons* ou des unités de décisions.

Le perceptron constitue la première génération de réseaux neuronaux au sens où on l'entend aujourd'hui, popularisés par Fran Rosenblatt dans les années 60 et dont on trouvera un très bon exposé dans son livre *Principles of Neurodynamics*.

Le lecteur pourra aussi trouver dans *Perceptrons* de Minsky et Papert (1969) de quoi satisfaire sa curiosité. Ces deux auteurs sont à l'origine d'un résultat très négatif pour les perceptrons : le théorème d'invariance sous l'action d'un groupe ("Group invariance theorem"), qui assure que les perceptrons ne peuvent pas apprendre des configurations invariantes sous l'action d'un groupe de transformations. Ce résultat est assez limitant puisqu'il empêche, par exemple, le perceptron d'apprendre des configurations de pixels qui sont invariantes par translations, typiquement ce qu'on aimerait lui faire faire. C'est là que les réseaux multicouches permettront de sauver la mise, comme nous le verrons.

Un réseau de neurones peut se formaliser de la façon suivante : plusieurs couches de neurones (que l'on imagine les unes superposées aux autres), chaque couche recevant des signaux de la couche précédentes. Chaque neurone N_i est en fait un opérateur, qui agit sur les signaux d'entrées qu'il reçoit y_j : il leur affecte chacun un poids w_{ij} et ensuite une fonction, signal qu'il transmet à son tour :

$$y_i = f(b + \sum_{j \rightarrow i} w_{ij} y_j) = f(w^T \cdot x)$$

où l'on prend w le vecteur poids associé au neurone N_i .

Voici une liste non-exhaustive de différents réseaux de neurones, que l'on présente en fonction de leur fonction de réponse :

- binaire : $y = 1_{b+w^T x > 0}$
- linéaire : $y = b + w^T x$
- linéaire rectifié : $y = \max(0, b + w^T x)$
- sigmoïde : $y = \frac{1}{1+e^{-z}}$
- binaire stochastique : $\mathbb{P}(s = 1) = \frac{1}{1+e^{-z}}$ L'output est traité comme le paramètre de Poisson.

Ainsi que plusieurs types d'architectures que l'on peut leur imposer, c'est-à-dire comment les neurones sont reliés entre eux :

- Une couche ou plusieurs (deep neural networks)
- Récurents : des cycles sont possibles.
- Complets.
- Toute structure imaginable sur un graphe ! En fait, le modélisateur n'est pas obligé d'adopter une structure en couches.

Ces réseaux peuvent se reformuler en termes de modèles statistiques, comme Herbert K.H. Lee l'expose dans *Bayesian Non Parametrics via Neural Networks*. Rappelons qu'un réseau est composé de plusieurs couches : la couche d'entrée, les couches cachées et la couche de sortie. Chaque noeud du réseau applique une fonction ψ , en général une sigmoïde, à une combinaison linéaire des signaux d'entrée. Si y est le signal de sortie, x l'entrée, le modèle statistique que représente le réseau à m couches cachées prend souvent la forme :

$$y = \beta_0 + \sum_{i=1}^m \beta_i \psi(w_i^T x_i) + \eta_i$$

$$\eta_i \sim \mathcal{N}(0, \sigma^2)$$

Cet équation montre qu'un réseau neuronal peut s'interpréter comme un modèle de régression non paramétrique sur une base donnée. Par exemple, si

$$\psi(x) = \frac{1}{1 + \exp(-x)}$$

est la fonction logistique, on sait que l'espace engendré par les translatés-échelonnés de cette fonction est tout l'espace des fonctions de carré intégrable.

Détaillons un peu différents modèles simples de RN.

2.2 Neurones binaires et algorithme du "perceptron convergence procedure"

Ces neurones ont une fonction de réponse de type indicatrice :

$$y = 1_{\{w^T x \geq 0\}}$$

Voici comment entraîner les neurones binaires comme classifiants :

1. Incorporer une composante 1 en plus au vecteur input, pour incorporer le biais dans les poids.
2. Choisir un exemple d'entraînement. La procédure de choix doit assurer que tous les exemples seront choisis.
3. Si l'output est correct, ne pas changer les poids. Sinon soustraire l'input du vecteur des poids.

On peut interpréter géométriquement cette procédure.

L'ensemble des poids est vu comme un \mathbb{R} -espace vectoriel de dimension le nombre de poids. Un input définit un hyperplan (son orthogonal), et un vecteur poids donne la bonne réponse par rapport à cet input ssi il est du "bon côté" du plan. Si on a deux inputs, le sous-ensemble des bons poids (qui répondent correctement aux deux inputs) forment alors un cône. On remarque que les bons poids forment un ensemble convexe : le problème est convexe.

Preuve :

On suppose qu'il existe un vecteur $w^* \in C$ qui donne la bonne réponse à tous les exemples.

$$|w_{n+1} - w^*|^2 = |w_n - x|^2 = |w_{n+1}|^2 + |x|^2 + 2\langle w_n, x \rangle + |w^*|^2 + 2\langle w^*, w_n - x \rangle \leq |x|^2$$

2.3 Neurones linéaires ou filtres linéaires

On appelle neurones linéaires des neurones dont la fonction de réponse est linéaire :

$$y = w^T x$$

Ici l'algorithme, plutôt que d'approcher de mieux en mieux le poids idéal, va optimiser la distance entre l'output et la cible, ce qui est une différence notable par rapport au cas précédent. La procédure des neurones binaires (perceptron) ne peut pas se généraliser à plusieurs couches et nous prévenons le lecteur : leur nom de "perceptron multicouches" utilisé à propos des neurones linéaires est un

faux ami !

Voici l'algorithme :

Choisir un pas d'apprentissage ϵ .

