Introduction aux réseaux de neurones et à l'apprentissage profond.

Julie Delon

21 août 2018

Table des matières

Notations

On s'intéresse dans ce cours à des problèmes d'apprentissage supervisé, typiquement de régression ou de classification. Le but est d'estimer des fonctions de régression ou de classification à partir de données. On utilisera des indices supérieurs avec des parenthèses pour indiquer un ensemble de données d'observations. Par exemple, pour un problème de régression à partir de M points dans $\mathbb{R}^2$, on notera les données observées $\{(x^{(m)}, y^{(m)})\}_{m=1,\dots,M}$. Les indices inférieurs seront utilisés pour indiquer les coordonnées des données dans l'espace. Par exemple si la donnée $x^{(1)}$ vit dans $\mathbb{R}^p$, on notera ses coordonnées $(x_1^{(1)}, \dots, x_p^{(1)})$.

1 Introduction

1.1 Exemple introductif

Imaginons que vous travaillez pour une banque et que vous deviez construire un modèle prédisant la somme S des dépenses annuelles que va faire un client pendant l'année en fonction d'un certain nombre d'informations $x = (x_1, x_2, \ldots, x_p)$ sur le client, comme par exemple son âge, ses revenus, la somme présente sur son compte en début d'année, son métier, etc. Pour déduire ce modèle, vous vous reposez sur une base d'apprentissage de plusieurs clients pour laquelle vous avez toutes ces informations à disposition.

Si vous utilisez un modèle de régression linéaire pour la prédiction, la somme S des dépenses en fonction des données x s'écrira

$$S(x) = \sum_{i=1}^{p} w_i x_i.$$

Si on regarde par exemple la somme S comme une fonction du seul revenu x_1, elle sera représentée par une droite, comme illustré par la Figure 1. Changer une autre donnée x_k (comme l'âge du client) changera l'ordonnée à l'origine de cette droite, mais pas sa pente. Or il n'y a aucune raison que cette pente soit indépendante des autres données à disposition. Une des limitations de ce modèle est qu'il ne tient pas compte des interactions entre les données.

Une manière de tenir compte de ces interactions entre les données d'entrée est d'introduire des fonctions intermédiaires non linéaires qui considèrent toutes les données à la fois, et de faire dépendre ces fonctions de paramètres à ajuster en fonctions des données d'apprentissage. Il y a de nombreuses manières de construire de telle fonctions, et les réseaux de neurones en proposent une particulièrement simple à écrire et surtout bien adaptée aux données complexes et de grande dimension. Ils alternent pour cela des combinaisons linéaires des données (dont les poids doivent être appris) avec des fonctions non linéaires fixées, sur plusieurs couches intermédiaires (dites *cachées*) pour tenir compte des interactions entre les données d'entrée (voir la Figure 2 pour une représentation graphique d'un réseau à deux couches). Plus le réseau a de couches, plus il est profond.

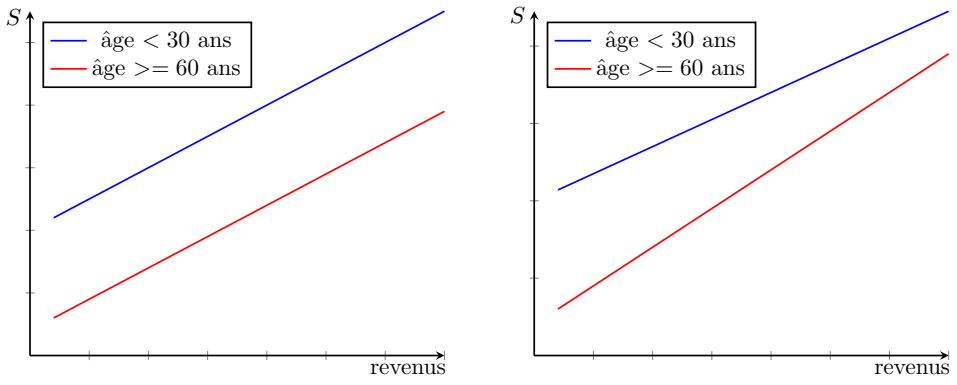

FIGURE 1 – A gauche, modèle dans lequel on ne tient pas compte des interactions entre les données (ici l'âge et le revenu). On considère que si l'âge change, la pente de la droite ne change pas. A droite, modèle tenant compte des interactions.

