Réseaux de neurones

Adam Salvail-Bérard

6 septembre 2012

Résumé

Le but de ce document est de donner une introduction simple, mais formelle, du procédé de construction des réseaux de neurones artificiels. En plus de la définition d'un réseau de neurones, sont données les versions scalaire, vectorielle et par lot des formules utilisées dans l'apprentissage des paramètres du réseau de neurones. Les algorithmes de propagation des données et de rétropropagation du gradient de l'erreur d'apprentissage sont abordés. Finalement, dans le contexte d'optimisation des réseaux de neurones, une méthode exacte de calcul du produit de la matrice hessienne et d'un vecteur est montrée.

1 Introduction

Les réseaux de neurones sont des constructions abstraites simulant l'activité d'un réseau de neurones biologique simplifié. Ils sont utilisés en apprentissage automatique pour construire un modèle à partir de données existantes dans le but d'effectuer des prédictions sur de nouvelles données soit à l'aide de régression, dans le cas continu, ou de classification, dans le cas discret. Une introduction plus complète à l'apprentissage automatique et son cousin le forage de données est présentée dans [5].

L'apprentissage automatique [1] concerne l'ajustement de paramètres d'une « boite » qui prend en entrée des données et fournit en sortie un résultat. C'est la métaphore classique décrivant une fonction. En statistiques, la boite est souvent simple et peut utiliser l'analyse mathématique pour caractériser les paramètres optimaux. Par exemple, la régression classique utilise une « boite » constituée d'une combinaison linéaire de fonctions élémentaires pertinentes et l'ajustement consiste simplement à optimiser les coefficients de la combinaison linéaire. Le contexte moderne du forage de données a motivé le développement de « boites » plus complexes, impossibles à optimiser analytiquement. Les réseaux de neurones sont un exemple de tels modèles complexes qui ont fait leurs preuves dans plusieurs domaines comme la détection de fraudes ou la reconnaissance de caractères manuscrits.

Ce texte s'intéresse donc aux réseaux de neurones, à ouvrir la boite métaphorique qui est associée à cette classe de modèles. Un réseau de neurones est

1. De l'anglais *machine learning*.

un graphe orienté de nœuds, appelés neurones, souvent disposés en couches. Ces nœuds reçoivent une information de leurs prédécesseurs (les neurones de la couche précédente) et combinent cette information selon des pondérations identifiées par $w_{j,k}^{(i)}$. Chaque nœud possède également un seuil d'activation $b_k^{(i)}$. Autour de ce seuil, la valeur de sortie est beaucoup plus sensible aux changements de la valeur d'entrée.

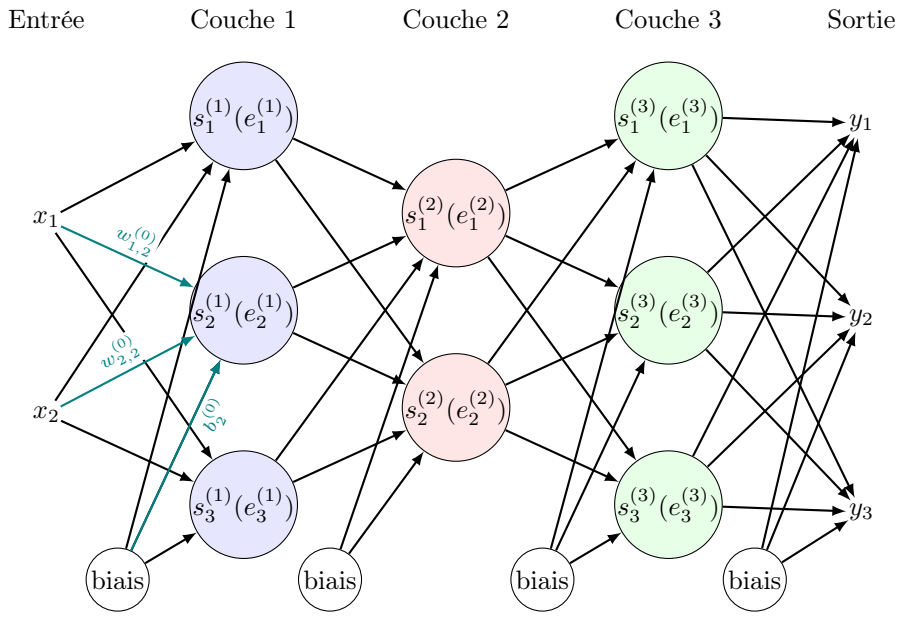

FIGURE 1 – Illustration d'un réseau de neurones *feed-forward* à 5 couches dont 3 couches cachées, une couche d'entrée et une couche de sortie. Ce type de réseau est caractérisé par le fait qu'un neurone ne tire ses entrées que de la couche précédente. À titre d'exemple, les entrées du deuxième nœud de la seconde couche ont été étiquettées en bleu.

Le comportement du réseau de neurones dépend donc des $w_{j,k}^{(i)}$ qui servent à pondérer les apports des nœuds d'une couche à la suivante, et aussi des seuils d'activation de chaque nœud. Si ces paramètres sont fixés arbitrairement, les résultats fournis seront médiocres. Il convient d'utiliser un jeu de données nommé ensemble d'entrainement pour optimiser les paramètres (les $w_{j,k}^{(i)}$ et les $b_j^{(i)}$) afin de minimiser l'erreur de prédiction du réseau. Pour cet ensemble d'entrainement, le résultat théorique est connu et l'optimisation consiste à minimiser la somme des carrées des différences entre les sorties calculées et les sorties attendues. Contrairement aux modèles classiques en statistique, il est impossible d'obtenir une solution analytique à ces problèmes d'optimisation. Il faut donc optimiser numériquement la fonction d'erreur.

Cet article présente une description rigoureuse de ces réseaux de neurones. De plus, une méthode est développée pour évaluer les dérivées premières et secondes (dans une direction donnée) d'une fonction d'erreur servant à identifier les paramètres optimaux. Cette information sur les dérivées est très utile pour les algorithmes d'optimisation qui s'en servent pour trouver rapidement une solution optimale au problème de minimisation de l'erreur.