1. On entraîne en donnant la cible t .
2. Actualiser les poids selon :

$$\Delta w_i = \epsilon x_i(t - y)$$

On comprend que cet algorithme n'est rien d'autre qu'une descente de gradient. En effet, si on dérive l'erreur quadratique :

$$E = \frac{1}{2} \sum_{n \in \text{training}} (t^n - y^n)^2$$

on obtient : $\frac{\partial E}{\partial w_j} = -\sum x_i^n(t^n - y^n)$. Cette règle d'actualisation des poids (*batch delta-rule*) modifie les poids proportionnellement à cette dérivée afin de se rapprocher du minimum de la surface d'erreur :

$$\Delta w_i = -\epsilon \frac{\partial E}{\partial w_j}$$

2.3.1 La surface d'erreur du neurone linéaire

On se place dans l'espace des poids, augmenté d'une dimension pour l'erreur. On s'intéresse donc au graphe de l'erreur :

$$\mathcal{S} = \{(w_1, \dots, w_n, E)\}$$

Dans le cas d'un RN linéaire avec erreur quadratique, c'est un paraboloïde elliptique. L'algorithme précédent effectue une descente le long de ce paraboloïde, perpendiculairement aux lignes de niveau (qui sont des ellipses). Cela explique que l'apprentissage puisse être très lent : si l'ellipse est très aplatie, le gradient peu pointer dans une direction très peu colinéaire au minimum recherché. (d'où la correction avec le gradient riemanien)

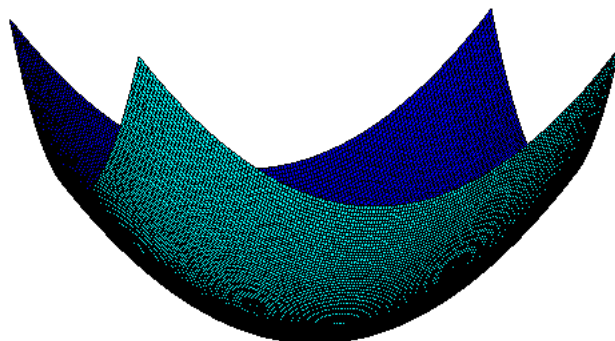


FIGURE 2 – La forme typique d’une surface d’erreur quadratique.

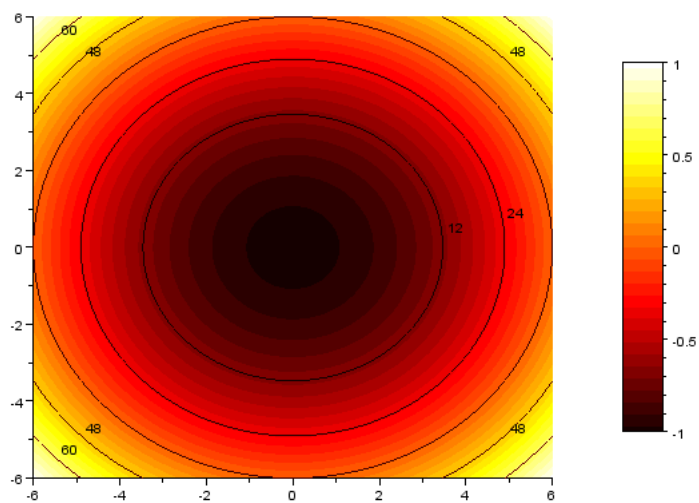
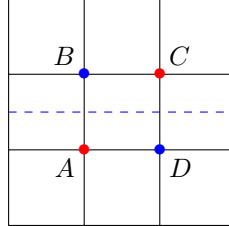


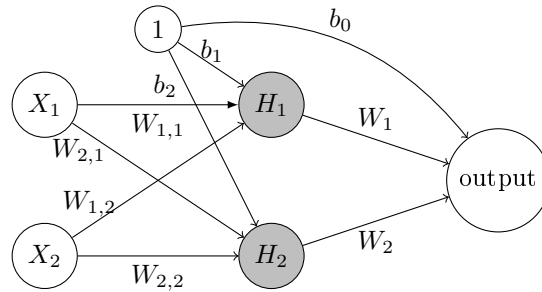
FIGURE 3 – Les lignes de niveau d’une surface d’erreur quadratique sont bien des ellipses.

2.3.2 Pourquoi plusieurs couches ?

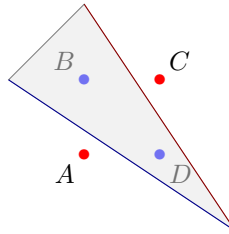
Ici considérons le cas particulier du XOR dessiné ci-dessous. $ABCD$ est un carré des les sommets diagonalement opposés sont de même couleur. On cherche à prédire leur couleur. Il y a deux couleurs, ce problème peut donc se rapporter à un problème de prédiction de vecteurs binaires.



Dans ce cas-là, une réseau à une seule couche nous donne un classifieur linéaire qui ne permet pas de séparer les points de couleurs différentes. Ainsi en rajoutant une couche intermédiaire, qui permet d'obtenir deux classifieurs linéaires et leur intersection permet de reconnaître le XOR.



Ainsi pour le premier neurone, on pose $u_1 = W_{1,1}x_1 + W_{1,2}x_2 + b_1$ et $u_2 = W_{2,1}x_1 + W_{2,2}x_2 + b_2$. On effectue la somme logique et on obtient par exemple la séparation de l'espace suivante :



La zone grise obtenue correspond à l'intersection des deux séparateurs linéaires. On voit qu'on peut bien reconnaître le xor qui permet bien de séparer les points de couleur différente. Le théorème de l'approximation universelle pour les réseaux neuronaux va plus loin : avec des fonctions d'activation sigmoïdales, on peut approximer pour n'importe quelle constante arbitraire une fonction continue sur un sous-ensemble compact de \mathbb{R}^n .