Parmi les succès les plus impressionnants des réseaux de neurones profonds, on peut citer l'exemple du *Cloud Vision* de Google, qui permet d'analyser et comprendre le contenu d'une image avec une efficacité redoutable. Tous les acteurs majeurs de la vente en ligne (Netflix, Spotify) utilisent le deep learning pour apprendre les préférences de leurs clients et leur proposer des recommandations personnalisées. Le programme *AlphaGo*, développé par Google DeepMind, est capable de jouer au jeu de Go et de battre les meilleurs joueurs humains. Il est aussi construit à partir de réseaux de neurones profonds.

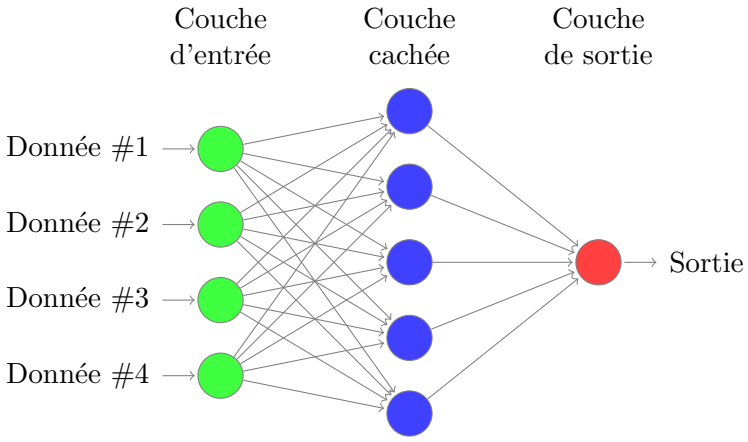

FIGURE 2 – Réseau de neurones à deux couches (couche intermédiaire + sortie).

1.2 Limites des modèles linéaires pour l'apprentissage supervisé

Les deux grandes familles de problèmes de *l'apprentissage supervisé* sont la *régression supervisée* et la *classification supervisée*.

Etant donné une base d'apprentissage constituée de M vecteurs d'observations $\{x^{(m)}\}_{m=1,\dots,M}$ vivant dans $\mathbb{R}^p$ et de vecteurs cibles $\{y^{(m)}\}_{m=1,\dots,M}$, le but de la régression supervisée est de construire une fonction $y(x)$ permettant de prédire les valeurs de y en fonction de la donnée d'un nouveau vecteur de données x, ou plus subtilement de modéliser la distribution $p(y|x)$. Dans le cas de la classification, on connaît pour chaque vecteur $x^{(m)}$ sa classe $\mathcal{C}_k$ (parmi K classes connues), et le but est de pouvoir classer n'importe quel nouveau vecteur de données x dans une des classes $\mathcal{C}_k$, $k = 1, \dots, K$.

Les **modèles linéaires pour la régression ou la classification** peuvent s'écrire de manière générale sous la forme

$$y(x; w) = f\left(\sum_{i=0}^{p} w_i x_i\right) = f\left(w^T x\right) \tag{1}$$

où

- on pose $x_0 = 1$ pour éventuellement tenir compte d'un *offset*
- f est l'identité dans le cas de la régression et une *fonction d'activation* non linéaire fixée dans le cas de la classification (qui sera seuillée pour trouver les classes).

Dans les deux cas (régression et classification linéaires supervisées), l'apprentissage supervisé consiste à ajuster les poids w_i pour faire coller le modèle (1) "le mieux possible" avec

la base d'apprentissage, par exemple en minimisant une fonction de perte sur l'ensemble des données.

Ces modèles linéaires ont de bonnes propriétés et sont simples à optimiser, donnant lieu le plus souvent à des problèmes d'optimisation convexes. Ils restent cependant trop figés et représentent mal les données complexes ou vivant en grande dimension. Une solution pour rendre ce modèle plus général est de remplacer x par une représentation $\phi(x)$ où ϕ est une transformation non linéaire. Le choix de ϕ est cependant très difficile et dépend fortement des applications considérées. Les réseaux de neurones permettent de répondre simplement à cette limitation en représentant ϕ par une classe assez générale de fonctions dépendant de paramètres, et en permettant d'apprendre les paramètres optimaux pour ϕ sur une base d'apprentissage.