2 Définitions

Un réseau de neurones agit sur des données. Ces données sont définies formellement comme étant un ensemble de couples $(\mathbf{x}, \mathbf{c})$ avec $\mathbf{x} \in \mathbb{R}^n$ et $\mathbf{c} \in \mathbb{R}^m$, $n, m \in \mathbb{N}^*$. En outre, un réseau de neurones peut être défini comme étant une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ qui prend en entrée les informations du vecteur $\mathbf{x}$ et qui effectue une prédiction $\mathbf{y} \in \mathbb{R}^m$ sur le vecteur $\mathbf{c}$. Le but de la construction d'un réseau de neurones devient de trouver la fonction f qui donne la meilleure approximation de $\mathbf{y}$ comparée à la cible $\mathbf{c}$.

Comme son nom l'indique, un réseau de neurones est une collection de nœuds, nommés neurones, disposée en r couches. La couche 0 représente l'entrée et possède n neurones, soit la dimension du vecteur d'entrée $\mathbf{x}$. De même, la r-ième couche, la couche de sortie, possède m neurones, soit la dimension du vecteur de sortie $\mathbf{y}$. Les couches intermédiaires, nommées couches cachées, possèdent un nombre de neurones arbitraires, souvent déterminé par la complexité du problème à résoudre.

Afin de propager les valeurs d'entrée, chaque neurone de la couche $i, 0 \leq i < r$, possède une connexion avec tous les neurones de la couche $i+1$ [2]. Chacune de ces connexions est pondérée par un poids $w_{j,k}^{(i)}$ défini comme étant le poids liant la sortie $s_j^{(i)}$ du j-ième neurone de la couche i à l'entrée $e_k^{(i+1)}$ du k-ième neurone de la couche $i+1$. Les poids d'une même couche $i+1$ forment une matrice notée $\mathbf{w}^{(i)}$. La figure 1 illustre ces notations à l'aide d'une représentation graphique d'un réseau de neurones.

Les neurones d'un réseau de neurones représentent les unités de calcul du modèle. Chaque neurone possède une entrée (un scalaire), une fonction d'activation et une sortie. Par exemple, le k-ième neurone de la couche i reçoit une valeur d'entrée

$$e_k^{(i)} = \sum_j w_{j,k}^{(i-1)} s_j^{(i-1)} + b_k^{(i-1)} \tag{1}$$

et retourne la valeur $s_k^{(i)} = a_k^{(i)}\left(e_k^{(i)}\right)$ qui est la fonction d'activation appliquée à l'entrée du neurone.

La valeur $b_k^{(i-1)}$ est un biais ajouté à l'entrée du k-ième neurone de la couche i servant, en quelque sorte, de seuil déterminant à partir de quelle valeur le neurone est significativement activé. En d'autres termes, le biais dicte à partir

2. Ceci est la représentation classique d'un réseau de neurones. D'autres topologies, c'est-à-dire dispositions des neurones, sont possibles. Dans ce document, seul ce type de réseau de neurones, appelé *feed-forward*, est discuté.

de quelle valeur la somme des produits des poids et des sorties de la couche précédente passe le seuil où un neurone est considéré actif.

Finalement, il y a la fonction d'erreur E (voir formule (3)) qui dénote l'erreur du réseau de neurones dans son approximation de la sortie $\mathbf{y}$ par rapport à la cible $\mathbf{c}$. C'est cette fonction qui détermine la qualité du modèle de comportement des données.

3 Réseau de neurones *feed-forward*

Un réseau de neurones typique est le réseau *feed-forward*. Ce réseau propage l'entrée du réseau vers les couches suivantes sans jamais revenir en arrière. Ce type de réseau est utilisé pour le reste de cet article.

En plus du type de réseau de neurones, il faut également choisir une fonction d'erreur et une fonction d'activation pour les neurones. Ces choix sont souvent guidés par le type de données traitées. Plus de détails sont disponibles dans l'ouvrage de Bishop [1] au chapitre 5.

Dans cet article, la fonction d'activation utilisée est une fonction logistique définie par $a_j^{(i)}(x) = \frac{1}{1+\exp(-x)}$, sauf pour la couche de sortie qui garde la fonction identité comme fonction d'activation. De cette façon, la valeur de sortie

$$s_j^{(i)} = a_j^{(i)}(e_j^{(i)}) = \frac{1}{1+\exp(-e_j^{(i)})} \text{ pour } i \neq r. \tag{2}$$

$$s_j^{(r)} = e_j^{(r)}.$$

Pour la fonction d'erreur, un choix simple est d'utiliser

$$E = \frac{1}{2}\|\mathbf{y} - \mathbf{c}\|_2^2 = \frac{1}{2}\sum_{i=1}^{m}(y_i - c_i)^2, \tag{3}$$

soit la moitié du carré de la distance euclidienne entre la sortie du réseau et la cible.

Avec ces définitions, tous les outils nécessaires à l'établissement d'une prédiction sont présents. Pour effectuer la propagation avec un couple $(\mathbf{x}, \mathbf{c})$, les sorties de la couche d'entrée sont assignées aux valeurs du vecteur $\mathbf{x}$, c'est-à-dire que $s_j^{(1)} = x_j$. Les valeurs sont ensuite propagées au travers du réseau de neurones grâce aux formules (1) et (2) pour donner les sorties $y_j = s_j^{(r)}$ qui sont comparées aux cibles c_j par la fonction d'erreur E (3). Il faut maintenant minimiser la moyenne des erreurs données par la fonction E sur l'ensemble des données fournies en entrée :

$$E_{\text{moyenne}} = \frac{1}{N}\sum_{t=1}^{N} E_t$$

où N est le nombre de couples donné en entrainement au réseau de neurones et E_t représente la t-ième erreur d'apprentissage.

4 Construction du réseau de neurones

Bien entendu, au départ, un réseau de neurones avec des poids initialisés aléatoirement donne de pauvres résultats. C'est pourquoi il y a une étape d'apprentissage, c'est-à-dire d'ajustement des poids du réseau afin que ceux-ci puissent donner de meilleures prédictions.

Une façon de voir le problème est d'essayer de minimiser l'erreur de prédiction du modèle. Cette formulation transforme le problème d'apprentissage en un problème d'optimisation pour lequel plusieurs algorithmes existent. L'optimisation de la fonction d'erreur s'effectue par l'ajustement des paramètres de la fonction, c'est-à-dire les poids et les biais du réseau de neurones.