Mathématiquement parlant, cela nous donne :

Théorème 2. Soit ϕ une fonction non constante, bornée et de limites finies en $+\infty$ et $-\infty$ différentes. Soit $X \subseteq \mathbb{R}^m$ et X compact. Alors :

$\forall f \in \mathcal{C}(X), \forall \epsilon > 0; \exists n \in \mathbb{N}, (a_{i,j}) \in \mathbb{R}^n \times \mathbb{R}^m, (b_i) \in \mathbb{R}^n, (w_i) \in \mathbb{R}^n : ||f - A_n f|| < \epsilon$

$$\text{avec } (A_n f)(x_1, \dots, x_m) = \sum_{i=1}^n w_i \phi \left(\sum_{j=1}^m a_{i,j} x_j + b_i \right)$$

2.3.3 Expériences autour du Xor

On souhaite vérifier si on ne surapprend pas les couleurs des points du Xor. Soient $A(-1, 1)$, $B(1, -1)$ des points de couleur rouge et $C(-1, -1)$ et $D(1, 1)$ de couleur bleue. On bruite aléatoirement ces points de telle sorte à ce que chaque point du tirage associé à un point demeure dans le quart de plan de ce point. On obtient ainsi la figure suivante.

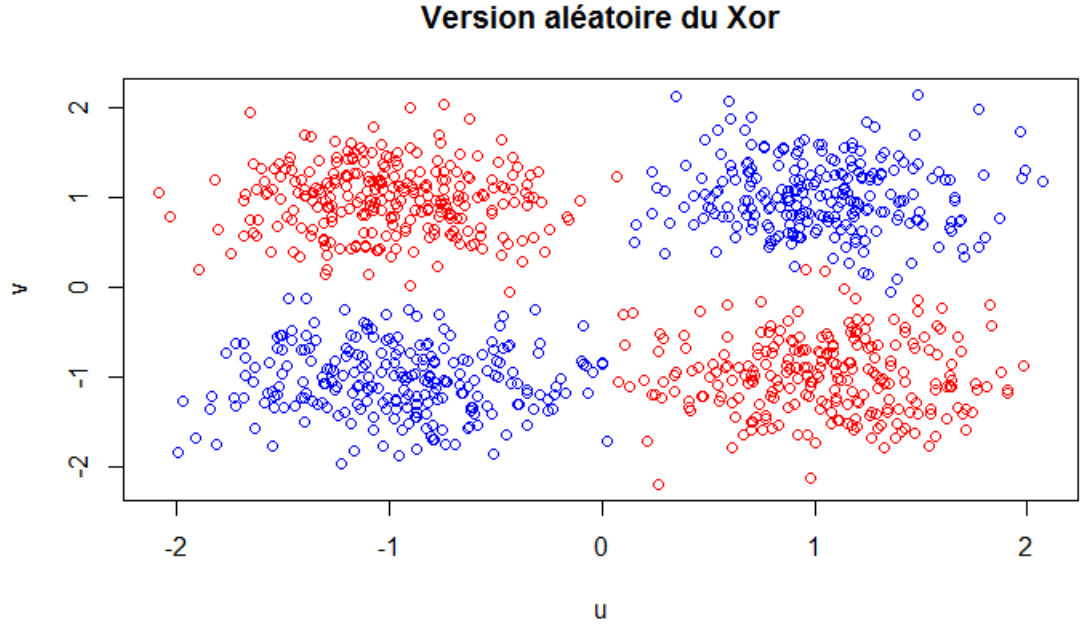


FIGURE 4 – Représentation d'un tirage de xor aléatoire

Une **SVM**, ou machine à vecteurs de support constituent une généralisation des classifieurs linéaires. Ils ont été développés dans les 1990 à partir des théories de Vladimir Vapnik.

Ils permettent de traiter des problèmes de discrimination non-linéaire, et de reformuler le problème de classement comme un problème d'optimisation quadratique. Leur fonctionnement repose sur deux idées :

- la recherche de la *marge maximale* : la marge est la distance entre la frontière (linéaire) de séparation et les échantillons les plus proches qu'on appelle *vecteur supports*. Dans les SVM, on cherche à maximiser la marge
- le problème principal des SVM si laissés comme tels est qu'ils ne permettent pas de reconnaître des formes non linéairement séparables. L'idée utilisée pour remédier à ce genre de problèmes est d'augmenter potentiellement la dimension de l'espace dans lequel on cherche les hyperplan séparateurs en appliquant par exemple une transformation non-linéaire ϕ aux vecteurs d'entrée x . Ainsi dans le cas d'un cercle non identifiable en coordonnées cartésiennes, le passage en coordonnées polaires permet de répondre à la question. On appelle ces transformations des *noyaux* qui doivent vérifier certaines conditions et ces techniques sont appelées *kernel trick*.