Exercice 1 *Considérons un problème de régression logistique pour de la classification binaire. On pose*

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

et on cherche les paramètres $w = (w_0, \dots, w_p)$ permettant de faire coller le modèle (1) le mieux possible avec les données observées $(x^{(1)}, y^{(1)}), \dots (x^{(m)}, y^{(m)})$. On propose pour cela de minimiser la fonction de perte

$$\mathcal{L}(w) = -\frac{1}{M} \sum_{m=1}^{M} y^{(m)} \log \hat{y}^{(m)} + (1 - y^{(m)}) \log(1 - \hat{y}^{(m)})$$

avec $\hat{y}^{(m)} = f\left(w^T x^{(m)}\right)$. Calculer le gradient de $\mathcal{L}$ par rapport à w. Proposer un algorithme pour résoudre numériquement le problème de minimisation

$$\min_w \mathcal{L}(w).$$

Exercice 2 *Montrer qu'un modèle de régression linéaire n'est pas approprié pour approcher la fonction binaire* $\mathrm{XOR}(x_1, x_2)$*, qui prend la valeur* 1 *uniquement si une des deux valeurs binaires x_1, x_2 vaut* 1*, et la valeur* 0 *sinon.*

2 Réseaux de neurones : modélisation

Le but d'un réseau de neurones est de représenter une fonction $\widehat{y}(x; w)$ (plus générale que dans le modèle linéaire) pour la régression ou la classification et d'apprendre la valeur du vecteur de paramètres w permettant une bonne prédiction sur une base d'apprentissage donnée. On s'intéresse ici aux réseaux *feedforward* (non-bouclés), pour lesquels il n'y a pas de connexion retour de la sortie vers l'entrée ou vers les couches intermédiaires. De tels réseaux sont d'une importance considérable aujourd'hui dans de nombreuses applications commerciales ou académiques.

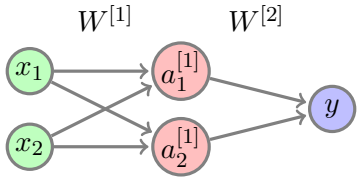

FIGURE 3 – Exemple de réseau de neurones à deux couches.

2.1 Exemple de réseau à deux couches

Commençons par un exemple très simple de réseau *feedforward* à deux couches, représenté par la Figure 3. Ce réseau a une unique couche intermédiaire, on va supposer typiquement que ces valeurs s'écrivent :

$$a_i^{[1]} = g(W_{i1}^{[1]} x_1 + W_{i2}^{[1]} x_2), \quad i = 1, 2$$

avec g une fonction non linéaire, par exemple la fonction **Rectified Linear Unit** ou ReLU (voir la Figure 4)

$$\text{ReLU}(x) = max(0, x). \tag{2}$$

On suppose que la sortie y est calculée à partir de la couche intermédiaire comme

$$y = W_1^{[2]} a_1^{[1]} + W_2^{[2]} a_2^{[1]}.$$

La fonction complète représentée par ce réseau est donc la suivante :

$$\widehat{y}(x) = \sum_{i=1}^{2} W_i^{[2]} \, g\left(\sum_{k=1}^{2} W_{ik}^{[1]} x_k \right) = W^{[2]} max(0, W^{[1]} x),$$

et elle dépend des paramètres $W^{[1]} = \begin{pmatrix} W_{11}^{[1]} & W_{12}^{[1]} \\ W_{21}^{[1]} & W_{22}^{[1]} \end{pmatrix}$ et $W^{[2]} = \begin{pmatrix} W_1^{[2]} & W_2^{[2]} \end{pmatrix}$. Le numéro entre crochets dans la notation $W^{[1]}$ désigne la couche sur laquelle on se trouve.

Exercice 3 *Montrer que la fonction binaire* $\text{XOR}(x_1, x_2)$ *peut être très simplement obtenue par le réseau précédent en choisissant bien ses différents paramètres. La solution est-elle unique ?*

Correction Plusieurs solutions sont possibles. Une solution est de prendre $W = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$, $\theta = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Représenter la fonction binaire (XOR) par un réseau à deux couches n'est pas très complexe et une solution peut être trouvée intuitivement. Dans les cas réels, on peut avoir des données massives et de grande dimension, que l'on cherche à approcher par un réseau profond dépendant de plusieurs millions de paramètres. Dans ce type de cas, déterminer le bon jeu de paramètres est donc beaucoup moins évident.