Soit la fonction $\dim(i)$ qui donnne le nombre de neurones de la couche i. Soit

$$\theta = \begin{bmatrix} w_{j,k}^{(i)} & b_j^{(i)} \end{bmatrix} \tag{4}$$

où $0 \leq i < r$, $0 \leq j < \dim(i)$ et $0 \leq k < \dim(i+1)$ qui représentent le vecteur rassemblant chacun des paramètres du réseau de neurones. Le problème d'optimisation associé est de trouver les paramètres du réseau de neurones permettant de minimiser l'erreur $E(\theta)$. Formellement, il faut trouver

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{t=1}^{N} E_t(\theta). \tag{5}$$

Néanmoins, l'application de ces algorithmes d'optimisation nécessite souvent d'avoir plus d'information sur la fonction d'erreur $E(\theta)$, comme ses dérivées de premier et second ordre par rapport aux paramètres θ de la fonction d'erreur.

Trouver les dérivées de la fonction d'erreur est un problème complexe. Puisque la fonction d'erreur, avec toutes les couches du réseau, peut facilement devenir difficile à appréhender, il devient impossible de déduire une forme analytique de la solution au problème de minimisation de l'erreur moyenne. Il faut donc procéder avec des méthodes itératives. La méthode la plus élégante pour l'obtention de la dérivée de premier ordre est la rétropropagation de l'erreur de prédiction. Pour la dérivée de second ordre, une méthode proposée par Pearlmutter [3] est décrite. Cette méthode donne le résultat exact du produit de la matrice hessienne et d'un vecteur arbitraire.

4.1 Rétropropagation

L'idée de la rétropropagation est de faire circuler l'information sur la dérivée de la fonction d'erreur à partir de la couche de sortie, où l'erreur de prédiction est connue (la différence entre $\mathbf{y}$ et $\mathbf{c}$), jusqu'à la couche d'entrée. Pour y arriver, il faut arriver à exprimer la dérivée de la fonction d'erreur d'un nœud en fonction de l'information donnée par les couches suivantes. De cette façon, les valeurs attendues de la couche de sortie pourront être rétropropagées vers l'entrée.

Tout d'abord, la dérivée, à la sortie, de la fonction d'erreur (3) est donnée par

$$\frac{\partial E}{\partial y_i} = y_i - c_i. \tag{6}$$

À partir de cette équation, il est possible de trouver la dérivée de la fonction d'erreur à l'entrée de la couche de sortie. La dérivée reste la même puisque la fonction d'activation de la dernière couche est la fonction identité, c'est-à-dire

$$\frac{\partial E}{\partial e_j^{(r)}} = \frac{\partial E}{\partial s_j^{(r)}} = \frac{\partial E}{\partial y_j}.$$

À partir de ce moment, il faudra se servir de la formule de dérivée en chaine pour trouver, de façon récursive, les autres dérivées. En d'autres mots, il faut pouvoir exprimer chaque dérivée de la fonction d'erreur en terme de dérivée venant plus loin dans le réseau de neurones.

Pour ce faire, il faut voir que la fonction d'erreur est affectée par la sortie d'un neurone de la i-ième couche à travers l'ensemble des neurones de la $(i+1)$-ième couche. Donc,

$$\frac{\partial E}{\partial s_j^{(i)}} = \sum_k w_{j,k}^{(i)} \frac{\partial E}{\partial e_k^{(i+1)}}. \tag{7}$$

De plus, la fonction d'erreur est affectée par l'entrée d'un neurone de la i-ième couche seulement au travers de la sortie de ce même neurone. Il en résulte que

$$\frac{\partial E}{\partial e_j^{(i)}} = \frac{\partial E}{\partial s_j^{(i)}} \frac{\partial s_j^{(i)}}{\partial e_j^{(i)}}$$

. Sachant que la dérivée de la fonction d'activation logistique est donnée par $\frac{\partial a}{\partial x} = a(1-a)$, la dérivée de la fonction d'erreur par l'entrée d'un neurone est donc

$$\frac{\partial E}{\partial e_j^{(i)}} = \frac{\partial E}{\partial s_j^{(i)}}\, s_j^{(i)}(1 - s_j^{(i)}). \tag{8}$$

La formule (8) permet de rétropropager la dérivée de l'erreur jusqu'à la première couche. À l'aide de ces valeurs, il est possible d'arriver à la dérivée de la fonction d'erreur par rapport à chacun des poids du réseau de neurones, en appliquant la règle de dérivée en chaine. Donc,

$$\frac{\partial E}{\partial w_{j,k}^{(i)}} = \frac{\partial E}{\partial e_k^{(i+1)}} \frac{\partial e_k^{(i+1)}}{\partial w_{j,k}^{(i)}} = \frac{\partial E}{\partial e_k^{(i+1)}} s_j^{(i)}. \tag{9}$$

Finalement, comme les biais sont également des paramètres impliqués dans l'apprentissage d'un réseau de neurones, il faut aussi pouvoir leur trouver une valeur optimale. Il est donc nécessaire de trouver la dérivée de l'erreur par rapport aux biais. De la même façon qu'avec les poids, un biais n'affecte l'erreur

de prédiction qu'au travers de l'entrée du nœud auquel il est rattaché. Donc, grâce à la dérivée en chaine et la formule (1),

$$\frac{\partial E}{\partial b_j^{(i)}} = \frac{\partial E}{\partial e_j^{(i+i)}} \frac{\partial e_j^{(i+1)}}{\partial b_j^{(i)}} = \frac{\partial E}{\partial e_j^{(i+i)}}. \tag{10}$$

4.2 Calcul de la dérivée seconde

Pour pouvoir minimiser la fonction d'erreur de manière efficace, plusieurs algorithmes d'optimisation nécessitent des informations sur la dérivée seconde qui représente le comportement de la fonction autour d'un point. Dans le cas de la fonction d'erreur, une fonction avec un domaine dans $\mathbb{R}^m$, sa dérivée seconde est en fait la matrice hessienne

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial {\theta_1}^2} & \frac{\partial^2 E}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 E}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 E}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 E}{\partial {\theta_2}^2} & \cdots & \frac{\partial^2 E}{\partial \theta_2 \partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 E}{\partial \theta_n \partial \theta_2} & \cdots & \frac{\partial^2 E}{\partial {\theta_n}^2} \end{bmatrix}.$$

Malheureusement, cette matrice, pour de grandes dimensions, devient difficile à stocker et à calculer, même pour un ordinateur. Puisque la plupart des algorithmes utilisent plutôt le produit de la matrice hessienne avec un vecteur arbitraire $\mathbf{v}$ (résultant en un vecteur de dimension m), il est plus facile de calculer directement ce vecteur plutôt que la matrice hessienne entière.