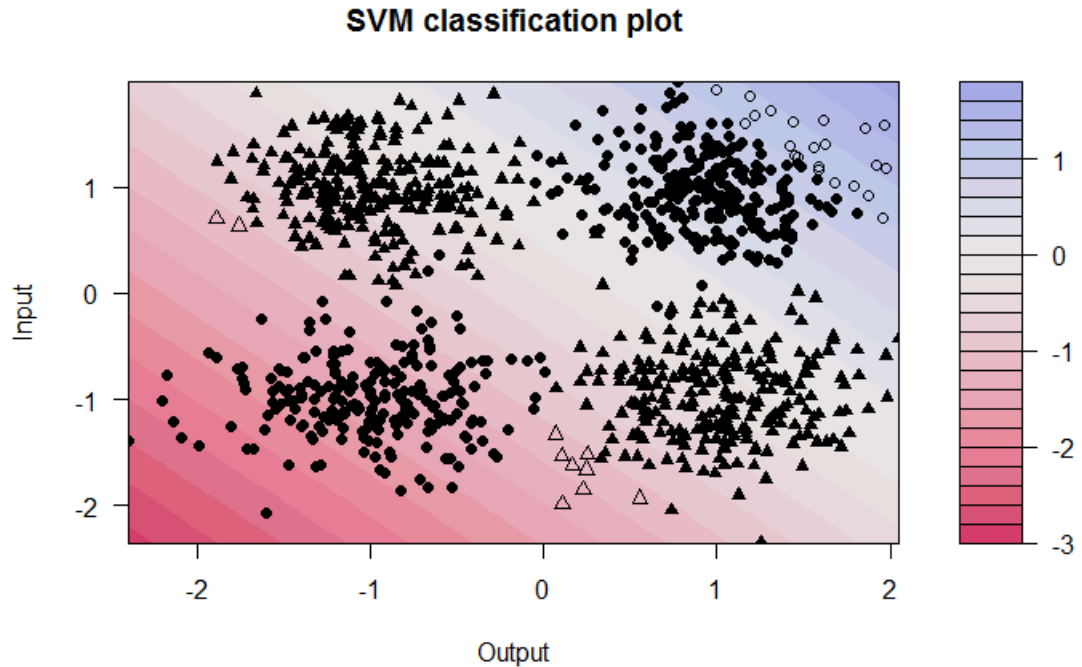


FIGURE 5 – Reconnaissance par un svm **linéaire** du xor

On voit sur la figure ci-dessus que les points noircis correspondent aux erreurs obtenues avec un SVM linéaire. Les résultats obtenus sont mauvais car les points du XOR ne sont pas linéairement séparables.

En revanche, en ayant recours au kernel trick, avec un noyau polynomial de degré 2, les résultats sont bien meilleurs. Ils sont de la forme $K(x, y) = (x^T y + c)^2$ où x et y sont les vecteur d'entrée et $c \geq 0$ une constante.