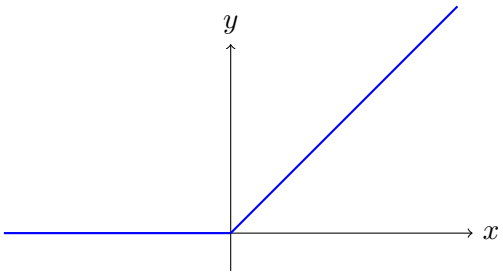

FIGURE 4 – La fonction $ReLU(x) = \max(0, x)$. Cette fonction est recommandée comme fonction d'activation par défaut dans les réseaux de neurones.

2.2 Architecture du réseau et *Forward Propagation*

Un réseau de neurone de type *feedforward* enchaîne différentes couches de neurones, chaque couche étant fonction de la précédente. Si le vecteur d'entrée est $x \in \mathbb{R}^p$, la première couche effectue d'abord une opération linéaire ou affine sur x, puis calcule une fonction non linéaire du résultat :

$$z^{[1]} = W^{[1]}x + b^{[1]} \quad \text{puis} \quad a^{[1]} = g^{[1]}(z^{[1]}) \tag{3}$$

avec $W^{[1]}$ une première matrice de poids de taille $n^{[2]} \times p$, $b^{[1]}$ un vecteur d'offset de taille $n^{[2]} \times 1$ et $g^{[1]}$ une fonction d'activation, généralement non linéaire (lorsqu'on écrit $g^{[1]}(z)$ avec z un vecteur, cela signifie que $g^{[1]}$ est appliquée à toutes les coordonnées de z). La k^{eme} couche s'écrit alors en fonction de la précédente comme

$$a^{[k]} = g^{[k]}\underbrace{\left(W^{[k]}a^{[k-1]} + b^{[k]}\right)}_{z^{[k]}}, \tag{4}$$

avec $W^{[k]}$ une matrice de poids de taille $n^{[k+1]} \times n^{[k]}$, $b^{[k]}$ un vecteur d'offset de taille $n^{[k+1]} \times 1$, et ainsi de suite jusqu'à la couche de sortie l

$$\widehat{y}(x; W; b) = g^{[l]}\left(W^{[l]}a^{[l-1]} + b^{[l]}\right). \tag{5}$$

Fonctions d'activation. Les fonctions $g^{[k]}$ sont appliquées sur les noeuds intermédiaires du réseau. Elle permettent au modèle de capturer des non linéarités dans les relations entre les données. Voici quelques exemples de fonctions d'activation que l'on trouve dans la littérature

- la fonction tanh et la fonction logistique $\sigma(z) = \frac{1}{1+e^{-z}}$ ont été très longtemps plébiscitées comme fonctions d'activation pour les réseaux de neurones ; ces deux fonctions présentent le défaut d'avoir une dérivée quasiment nulle dès que z s'éloigne trop de 0, on verra que cela peut être un handicap pour entraîner le réseau ;
- la fonction $ReLU(x) = \max(x, 0)$ est aujourd'hui considérée comme la fonction d'activation standard à la fois dans l'industrie et dans la recherche académique, mais de très nombreux autres choix sont possibles et fonctionnent bien selon les applications, l'idée étant de garder des fonctions suffisamment simples pour pouvoir optimiser le réseau rapidement ;
- la fonction $ReLU$ peut parfois être avantageusement remplacée par une fonction du type $\max(x, 0) + \epsilon \min(x, 0)$ avec ϵ petit ;

— La fonction valeur absolue $|x|$ est utilisée lorsque l'on souhaite modéliser une invariance par changement de signe (comme par exemple en reconnaissance d'objets, pour approcher une invariance en fonction des conditions d'illumination).

Forward Propagation. Le processus consistant à évaluer la fonction $\widehat{y}(x; W; b)$ dans un réseau *feedforward* est appelé *Forward Propagation*, puisqu'il consiste à propager l'information de gauche à droite dans le réseau. C'est ce processus qui permet à un réseau de neurones de faire des prédictions à partir de données.

Justification biologique. On parle de réseaux de **neurones** car ces réseaux sont en partie inspirés par les neurosciences, chaque noeud du réseau jouant un rôle « analogue » à celui d'un neurone dans le cerveau. Si de nombreux articles font des parallèles entre les deux, la comparaison a ses limites : les réseaux de neurones que l'on étudie et utilise actuellement en informatique sont extrêmement simplistes si on les compare au cerveau humain... L'intelligence artificielle n'est pour l'instant pas aussi intelligente que les médias veulent le croire.