En se basant sur le principe de la propagation de valeurs exposé plus haut, il est possible de trouver la valeur de $\mathbf{Hv}$ en apportant quelques changements aux formules déjà données. Cette technique, sous la forme présentée dans cet article, est attribuée à Pearlmutter [3].

Le produit hessienne-vecteur peut être défini par

$$\mathbf{Hv} = \frac{\partial}{\partial q} \nabla_\theta (\theta + qv) \bigg|_{q=0}.$$

De cette définition, il est possible d'extraire un nouvel opérateur

$$\mathcal{R}\{f(\theta)\} = \frac{\partial}{\partial q} f(\theta + qv) \bigg|_{q=0}$$

où $\mathbf{v}$ représente le vecteur par lequel la matrice hessienne est multipliée et qui est de même dimension que le vecteur θ. Pour fins de concordance, les éléments $v_{j,k}^{(i)}$ et $v_j^{(i)}$ du vecteur $\mathbf{v}$ sont respectivement à la même position que les éléments $w_{j,k}^{(i)}$ et $b_j^{(i)}$ du vecteur θ.

Par définition, $\mathcal{R}\left\{w_{j,k}^{(i)}\right\} = v_{j,k}^{(i)}$ et $\mathcal{R}\left\{b_k^{(i)}\right\} = v_k^{(i)}$ puisque les $w_{j,k}^{(i)}$ et des $b_k^{(i)}$ sont les paramètres à optimiser. En appliquant les règles usuelles de dérivée et

de dérivée en chaine, il est possible d'obtenir des formules de propagation pour les calculs de $\mathcal{R}\{\cdot\}$. Pour l'entrée dans un neurone, la formule devient, avec la règle de la dérivée d'un produit,

$$\begin{aligned}\mathcal{R}\left\{e_k^{(i+1)}\right\} &= \mathcal{R}\left\{\sum_j w_{j,k}^{(i)} s_j^{(i)} + b_k^{(i)}\right\} \\ &= \sum_j w_{j,k}^{(i)} \mathcal{R}\left\{s_j^{(i)}\right\} + \mathcal{R}\left\{w_{j,k}^{(i)}\right\} s_j^{(i)} + \mathcal{R}\left\{b_k^{(i)}\right\} \\ &= \sum_j w_{j,k}^{(i)} \mathcal{R}\left\{s_j^{(i)}\right\} + v_{j,k}^{(i)} s_j^{(i)} + v_k^{(i)}. \end{aligned} \tag{11}$$

Pour la sortie d'un neurone, la formule devient

$$\mathcal{R}\left\{s_j^{(i)}\right\} = s_j^{(i)}(1 - s_j^{(i)})\mathcal{R}\left\{e_j^{(i)}\right\}. \tag{12}$$

À partir des formules (11) et (12), il est possible de faire la propagation des valeurs données en entrée pour le calcul de la dérivée seconde. Cette propagation peut facilement être calculée en parallèle de la propagation usuelle des valeurs d'entrée.

À l'image de la rétropropagation, il faut également rétropropager la dérivée de la fonction d'erreur sur laquelle est appliqué l'opérateur $\mathcal{R}\{\cdot\}$. Il faut donc appliquer l'opérateur $\mathcal{R}\{\cdot\}$ à la formule de la dérivée de la sortie :

$$\mathcal{R}\left\{\frac{\partial E}{\partial y_j}\right\} = \mathcal{R}\{y_j - c_j\} = \mathcal{R}\{y_j\} = \mathcal{R}\left\{s_j^{(r)}\right\}. \tag{13}$$

De façon identique à la rétropropagation,

$$\mathcal{R}\left\{\frac{\partial E}{\partial y_j}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial s_j^{(r)}}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial e_j^{(r)}}\right\}$$

. Les formules récursives de la rétroprapagation deviennent

$$\begin{aligned}\mathcal{R}\left\{\frac{\partial E}{\partial e_j^{(i)}}\right\} &= \mathcal{R}\left\{\frac{\partial E}{\partial s_j^{(i)}}\frac{\partial s_j^{(i)}}{\partial e_j^{(i)}}\right\} \\ &= \frac{\partial E}{\partial s_j^{(i)}}\mathcal{R}\left\{\frac{\partial s_j^{(i)}}{\partial e_j^{(i)}}\right\} + \mathcal{R}\left\{\frac{\partial E}{\partial s_j^{(i)}}\right\}\frac{\partial s_j^{(i)}}{\partial e_j^{(i)}} \\ &= \frac{\partial E}{\partial s_j^{(i)}}\frac{\partial^2 s_j^{(i)}}{\partial e_j^{(i)2}}\mathcal{R}\left\{e_j^{(i)}\right\} + \mathcal{R}\left\{\frac{\partial E}{\partial s_j^{(i)}}\right\}\frac{\partial s_j^{(i)}}{\partial e_j^{(i)}} \end{aligned} \tag{14}$$

où $\frac{\partial s_j^{(i)}}{\partial e_j^{(i)}} = s_j^{(i)}(1 - s_j^{(i)})$ et $\frac{\partial^2 s_j^{(i)}}{\partial e_j^{(i)2}} = s_j^{(i)}(1 - s_j^{(i)})(1 - 2s_j^{(i)})$.