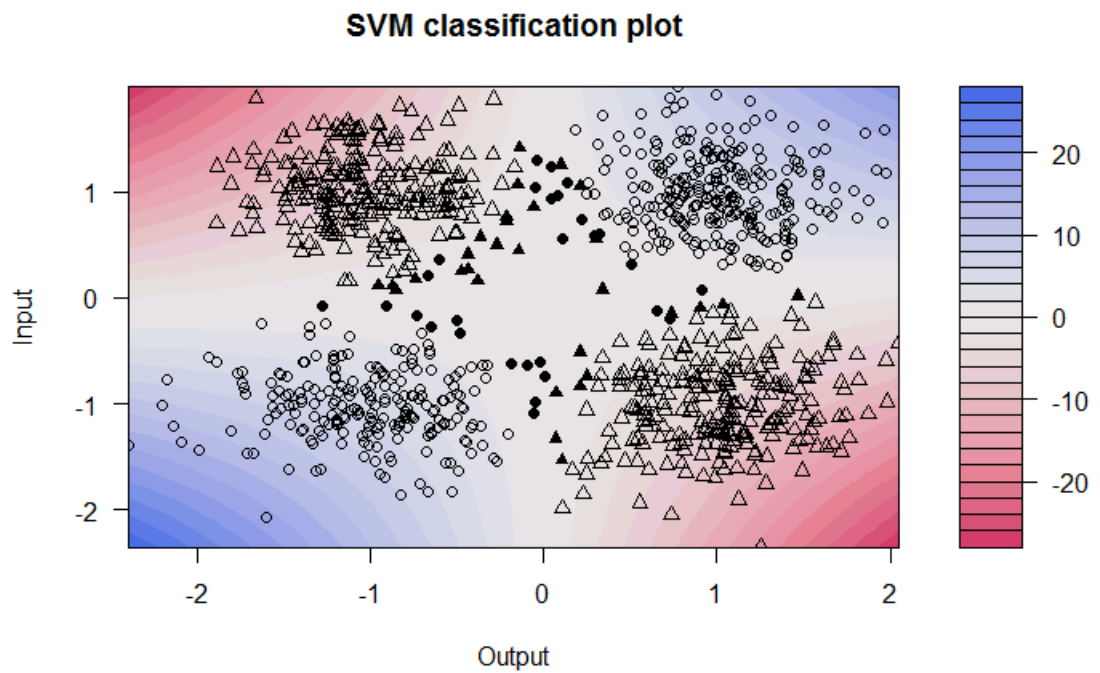


FIGURE 6 – Reconnaissance par un svm avec un noyau de degré 2 du xor

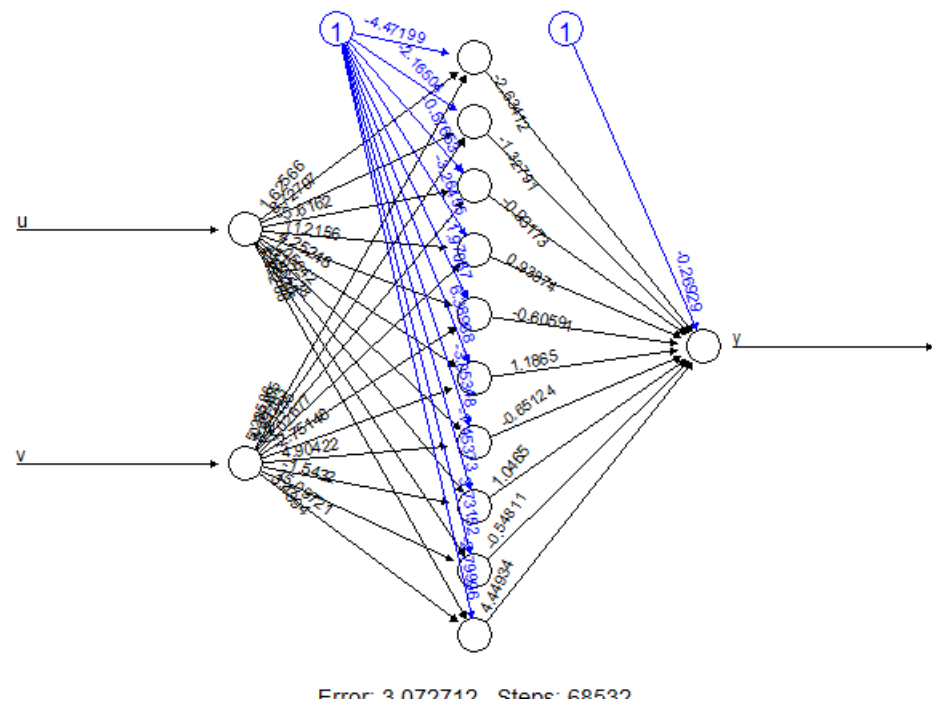


FIGURE 7 – Reconnaissance par un réseau neuronal du xor

2.4 Neurones logistiques

On va généraliser la procédure précédente aux réseaux de neurones logistiques, ceux qui ont une fonction de réponse du type :

$$z = b + w^T x$$

$$y = \frac{1}{1 + e^{-z}}$$

On remarque que l'image de la fonction logistique est l'intervalle $(0; 1)$, ce qui nous permettra d'interpréter sa valeur comme la probabilité de l'output sachant l'input, dans le cas de réseaux de neurones stochastiques.

$$\frac{\partial y}{\partial w_i} = x_i y(1 - y)$$

$$\frac{\partial E}{\partial w_j} = - \sum_n x_i^n (1 - y^n) y^n (t^n - y^n)$$

2.5 Algorithme de rétropropagation

Maintenant que l'on sait modifier les poids pour une seule couche, on va donner un moyen de modifier les poids dans le cas de couches cachées. Cette généralisation n'est pas innocente : les réseaux sans couche cachée ne sont pas très adaptables, au sens où leur modélisation est beaucoup plus limitée que celle des réseaux multicouches. L'idée de la rétropropagation est de remonter les dérivées des erreurs par rapport au output récursivement à travers les couches cachées, de la couche des outputs à celle des inputs. On suppose bien sûr que l'erreur est dérivable, rendant cet algorithme inutile pour un réseau binaire par exemple.

Soit donc un neurone N_j situé dans une couche cachée. Il reçoit un input total

$$z_j = \sum_{i \rightarrow j} w_{ij} y_i.$$

On cherche alors à calculer l'erreur $\frac{\partial E}{\partial y_i}$ pour tous les neurones qui lui sont connectés dans la couche précédente, *ie* les neurones N_i tels que $i \rightarrow j$. Mais une simple application de la règle de la chaîne donne une expression simple pour ce terme :

$$\begin{aligned} \frac{\partial E}{\partial z_j} &= \frac{\partial y_j}{\partial z_j} \frac{\partial E}{\partial y_j} = y_j(1 - y_j) \frac{\partial E}{\partial y_j} \\ \frac{\partial E}{\partial y_i} &= \sum_j \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j} \end{aligned}$$

Donc :

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

On peut alors calculer la variation de l'erreur en fonction d'une variation des poids :

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}.$$

Bien que cette procédure nous guide pour adapter les poids, plusieurs questions se posent en pratique, par exemple : à quelle fréquence doit on mettre à jour nos poids ? Comment choisir la vitesse d'apprentissage ? On peut adapter les poids à chaque exemple (*online*), après tout le stock d'exemple (*full batch learning*), ou bien encore après un petit nombre (*mini batch*).

2.6

Softmax : forcer les outputs à former une probabilité Problème de l'erreur quadratique : mauvais avec une fonction logistique, les dérivées tendent à "plateau out" lorsque l'output y est proche de 0 ou 1 :

$$\frac{dE}{dz} = (t - y)y(1 - y)$$

Une erreur plus convenable est la cross entropy function :

$$E = -t \log(y) - (1 - t) \log(1 - y)$$

car alors

$$\frac{dE}{dz} = y - t$$

Pour un groupe de neurones N_i qui reçoivent en entrée z_i appelée *logit*, l'output est donné par

$$y_i = \frac{e^{z_i}}{\sum_{j \in \text{group}} e^{z_j}}$$

$$\frac{dy_i}{dz_i} = y_i(1 - y_i)$$

Cross entropy : la bonne fonction de coût

$$C = - \sum_j t_j \log(y_j) \quad t_j \text{ target value}$$

2.7 Point de vue géométrique

Nous présentons dans cette section les idées de différents articles d'Amari, ainsi que de son livre *Methods of information Geometry*.

Comme nous l'avons remarqué, les réseaux de neurones peuvent s'interpréter comme un modèle statistique à réponse non linéaire : $y = f(x, w)$, et en supposant que les données suivent la densité $q(x)$ et qu'on a la densité conditionnelle $p(y|x, w)$, l'ensemble des données (x, y) suivent la loi :

$$q(x)p(y|x, w)$$

Les modèles de perceptrons que nous présenterons ensuite sont considérés par Amari comme des points d'une variété qu'il appelle *Neuromanifold*, où les coordonnées sont les w et la métrique est donnée par l'information de Fisher.