2.3 Propriété d'approximation universelle et réseaux profonds

Le théorème suivant [1, 2] dit qu'un réseau feedforward à deux couches avec une sortie linéaire et une fonction d'activation g bornée croissante, continue (non constante) peut approcher aussi finement que l'on veut n'importe quelle fonction continue sur l'hypercube $[0,1]^D$ du moment qu'il a suffisamment de neurones sur sa couche intermédiaire.

Théorème 1 (Théorème d'approximation universel) *Soit g une fonction non constante, bornée et croissante et continue. L'ensemble des fonctions sur l'hypercube $[0,1]^D$ de la forme*

$$y(x) = \theta^T g\left(Wx + b\right)$$

avec $N \in \mathbb{N}$, $W \in \mathcal{M}_{N\times D}(\mathbb{R})$, $\theta \in \mathbb{R}^N$ et $b \in \mathbb{R}^N$, est dense dans l'ensemble des fonctions continues sur $[0,1]^D$.

Sous des conditions un peu plus fortes sur g, on peut approcher n'importe quelle fonction mesurable sur l'hypercube. En d'autres termes, on peut approcher aussi finement que l'on veut n'importe quelle fonction sur un compact en dimension finie avec un réseau de neurones à deux couches suffisamment grand. Le problème est que la taille de la couche intermédiaire nécessaire peut-être tellement grande que le réseau n'est pas utilisable ou optimisable en pratique. Or, en pratique, il est souvent beaucoup plus utile d'utiliser des réseaux moins larges mais plus profonds.

Deep learning. Pendant longtemps, on se contentait d'utiliser des réseaux de neurones à quelques couches, donc peu profonds, pour des raisons de temps de calcul. On se contentait en général d'une quinzaine de couches au maximum. Les réseaux de neurones modernes doivent un partie de leur succès retentissant dans de nombreux domaines (image, son, vidéo, médecine, etc) au fait qu'ils utilisent de très nombreuses couches, on parle alors d'apprentissage profond (ou deep learning). Certains de ces réseaux peuvent avoir des milliers de couches, et des millions de neurones. Pour entraîner de tels réseaux, qui dépendent de milliers ou dizaines de milliers de paramètres, il faut des bases massives de données d'entraînement. Le succès

actuel des réseaux de neurones profonds est en grande partie dû à l'utilisation de ces immenses masses de données.

Les réseaux de neurones et en particulier les réseaux profonds permettent de construire des fonctions extrêmement sophistiquées des données d'entrée, à partir des paramètres entre les noeuds du réseau. On parle aussi de *representation learning*. Ces paramètres sont appris automatiquement à partir d'une base d'apprentissage. Cette délicate étape d'optimisation est l'objet de la prochaine section.

3 Optimisation du réseau

On explique dans cette section comment estimer les paramètres W du réseau. Etant donnée une base d'apprentissage composée de N vecteurs d'entrée $\{x^{(m)}\}_{m=1,\dots,M}$ et d'un ensemble correspondant de vecteurs cibles $\{y^{(m)}\}_{m=1,\dots,M}$, on cherche les poids qui minimisent une fonction de perte (*loss function* en anglais) agrégeant les erreurs de prédiction sur la base.

Une fonction de perte classique est l'erreur quadratique moyenne :

$$\mathcal{L}(W) = \frac{1}{2}\sum_{m=1}^{M} \|\widehat{y}(x^{(m)}, W) - y^{(m)}\|^2.$$

Cette fonction de perte quadratique apparaît naturellement dans un problème de régression si on suppose par exemple que les $x^{(m)}$ sont des réalisations de variables i.i.d., que chaque $y^{(m)}$ suit une distribution gaussienne $\mathcal{N}(\widehat{y}(x^{(m)}, W), \sigma^2)$ et que l'on cherche les paramètres par maximum de vraisemblance. En effet, le maximum de vraisemblance est calculé dans ce cas comme

$$\text{argmax}_W \Pi_{m=1}^{M} e^{-\frac{\|y^{(m)} - \widehat{y}(x^{(m)}, W)\|^2}{2\sigma^2}},$$

ce qui revient à minimiser $\mathcal{L}(W)$.