De même, pour la sortie d'un neurone,

$$\begin{aligned}\mathcal{R}\left\{\frac{\partial E}{\partial s_j^{(i)}}\right\} &= \mathcal{R}\left\{\sum_j w_{j,k}^{(i)} \frac{\partial E}{\partial e_k^{(i+1)}}\right\} \\ &= \sum_j \left(w_{j,k}^{(i)} \mathcal{R}\left\{\frac{\partial E}{\partial e_k^{(i+1)}}\right\} + \mathcal{R}\left\{w_{j,k}^{(i)}\right\} \frac{\partial E}{\partial e_k^{(i+1)}} \right) \\ &= \sum_j \left(w_{j,k}^{(i)} \mathcal{R}\left\{\frac{\partial E}{\partial e_k^{(i+1)}}\right\} + v_{j,k}^{(i)} \frac{\partial E}{\partial e_k^{(i+1)}} \right). \end{aligned} \tag{15}$$

Finalement, pour les poids

$$\begin{aligned}\mathcal{R}\left\{\frac{\partial E}{\partial w_{j,k}^{(i)}}\right\} &= \mathcal{R}\left\{s_j^{(i)} \frac{\partial E}{\partial e_k^{(i+1)}}\right\} \\ &= s_j^{(i)} \mathcal{R}\left\{\frac{\partial E}{\partial e_k^{(i+1)}}\right\} + \mathcal{R}\left\{s_j^{(i)}\right\} \frac{\partial E}{\partial e_k^{(i+1)}} \end{aligned} \tag{16}$$

et les biais

$$\mathcal{R}\left\{\frac{\partial E}{\partial b_k^{(i)}}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial e_k^{(i+1)}}\right\}. \tag{17}$$

Après tous ces calculs, il est possible de reconstituer le vecteur du produit hessienne-vecteur

$$\begin{aligned}\mathbf{Hv} &= \mathcal{R}\left\{\nabla_\theta E(\theta)\right\} \\ &= \left[\mathcal{R}\left\{\frac{\partial E}{\partial w_{1,1}^{(1)}}\right\} \quad \dots \quad \mathcal{R}\left\{\frac{\partial E}{\partial w_{\dim(r-2)-1,m}^{(r-1)}}\right\} \quad \mathcal{R}\left\{b_1^{(1)}\right\} \quad \dots \quad \mathcal{R}\left\{b_m^{(r-1)}\right\}\right]^\top. \end{aligned} \tag{18}$$

5 Version vectorielle

Alors que la version scalaire du réseau de neurones est utile à la compréhension, une version vectorielle est plus concise. La plupart des implémentations de cet algorithme d'apprentissage utilisent d'ailleurs la version vectorielle pour utiliser la puissance de calcul des librairies d'algèbre linéaire.

La notation vectorielle se base sur la notation scalaire à partir de laquelle les indices se rattachant à un neurone particulier ont été omis. Par exemple, l'entrée d'un neurone $\mathbf{e}^{(i)} = \left[e_1^{(i)}\ e_2^{(i)} \dots e_n^{(i)}\right]^\top$.

L'utilisation de vecteurs apporte quelques changements dont il faut tenir compte. D'abord, l'ensemble des dérivées par rapport à l'erreur est remplacé par le gradient de cette même fonction. Dans cet article, le gradient d'une fonction $f : \mathbb{R}^n \to \mathbb{R}$ est défini comme étant

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}^\top.$$

Une exception notable à cette règle (tiré du livre de Rojas [4]) est la fonction d'activation qui est une fonction de $\mathbb{R}^n$ dans $\mathbb{R}^n$, donc qui nécessite de calculer la matrice jacobienne de la fonction, c'est-à-dire que

$$\mathbf{J}_{\mathbf{e}^{(i)}}\left(a^{(i)}\right) = \begin{bmatrix} \frac{\partial a_1^{(i)}}{\partial e_1^{(i)}} & \frac{\partial a_1^{(i)}}{\partial e_2^{(i)}} & \cdots & \frac{\partial a_1^{(i)}}{\partial e_n^{(i)}} \\ \frac{\partial a_2^{(i)}}{\partial e_1^{(i)}} & \frac{\partial a_2^{(i)}}{\partial e_2^{(i)}} & \cdots & \frac{\partial a_2^{(i)}}{\partial e_n^{(i)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_n^{(i)}}{\partial e_1^{(i)}} & \frac{\partial a_n^{(i)}}{\partial e_2^{(i)}} & \cdots & \frac{\partial a_n^{(i)}}{\partial e_n^{(i)}} \end{bmatrix}.$$

Mais comme la dérivée partielle de la fonction d'activation du j-ième neurone n'a des termes qu'en fonction de l'entrée du même neurone, tous les éléments non diagonaux de la matrice $\mathbf{J}_{\mathbf{e}^{(i)}}\left(a^{(i)}\right)$ sont nuls. Donc, dans le cas où la fonction d'activation est la fonction logistique,

$$\mathbf{J}_{\mathbf{s}^{(i)}} = \mathbf{J}_{a^{(i)}_{\text{logistique}}} = \begin{bmatrix} s_1^{(i)}(1 - s_1^{(i)}) & 0 & \cdots & 0 \\ 0 & s_2^{(i)}(1 - s_2^{(i)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_n^{(i)}(1 - s_n^{(i)}) \end{bmatrix}. \quad (19)$$

Sachant ceci, il est plus efficace d'implémenter cette matrice comme un vecteur colonne qui est multiplié terme-à-terme à un autre. Par exemple, la formule (22) multiplie un gradient et une matrice jacobienne. Pour implémenter cette multiplication de manière efficace, il est possible de multiplier le gradient et le vecteur contenant les éléments diagonaux de la matrice jacobienne terme à terme. Il en résulte un nouveau vecteur de la même dimension que le gradient, ce qui correspond à la dimension du produit du gradient et de la matrice jabobienne. Cette manipulation évite de mettre en mémoire tous les zéros qui ne sont pas situés sur la diagonale de la matrice jacobienne, ce qui accélère également les calculs.

Pour profiter de ces avantages, l'opérateur $\text{diag}(A)$ est utilisé pour représenter le vecteur colonne composé des éléments de la diagonale de la matrice A. De plus, l'opérateur $\odot$ est utilisé pour signifier un produit terme à terme de deux matrices de même dimensions.