2.8 Machines de Boltzmann

2.8.1 Introduction

Un autre exemple fortement lié aux réseaux de neurones est celui de la **machine de Boltzmann**, qui est un réseau totalement connecté de n neurones stochastiques. Chacun des neurones N_i possède un état x_i qui est 0 ou 1, qu'il communique en *output* aux autres neurones. L'algorithme d'apprentissage est très lent de manière générale pour les réseaux multicouches mais est rapide dans le cadre des "restricted Boltzmann machines" (que l'on notera RBM) qui ne comporte qu'une seule couche de détecteurs de caractéristiques.

Les machines de Boltzmann sont utilisées pour la résolution de deux types de problèmes. Pour un problème de recherche, les poids (même typologie que pour les réseaux neuronaux) sont fixés et sont utilisés dans une fonction de coûts. Ce type de machine permet alors d'échantillonner des vecteurs à valeurs binaires de sorte pour lesquels la fonction de coût est faible.

Dans le cas d'un problème d'apprentissage, la machine de Boltzmann dispose en entrée de vecteurs de données binaires et apprend à générer ce type de vecteurs avec une forte probabilité. Pour ce faire, les poids doivent être modifiés de sorte à ce que par rapport à d'autres vecteurs binaires les données d'entrées prennent de faibles valeurs de la fonction de coût. Afin de résoudre ce type de problèmes, la machine met à jour les poids et chaque mise à jour nécessite la résolution de beaucoup de problèmes de recherche différents.

2.8.2 Dynamique stochastique d'une machine de Boltzmann

Chaque neurone calcule la quantité

$$u_i = \sum_{j \neq i} w_{ij} x_j - h_i$$

en fonction des signaux qu'il reçoit. Le poids $w_{i,j}$, appelé poids de la connexion synaptique, mesure l'influence du neurone N_j sur le neurone N_i , et h_i est appelé le seuil de N_i . On suppose que la matrice des poids $W = [w_{i,j}]$ est symétrique à diagonale nulle.

A chaque étape, chaque neurone N_i détermine s'il sera dans l'état excité 1 ou dans l'état de repos 0 selon la probabilité :

$$\mathbb{P}(x_i = 1) = \frac{e^{u_i}}{1 + e^{u_i}}$$

L'état de la machine de Boltzmann est représenté par le vecteur $x = (x_1, \dots, x_n)$, qui suit une chaîne de Markov sur l'espace de tous les états possibles, espaces à 2^n points. Cet chaîne admet comme loi stationnaire :

$$p^{W,h}(x) = \frac{1}{Z} \exp\{-E(x)\}$$

$$\text{avec } E(x) = -\frac{1}{2}x^T W x + h^T x$$

$$\text{et } Z = \sum_x \exp\{-E(x)\}$$

On peut prendre le point de vue géométrique en se représentant la machine de Boltzmann comme un système qui se comporte selon la loi stationnaire $p^{W,h}(x)$, donc comme un point de coordonnées (W, h) dans la variété de toutes les machines de Boltzmann. (Encore un exemple de famille exponentielle.)

Afin de déterminer dans lequel la i ème unité se trouve, dans un premier temps on calcule l'input total noté z_i qui est la somme du biais associé à cette unité b_i et des poids sur les connections provenant des autres unités **activées** :

$$z_i = b_i + \sum_j s_j w_{i,j}$$

avec s_j l'état de l'unité j et $w_{i,j}$ le poids associé à la connection entre i et j . L'unité i passe en position "on" avec la probabilité donnée par la fonction logistique :

$$\mathbb{P}(s_i = 1) = \frac{1}{1 + \exp(-z_i)}$$

Si les unités sont mises à jour séquentiellement dans un ordre quelconque, le réseau tend vers sa distribution stationnaire dans laquelle la probabilité d'un vecteur d'état est déterminé uniquement par son "énergie" par rapport à toutes les énergies de tous les autres vecteur d'état binaire.

$$\mathbb{P}(\vec{v}) = \frac{\exp(-E(\vec{v}))}{\sum_{\vec{u}} \exp(-E(\vec{u}))}$$

avec E définie par

$$E(\vec{v}) = -\sum_i s_i^v b_i - \sum_{i < j} s_i^v s_j^v w_{i,j}$$

où s_i^v est l'état de l'unité donnée par le vecteur d'état \vec{v} .

2.8.3 Learning dans les machines de Boltzmann

Sans couches cachées

Etant donné un ensemble de vecteurs d'état d'entraînement, apprendre consiste à trouver les poids et biais (paramètres) qui permettent d'obtenir une distribution stationnaire pour laquelle les vecteurs d'état ont une forte probabilité. Par différentiation dans les équations ci-dessus, on obtient :

$$\begin{aligned} \left\langle \frac{\partial \log \mathbb{P}(\vec{v})}{\partial w_{i,j}} \right\rangle_{data} &= -\frac{\partial E(-\vec{v})}{\partial w_{i,j}} - \frac{\partial \sum_{\vec{u}} \exp(-E(\vec{u}))}{\partial w_{i,j}} \\ \left\langle \frac{\partial \log \mathbb{P}(\vec{v})}{\partial w_{i,j}} \right\rangle_{data} &= \langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model} \end{aligned}$$

avec $\langle . \rangle_{data}$ est une espérance associée à la distribution des datas et $\langle . \rangle_{model}$ est l'espérance obtenue lorsque la machine de Boltzmann a une distribution stationnaire. La remontée de gradient s'effectue à partir de : $w_{i,j} = w_{i,j} + \epsilon \langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model}$

De même, $b_i = b_i + \epsilon \langle s_i \rangle_{data} - \langle s_i \rangle_{model}$.