D'autres fonctions de perte beaucoup plus générales peuvent être utilisées selon le type de problème à résoudre. Typiquement, pour des problèmes de classification, on utilisera plutôt une fonction d'entropie croisée

$$\mathcal{L}(W) = -\sum_{m=1}^{M} \left(y^{(m)} ln \widehat{y}(x^{(m)}, W) + (1 - y^{(m)}) ln(1 - \widehat{y}(x^{(m)}, W)) \right).$$

Pour ce type de problème, la couche de sortie a généralement un noeud séparé pour chaque classe et utilise une fonction d'activation de type 'softmax', permettant que la somme des sorties soit 1.

Exercice 4 *Dans un problème de classification à deux classes C_0 et C_1, on suppose que les $x^{(m)}$ sont des réalisations de variables i.i.d. et que la sortie $\widehat{y}(x^{(m)}, W)$ représente la probabilité que la classe de $x^{(m)}$ soit C_1, autrement dit*

$$\mathbb{P}[y^{(m)} = 1 | x^{(m)}, W] = \widehat{y}(x^{(m)}, W).$$

Quelle fonction de perte trouve t'on si on cherche les poids W par maximum de vraisemblance ?

Dans le cas d'un problème de classification à plus de deux classes, il est usuel de définir la fonction de perte de la manière suivante. Si $\widehat{y} = (\widehat{y}_1 \dots, \widehat{y}_3)$ est la sortie du réseau de neurones :

$$\mathcal{L}(\widehat{y}, y) = \sum_{m=1}^{M} -\log\left(\frac{e^{\widehat{y}_{y(m)}}}{e^{\widehat{y}_1} + \cdots + e^{\widehat{y}_3}}\right).$$

Remarquons qu'en général, la fonction de perte $\mathcal{L}(W)$ n'est pas une fonction convexe des poids, elle peut donc être très difficile à optimiser. En pratique, on l'optimise quand même par descente de gradient, en faisant donc *comme si* elle était convexe.

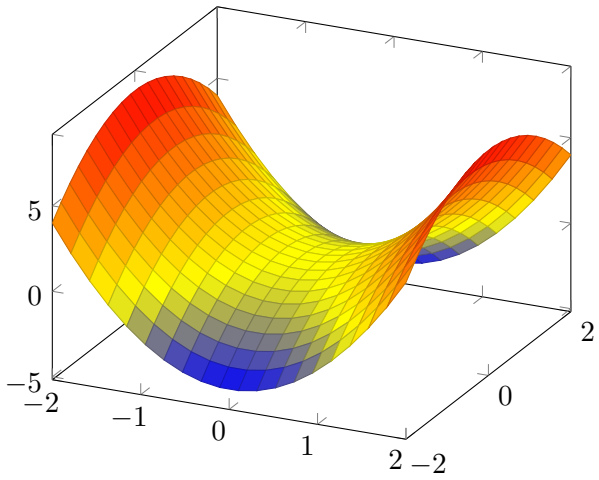

FIGURE 5 – Exemple de fonction de perte non convexe en les poids w.

Exercice 5 ***Symétries du réseau.*** *Pour un réseau à 1 couche, M neurones intermédiaires et une fonction d'activation impaire, montrer que la fonction de perte quadratique $\mathcal{L}(W)$ présente M axes de symétrie (on peut remplacer un poids par son inverse et compenser ce changement de signe sur l'arête suivante) et que chaque vecteur de poids a donc 2^M vecteurs de poids qui lui sont équivalents. Montrer également qu'en échangeant les arêtes entre elles, la fonction représentée par le réseau ne change pas, ce qui implique également que chaque vecteur de poids a $M!$ vecteurs de poids qui lui sont équivalents de cette manière. En déduire que chaque vecteur de poids a en fait $M!2^M$ vecteurs équivalents.*

3.1 Descente de gradient

L'optimisation du réseau se fait par descente de gradient, avec les étapes suivantes :

1. On commence par initialiser les poids W aléatoirement ;
2. On calcule le gradient de la fonction de perte $\mathcal{L}$ par rapport à tous les poids du réseau

$$\nabla\mathcal{L} = \left\{\frac{\partial\mathcal{L}}{\partial W_{ji}^{[k]}}\right\}_{i,j,k}$$

en utilisant la dérivation en chaîne (*backpropagation*, voir la section 3.3) ;

3. Pour un pas de descente $\alpha > 0$ donné, on met à jour les poids W

$$W_{ji}^{[k]} \longleftarrow W_{ji}^{[k]} - \alpha \frac{\partial \mathcal{L}}{\partial W_{ji}^{[k]}}$$

et on retourne à l'étape 2, jusqu'à convergence.