5.1 Propagation

Pour propager les entrées dans le réseau de neurones, il convient de convertir les formules vues précédemment. Dans ce contexte, une entrée du réseau de

neurones est représentée par le vecteur $x = \mathbf{s}^{(1)}$. Ensuite, pour calculer les entrées des neurones des couches suivantes, il faut transformer la formule (1) en

$$\mathbf{e}^{(i+1)} = \mathbf{w}^{(i)}\mathbf{s}^{(i)} + \mathbf{b}^{(i)}.$$

De même pour la sortie d'un neurone, la formule (2) devient

$$\mathbf{s}^{(i)} = a^{(i)}(\mathbf{e}^{(i)}) = \left[\frac{1}{1+\exp(-e_1^{(i)})} \; \frac{1}{1+\exp(-e_2^{(i)})} \; \cdots \; \frac{1}{1+\exp(-e_n^{(i)})}\right]^\top.$$

La fonction d'erreur (3) se simplifie pour donner

$$E(\theta) = \frac{1}{2}\|\mathbf{y} - \mathbf{c}\|_2^2 \tag{20}$$

dont le gradient est tout simplement

$$\nabla_{\mathbf{s}^{(r)}} E = \nabla_{\mathbf{y}} E = \mathbf{y} - \mathbf{c}. \tag{21}$$

Ces fonctions permettent de faire l'ensemble des calculs en faisant abstraction des neurones et en ne manipulant que les vecteurs ou matrices se rattachant aux couches.

5.2 Rétropropagation

Pour retrouver la version vectorielle de la rétropropagation, il faut refaire les mêmes opérations que pour la version scalaire à partir de la dernière couche. L'erreur par rapport à la sortie est déjà connue (21) et la fonction d'activation de la dernière couche est une fonction linéaire ($\mathbf{s}^{(r)} = \mathbf{e}^{(r)}$). Puisque la dérivée de cette fonction linéaire donne 1, l'entrée de la dernière couche est égale à sa sortie, c'est-à-dire que $\nabla_{\mathbf{e}^{(r)}} E = \nabla_{\mathbf{s}^{(r)}} E = y - c$.

À partir de là, il est possible de retrouver la formulation des gradients des autres couches de façon récursive. Le gradient de la sortie s'obtient, à l'image de la formule (7), par

$$\nabla_{\mathbf{s}^{(i)}} E = (\mathbf{w}^{(i)})^\top \, \nabla_{\mathbf{e}^{(i+1)}} E.$$

Quant à lui, le gradient de l'entrée d'un neurone suit le principe de la formule (8) :

$$\nabla_{\mathbf{e}^{(i)}} E = \nabla_{\mathbf{s}^{(i)}} E \, \mathbf{J}_{\mathbf{e}^{(i)}}\Big(\mathbf{s}^{(i)}\Big) = \nabla_{\mathbf{s}^{(i)}} E \odot \operatorname{diag}\left(\mathbf{J}_{\mathbf{e}^{(i)}}\Big(\mathbf{s}^{(i)}\Big)\right) \tag{22}$$

où $\mathbf{J}_{\mathbf{e}^{(i)}}\big(\mathbf{s}^{(i)}\big)$ est définie par la formule (19).

Ne reste qu'à utiliser ces deux vecteurs pour pouvoir calculer la valeur des gradients de l'erreur par rapport aux poids et aux biais, chacun décrit par les formules (9) et (10). Ils s'écrivent

$$\nabla_{\mathbf{w}^{(i)}} E = \nabla_{\mathbf{e}^{(i+1)}} E \, \nabla_{\mathbf{w}^{(i)}} \mathbf{e}^{(i+1)} = \nabla_{\mathbf{e}^{(i+1)}} E \, \mathbf{s}^{(i)\top}$$

et

$$\nabla_{\mathbf{b}^{(i)}} E = \nabla_{\mathbf{e}^{(i+1)}} E \, \nabla_{\mathbf{b}^{(i)}} \mathbf{e}^{(i+1)} = \nabla_{\mathbf{e}^{(i+1)}} E.$$

5.3 Calcul de la matrice hessienne dans une direction v

Au lieu de parler de la dérivée seconde, il est maintenant question de trouver la hessienne de la fonction d'erreur. C'est celle-ci qui permet de donner une idée du comportement de la fonction d'erreur au point d'évaluation.

Par contre, la plupart des algorithmes d'optimisation s'intéressent plutôt à la hessienne dans une direction particulière v et non à toute la matrice hessienne. En d'autres termes, au lieu de calculer la hessienne $\mathbf{H}$, il est plus efficace de calculer directement le produit hessienne-vecteur $\mathbf{Hv}$ puisque la matrice $\mathbf{H}$ n'a pas à être gardée en mémoire.

À l'image de la version scalaire où le produit hessienne-vecteur de la fonction était composé des dérivées secondes de la fonction d'erreur par rapport aux poids et aux biais, la version vectorielle manipule directement les matrices de poids et de biais.

De la même façon que la version vectorielle de la propagation des données, la version vectorielle du calcul du produit hessienne-vecteur peut s'écrire

$$\begin{aligned}\mathcal{R}\left\{\mathbf{e}^{(i+1)}\right\} &= \mathcal{R}\left\{\mathbf{w}^{(i)}\mathbf{s}^{(i)} + \mathbf{b}^{(i)}\right\} \\ &= \mathbf{w}^{(i)}\mathcal{R}\left\{\mathbf{s}^{(i)}\right\} + \mathcal{R}\left\{\mathbf{w}^{(i)}\right\}\mathbf{s}^{(i)} + \mathcal{R}\left\{\mathbf{b}^{(i)}\right\} \\ &= \mathbf{w}^{(i)}\mathcal{R}\left\{\mathbf{s}^{(i)}\right\} + \mathbf{v}_w^{(i)}\mathbf{s}^{(i)} + \mathbf{v}_b^{(i)}\end{aligned}$$

et

$$\mathcal{R}\left\{\mathbf{s}^{(i)}\right\} = \mathcal{R}\left\{\mathbf{e}^{(i)}\right\}\ \mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right).$$

La méthode reste la même pour la rétropropagation. Par contre, il faut savoir ce que représente la dérivée seconde de la sortie d'un neurone par rapport à son entrée, c'est-à-dire de trouver $\mathcal{R}\left\{\mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\right\}$. Or, comme la fonction d'activation n'a pas une image à une seule dimension, la matrice résultante serait en fait un tenseur (une matrice à plus de deux indices). Pour simplifier les choses, il est possible de se concentrer sur la deuxième forme de la formule (22) qui fait intervenir un vecteur multiplié à un autre vecteur de même dimension de façon terme-à-terme.