Avec couches cachées

On dit qu'il y a des couches cachées si les états de certaines unités ne sont pas déterminés par les données visibles. L'intérêt de ce type de couches est qu'elles permettent à la machine de Boltzmann de modéliser des distributions sur les données visibles qui ne pourraient pas être modélisées par des interactions par paires entre les unités visibles. Toutefois la règle d'apprentissage demeure inchangée. Dans ce cas, $\langle s_i s_j \rangle_{data}$ est la moyenne sur tous les vecteurs d'entrées de $s_i s_j$ lorsque le vecteur d'entrée est fixé et qu'on actualise les couches cachées jusqu'à l'équilibre.

2.9 Restricted Boltzmann machines

2.9.1 Introduction

Une RBM est constituée d'une couche visible et d'une couche cachée. Il n'y a pas de connections entre les unités visibles ni entre les unités cachées. Par conséquent, les unités de la couche cachées sont conditionnellement indépendantes étant donné un vecteur d'entrée "visible". Ainsi échantillonner à partir de $\langle s_i s_j \rangle_{data}$ s'obtient en une étape parallélisée. On a néanmoins toujours besoin de plusieurs itérations d'actualisation des poids de la couche cachée et de la couche visible pour échantillonner à partir $\langle s_i s_j \rangle_{model}$. Toutefois on peut remplacer $\langle s_i s_j \rangle_{model}$ par $\langle s_i s_j \rangle_{reconstruction}$ que l'on obtient de la manière suivante :

- Un vecteur visible en entrée et on met à jour toutes les unités de la couche cachée
- Mettre à jour les unités de la couche visible pour obtenir une "reconstruction"
- Mettre à jour à nouveau toutes les unités de la couche cachée.

Cette procédure d'apprentissage approxime la descente de gradient et on l'appelle "contrastive divergence".

2.9.2 Deep Learning à l'aide de RBM

Après apprentissage dans la couche cachée, les vecteurs d'activation de cette couche peuvent être considérés à leur tour comme des données d'entraînement pour une nouvelle machine de Boltzmann et ce de manière répétée. C'est la partie d'apprentissage non-supervisé.

Les probabilités conditionnelles peuvent donc s'écrire, avec \vec{v} le vecteur des données visibles et \vec{h} celui des données de la couche cachée :

$$\mathbb{P}(\vec{v}|\vec{h}) = \prod_{i=1}^m \mathbb{P}(v_i|\vec{h})$$

$$\mathbb{P}(\vec{h}|\vec{v}) = \prod_{j=1}^n \mathbb{P}(h_j|\vec{v})$$

On peut écrire :

$$\mathbb{P}(h_j = 1|\vec{v}) = \sigma \left(b_j + \sum_{i=1}^m w_{i,j} v_i \right)$$

$$\mathbb{P}(v_i = 1|\vec{h}) = \sigma \left(b_i + \sum_{j=1}^n w_{i,j} h_j \right)$$

avec σ la fonction d'activation logistique.

Ainsi les RBM sont entraînées à maximiser le produit des probabilités associés à un ensemble d'entraînement V et détermine $\operatorname{argmax}_W \prod_{\vec{v} \in V} \mathbb{P}(\vec{v})$ ou de manière équivalente $\operatorname{argmax}_W \mathbb{E} [\sum_{\vec{v} \in V} \log(\mathbb{P}(\vec{v}))]$

2.9.3 Précisions sur l'algorithme de Contrastive divergence (noté CD-1)

Revenons sur l'algorithme décrit brièvement plus haut :

- On prend un échantillon d'entraînement \vec{v} , on calcule les probabilités des unités cachées et on échantillonne un vecteur d'activation caché h de cette distributions
- On calcule le produit dyadique de v et h que l'on appellera le gradient positif
- De h , on échantillonne une reconstruction v' d'unités visibles puis on ré-échantillonne h' de v'
- On calcule le produit dyadique de v' et h' que l'on appellera le gradient négatif

- On actualise les poids en leur ajoutant le gradient positif moins le gradient négatif multiplié par un taux d'apprentissage ϵ : $\Delta w_{i,j} = \epsilon(vh^T - v'h'^T)$.
On effectue la même chose pour les biais a et b .

Ainsi si l'on note \mathbf{v}^0 le vecteur initial et qu'on répète la procédure initiale en échantillonnant selon l'algorithme de Gibbs, on obtient :

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{i,j}} = \langle v_i^0 h_j^0 \rangle - \langle v_i^\infty h_j^\infty \rangle$$

2.10 Multilayer Neural Network

On se donne un réseau de neurones spécifié par un paramètre $w \in \mathbb{R}^n$, qui représente les poids modifiables des connexions entre les synapses. En entrée du réseau, le signal x , qui suit une loi de probabilité inconnue $q(x)$, est traité, et le réseau calcule une sortie $f(x, w)$.

Le but est d'entraîner les neurones : on est en apprentissage supervisé. Lorsque l'on donne l'entrée x au réseau, on peut donc lui spécifier quelle sortie y lui correspond. La discussion qui suit permettra de comprendre comment nous pouvons élaborer un algorithme qui, par itération, améliore les poids jusqu'à atteindre un poids possiblement optimal w^* .