Dans cette notation, le j-ième élément du vecteur de la dérivée seconde est donné par

$$\operatorname{diag}\left(\mathcal{R}\left\{\mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\right\}\right)_j = s_j^{(i)}(1 - s_j^{(i)})(1 - 2s_j^{(i)})\mathcal{R}\left\{e_j^{(i)}\right\}.$$

Avec cette information, les formules (13), (14) et (15) deviennent respectivement

$$\begin{aligned}\mathcal{R}\left\{\nabla_{\mathbf{y}} E\right\} &= \mathcal{R}\left\{\mathbf{y} - \mathbf{c}\right\} \\ &= \mathcal{R}\left\{\mathbf{y}\right\}\end{aligned} \tag{23}$$

$$\begin{aligned}\mathcal{R}\left\{\nabla_{\mathbf{e}^{(i)}}\right\} &= \mathcal{R}\left\{\nabla_{\mathbf{s}^{(i)}} E\ \mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\right\} \\ &= \nabla_{\mathbf{s}^{(i)}} E\ \mathcal{R}\left\{\mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\right\} + \mathcal{R}\left\{\nabla_{\mathbf{s}^{(i)}} E\right\}\ \mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right) \\ &= \nabla_{\mathbf{s}^{(i)}} E \odot \operatorname{diag}\left(\mathcal{R}\left\{\mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\right\}\right) + \mathcal{R}\left\{\nabla_{\mathbf{s}^{(i)}} E\right\}\ \mathbf{J}_{\mathbf{e}^{(i)}}\left(\mathbf{s}^{(i)}\right)\end{aligned}$$

$$\begin{aligned}\mathcal{R}\left\{\nabla_{\mathbf{s}^{(i)}} E\right\} &= \mathcal{R}\left\{(\mathbf{w}^{(i)})^\top \nabla_{\mathbf{e}^{(i+1)}} E\right\} \\ &= (\mathbf{w}^{(i)})^\top \mathcal{R}\left\{\nabla_{\mathbf{e}^{(i+1)}} E\right\} + \mathcal{R}\left\{(\mathbf{w}^{(i)})^\top\right\} \nabla_{\mathbf{e}^{(i+1)}} E \\ &= (\mathbf{w}^{(i)})^\top \mathcal{R}\left\{\nabla_{\mathbf{e}^{(i+1)}} E\right\} + (\mathbf{v}_w^{(i)})^\top \nabla_{\mathbf{e}^{(i+1)}} E\end{aligned}$$

Encore une fois, à partir de ces résultats il est possible de reconstituer le vecteur du produit **Ht** à partir de la concaténation des éléments des matices, comme dans l'équation (18) :

$$\begin{aligned}\mathcal{R}\left\{\nabla_{\mathbf{w}^{(i)}} E\right\} &= \mathcal{R}\left\{\nabla_{\mathbf{e}^{(i+1)}} E\ (\mathbf{s}^{(i)})^\top\right\} \\ &= \nabla_{\mathbf{e}^{(i+1)}} E\ \mathcal{R}\left\{(\mathbf{s}^{(i)})^\top\right\} + \mathcal{R}\left\{\nabla_{\mathbf{e}^{(i+1)}} E\right\} (\mathbf{s}^{(i)})^\top\end{aligned}$$

et des vecteurs

$$\mathcal{R}\left\{\nabla_{\mathbf{b}^{(i)}} E\right\} = \mathcal{R}\left\{\nabla_{\mathbf{e}^{(i+1)}} E\right\}$$

qui sont dérivés des formules (16) et (17).

6 Apprentissage d'un réseau de neurones

Comme il a été mentionné plus tôt, le but de la phase d'apprentissage est de trouver le vecteur de paramètres θ défini en (4) qui minimise la fonction d'erreur comme illustré à la formule (5).

Cette section porte sur quelques façons d'y arriver, chacune présentant leurs avantages et leurs inconvénients.

6.1 Descente de gradient

Une première méthode consiste à appliquer un algorithme de descente de gradient sur l'ensemble des couples $(\mathbf{x}, \mathbf{c})$. Cette façon de procéder consiste à trouver le gradient de l'erreur de chacun des couples et d'en faire la moyenne. Comme le gradient est un vecteur pointant vers la direction de plus forte croissance de la fonction d'erreur, déplacer les paramètres dans la direction opposée au gradient fait diminuer l'erreur.

Dans cette optique, les paramètres seront mis-à-jour selon l'équation $\theta := \theta - \alpha\left(\nabla_\theta E(\theta)\right) = \alpha(\frac{1}{N}\sum_{t=1}^{N} \nabla E_t(\theta))$ où α est appelé le taux d'apprentissage

et contrôle la vitesse d'apprentissage. Un taux trop élevé pourrait nuire à l'algorithme qui sauterait par-dessus le déplacement idéal.

Selon cette méthode, l'algorithme d'apprentissage ressemble à l'algorithme 1. Dans cet algorithme, le critère d'arrêt peut être en lien avec un seuil d'erreur à atteindre ou un nombre d'itérations limite.

Algorithme 1: Algorithme de la descente de gradient pour l'apprentissage d'un réseau de neurones.

Données : Une base de données B où B_t est le t-ième couple (x, c) et dont la taille est N.

Entrées : Une fonction $CritereArret()$ qui renvoie vrai quand l'apprentissage est terminé.
Un réseau de neurones muni d'une collection de paramètres θ.
Un hyper-paramètre α donnant la vitesse d'apprentissage de l'algorithme.

tant que $CritereArret(\theta)$ **faire**

  pour chaque *couple* (x, c) *de* B **faire**

    $y \leftarrow Propagation(x)$; `/* La prédiction du réseau */`

    $g_t \leftarrow Retropropagation(y - c)$; `/* Le` t`-ième gradient */`

  $g \leftarrow \frac{1}{N} \sum_{t=1}^{N} g_t$; `/* Le gradient moyen */`

  $\theta \leftarrow \theta - \alpha g$; `/* Mise à jour des paramètres */`

  $E \leftarrow \frac{1}{N} \sum_{t=1}^{N} \|y_t - c_t\|_2^2$; `/* Erreur moyenne de prédiction */`

Sorties : Un réseau de neurones ayant des paramètres θ optimisés.

6.2 Descente de gradient par lots

Une autre méthode, l'apprentissage par lots [3] tente d'améliorer la vitesse d'apprentissage en utilisant un certain nombre de couples $(\mathbf{x}, \mathbf{c})$ par lot plutôt que l'ensemble des éléments de la base de données.

Puisque les calculs ne se font pas sur l'ensemble des données, mais seulement sur un échantillon, l'estimation de la moyenne des gradients reste sans biais alors que la variance de l'estimation de la moyenne augmente. Or, il peut être plus efficace de procéder ainsi pour avancer plus rapidement dans l'espace solution. L'algorithme 2 présente le fonctionnement de cette méthode.

Une façon d'accélérer les choses, surtout dans l'optique de l'utilisation de librairies d'algèbre linéaire optimisées, est de passer à la propagation tous les vecteurs d'entrée d'un lot à la fois, sous forme de matrice. De cette façon, les calculs peuvent tous se faire qu'en une seule propagation et rétropropagation.

Dans ce cas, un lot est un couple $(\mathbf{X}, \mathbf{C})$ où $\mathbf{X}$ et $\mathbf{C}$ sont des matrices de dimensions $n \times l$ composées de l vecteurs $\mathbf{x}$ et $\mathbf{c}$ respectivement.

À partir de ceci, chaque vecteur colonne des formules de la section 5 devient une matrice à l colonnes pour accommoder l'entrée et la cible de dimension l.

3. De l'anglais *mini-batch training*.

Algorithme 2: Algorithme de la descente de gradient pour l'apprentissage d'un réseau de neurones par lots.

Données : Une base de donnée B où B_t est le t-ième couple (x, c) et dont la taille est N.
Entrées : Une fonction $CritereArret()$ qui renvoie vrai quand l'apprentissage est terminé.
Un réseau de neurones muni d'une collection de paramètres θ.
Un hyper-paramètre α donnant la vitesse d'apprentissage de l'algorithme.
Le nombre de données l à mettre dans chaque lots.

tant que $CritereArret(E)$ **faire**
- $B_{\text{lot}} \leftarrow l$ couples (x, c) tirés au hasard et sans remise de B;
- **pour chaque** *couple* (x, c) *de* B_{lot} **faire**
  - $y \leftarrow Propagation(x)$; `/* La prédiction du réseau */`
  - $g_t \leftarrow Retropropagation(y - c)$; `/* Le` t`-ième gradient */`
- $g \leftarrow \frac{1}{N} \sum_{t=1}^{N} g_t$; `/* Le gradient moyen */`
- $\theta \leftarrow \theta - \alpha g$; `/* Mise à jour des paramètres */`
- $E \leftarrow \frac{1}{N} \sum_{t=1}^{N} \|y_t - c_t\|_2^2$; `/* Erreur moyenne de prédiction */`

Sorties : Un réseau de neurones ayant des paramètres θ optimisés.

Les formules vectorielles restent valides ; seulement la dimension des entrées/sorties et de leurs gradients changent. Toutefois, une différence est à noter au niveau de la fonction d'erreur (20) qui, pour compenser la gestion de plusieurs vecteurs à la fois, devient, selon l'article de Møller [2],

$$E = \frac{1}{2l} \|\mathbf{Y} - \mathbf{C}\|_F^2$$

où $\mathbf{Y}$ est la matrice des sorties prédites. En fait, ce calcul revient à trouver l'erreur moyenne commise sur chacun des couples de vecteurs $(\mathbf{x}, \mathbf{c})$. La dérivée de cette fonction d'erreur devient : $\frac{\partial E}{\partial \mathbf{Y}} = \frac{1}{l}(\mathbf{Y} - \mathbf{C})$. Le facteur $\frac{1}{l}$ peut ensuite être factorisé du reste des calculs, les laissant inchangés.

Pour le calcul du produit hessienne-vecteur, il faut également appliquer le facteur $\frac{1}{l}$ à la formule (23) qui devient

$$\mathcal{R}\{\nabla_{\mathbf{y}} E\} = \frac{1}{l} \mathcal{R}\{\mathbf{y}\}.$$

Grâce à ces deux petits changements, il est possible de supporter l'apprentissage par lots.

6.3 Descente de gradient stochastique

Il est également possible de pousser plus loin l'idée des lots en utilisant des lots de taille 1. Pour ce faire, la méthode consiste à prendre chaque couple

$(\mathbf{x}, \mathbf{c})$ individuellement et à appliquer les formules de la section 5. Cette façon de faire est appelée l'apprentissage en ligne [4] lorsqu'elle se fait dans l'ordre des données ou apprentissage par descente de gradient stochastique lorsque le couple en entrée est choisi au hasard.

6.4 Méthodes d'optimisation de second ordre

Une autre possibilité est d'utiliser des algorithmes d'optimisation requierant des informations sur la courbure de la fonction d'erreur, c'est-à-dire sur la matrice hessienne de E. Pour aider ces algorithmes, une méthode semblable à la rétropropagation a été exposée dans cet article pour trouver le produit de la matrice hessienne et d'un vecteur. Ce produit est souvent utilisé dans ces techniques pour observer la courbure de la fonction d'erreur dans une direction donnée, ce qui ne nécessite pas le calcul de la hessienne en entier.

Bien entendu, il est possible de combiner ces méthodes avec la philosophie des lots afin d'accélérer l'apprentissage.

Références

[1] C. M. Bishop. *Pattern recognition and machine learning.* Springer New York, 2006.

[2] M. Møller. Supervised learning on large redundant training sets. In *Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop*, pages 79–89. IEEE, 1992.

[3] B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1) :147–160, 1994.

[4] R. Rojas. *Neural networks : a systematic introduction.* Springer, 1996.

[5] A. Salvail-Bérard. Les arbres de décision hybrides. *Cahier de Mathématique de l'Université de Sherbrooke*, 2 :34–58, 2012.

4. De l'anglais *online training.*