Soit L une fonction de perte, typiquement :

$$L(x, y, w) = ||y - f(x, w)||^2.$$

En considérant un modèle statistique où le but est une version bruitée du signal de sortie, avec un bruit normal centré, ie :

$$y = f(x, w) + \eta$$

$$\text{avec } \eta \sim \mathcal{N}(0, I_n)$$

la densité du couple (x, y) prend la forme :

$$cq(x) \exp\{-\frac{1}{2}||y - f(x, w)||^2\}.$$

Face à une série d'exemples $(x_1, y_1), \dots, (x_N, y_N)$, l'algorithme naturel de descente est donné par :

$$w_{n+1} = w_n - \epsilon_n \nabla l(x_n, y_n, w_n),$$

où l est le log de la densité du couple (x, y) , et ϵ_n est le pas d'apprentissage. Cet algorithme nous fait nous déplacer sur l'espace des paramètres w . Dans Amari1985, on peut donner une structure de variété riemannienne à cet espace, structure dont la métrique est donnée par :

$$g_{ij}(w) = \mathbb{E}[\partial_i p(x, y, w) \partial_j p(x, y, w)].$$

2.11 Cas du perceptron

On peut obtenir une forme explicite pour le perceptron multicouche. Ici, la fonction signal est donnée par :

$$f(u) = \frac{1 - e^{-u}}{1 + e^{-u}}$$

, et $y = f(w.x) + \eta$, $\eta \sim \mathcal{N}(0, \sigma^2)$.

La densité conditionnelle de y sachant x est alors :

$$p(y|x, w) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2\sigma^2} \|y - f(x, w)\|^2\right\},$$

ce qui, combiné à l'hypothèse $q(x)$ gaussienne, donne une densité jointe :

$$p(x, y, w) = q(x)p(y|x, w)$$

Le théorème suivant, dont la preuve se trouve dans *Natural Gradient work-efficiently in learning*, donne explicitement la forme de la métrique de Fisher dans le cas du perceptron multicouches, sous nos hypothèses.

Théorème 3. (Amari) La métrique de Fisher vaut :

$$G(w) = |w|^2 c_1(w) I_n + (c_2(w) - c_1(w)) w \otimes w$$

où :

$$c_1(w) = \frac{1}{4\sqrt{2\pi}\sigma^2 |w|^2} \int (f^2(wt) - 1)^2 \exp\left(-\frac{1}{2}t^2\right) dt$$

$$c_2(w) = \frac{1}{4\sqrt{2\pi}\sigma^2 |w|^2} \int (f^2(wt) - 1)^2 t^2 \exp\left(-\frac{1}{2}t^2\right) dt$$

La matrice inverse est :

$$G^{-1}(w) = \frac{1}{|w|^2 c_1(w)} I_n + \frac{1}{|w|^4} \left(\frac{1}{c_2(w)} - \frac{1}{c_1(w)} \right) w \otimes w$$

Nous pouvons alors donner une formule explicite pour l'algorithme de gradient naturel :

$$w_{n+1} = w_n + \epsilon_n (y_n - f(w_n.x_n)) f'(w_n.x_n) \left\{ \frac{1}{|w_n|^2 c_1(w_n)} x_n + \frac{1}{|w_n|^4} \left(\frac{1}{c_2(w)} - \frac{1}{c_1(w)} \right) w_n.x_n w_n \right\}$$

En guise de remarque finale pour cette partie, mentionnons que cette méthode se généralise facilement (bien qu'avec plus de calculs) au cas d'un perceptron multicouche à sortie linéaire, possédant m couches. La relation *input-output* s'écrit ici :

$$y = \sum_{i=1}^m v_i f(w_i.x) + \eta$$

$$\eta \sim \mathcal{N}(0, I_n)$$

Le calcul de G^{-1} est plus facile que dans le cas classique (comparez l'inversion d'une matrice $(n+1) \times m$ à celle d'une matrice $2 \times (m+1)$), et Amari et G. Yang ont effectué des études sur cette méthode : elle pourrait éviter l'effet plateau que les méthodes classiques peinent tant à éviter.

3 Application à la reconnaissance de caractères manuscrits

3.1 Rappel sur les ondelettes

Les ondelettes forment des bases hilbertiennes de $L^2(\mathbb{R})$ en partant d'une fonction ϕ , appelée fonction mère, que l'on translate-échelonne ensuite :

$$\{2^{j/2}\phi(2^j \cdot - k)\}_{k \in \mathbb{Z}, j \in \mathbb{Z}}.$$

Par exemple la fonction mère $\phi(u) = 1_{0 < u < \frac{1}{2}} - 1_{\frac{1}{2} < u < 1}$ donne la base de Haar, très utilisée en traitement d'image. Voici quelques une de ces fonctions :

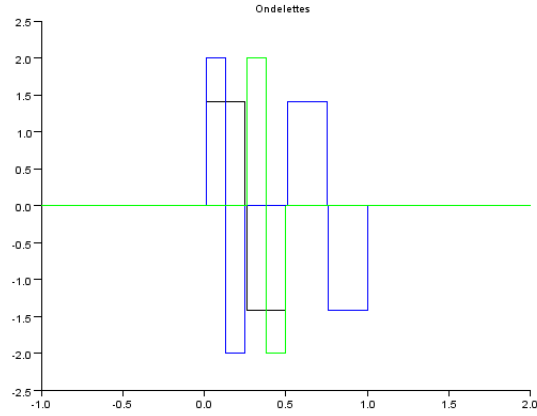


FIGURE 8 – Ondelettes tracées avec Scilab

3.2 Description de l'algorithme

On entraîne le réseau de neurones avec des images de lettres manuscrites et de la lettre correspondante. Les images sont représentées par un vecteur N dimensionnel, dont les coordonnées sont les N premiers coefficients sur la base d'ondelette.

4 Deep Learning vs Support Vector Machine

Première Deep Learning architecture : Neocognitron.

Idée : entraîner une couche à la fois en la traitant comme une machine de Boltzmann restreinte (RBM), et ensuite appliquer un algorithme de Backpropagation.

Rôle de Yan Lecun : première utilisation de l'algorithme de Backpropagation.

Disponible depuis les années 70s, cet algorithme était considéré comme trop lent en pratique : problème du vanishing gradient.

Algo *em* et *EM* ?

5 Appendice

5.1 Géométrie différentielle