On peut aussi utiliser un algorithme de gradient conjugué ou du quasi-newton pour cette optimisation, *ce qui permet d'assurer que la fonction de perte décroît à chaque itération* (ce n'est pas le cas avec une descente de gradient simple). Comme la fonction $\mathcal{L}$ n'est pas convexe en W, il peut être nécessaire de relancer l'algorithme avec plusieurs initialisations différentes et de choisir le résultat optimal. Le calcul du gradient de $\mathcal{L}$, s'il peut être effectué directement pour des réseaux très simples, nécessite en général un algorithme spécifique, appelé *Backpropagation*. Nous le verrons en Section 3.3.

3.2 Descente de gradient stochastique

Pour entraîner les réseaux de neurones sur de très grandes bases de données, on peut utiliser une méthode de descente de gradient séquentielle, aussi appelée *descente de gradient stochastique*. Le principe est de décomposer la fonction d'erreur sous la forme

$$\mathcal{L}(w) = \sum_{n=1}^{N} \mathcal{L}_n(w)$$

où l'indice $n = 1 \dots N$ désigne différentes observations (ou groupes d'observations) indépendantes (on parle aussi de batch de données). Un pas de descente s'écrit

$$W^{k+1} = W^k - \alpha \nabla \mathcal{L}_n(w^k).$$

Cette mise à jour est répétée soit en choisissant n au hasard, soit dans un ordre prédéfini. On cycle sur les valeurs de n et on recommence quand les N données ont toutes été utilisées. Chaque cycle sur l'ensemble des données est appelé *une époque*.

On peut réinterpréter cette méthode comme une descente avec une version bruitée du gradient à chaque étape. Si on suppose que $\nabla \mathcal{L}(w)$ varie lentement, une époque est quasiment équivalente à une descente dans la direction $-\nabla \mathcal{L}(w)$ avec le poids α.

Ce type de méthode a la bonne propriété de pouvoir échapper au minima locaux, puisqu'un point stationnaire par rapport à L n'est généralement pas stationnaire pour $\mathcal{L}_n$. Elle s'adapte aussi bien à l'apprentissage *online*, pour lequel les données ne sont pas toutes disponibles en même temps.

3.3 Calcul du gradient par *Backpropagation*

Pour calculer efficacement le gradient $\nabla \mathcal{L}$, on utilise l'algorithme dit de *backpropagation*. Cet algorithme part de l'erreur de prédiction et la repropage à l'envers dans le réseau vers les valeurs d'entrée pour calculer les dérivées par rapport à chaque poids.

La *backpropagation* repose sur le principe de dérivation en chaîne. On rappelle que si l'on a une fonction f de $\mathbb{R}^n$ dans $\mathbb{R}$ et une fonction h de $\mathbb{R}^p$ dans $\mathbb{R}^n$ et que l'on note $L(x) = f \circ h(x)$ (avec h_j la cordonnée j de h), alors

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial h_j} \frac{\partial h_j}{\partial x_i}.$$

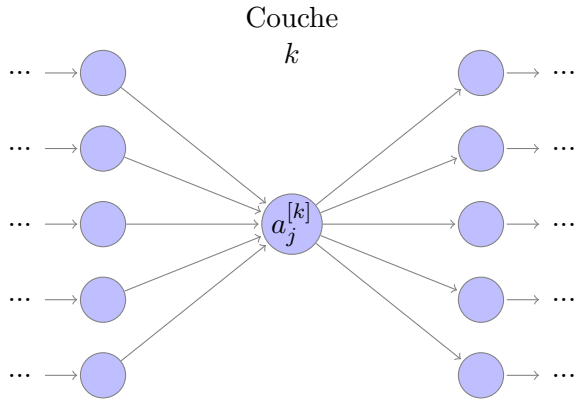

FIGURE 6 – Trois couches intermédiaires successives d'un réseau de neurones.

Dans ce qui suit, on fera un léger abus de notation et on écrira $\frac{\partial L}{\partial h_j}$ pour désigner le gradient $\frac{\partial f}{\partial h_j}$. La dérivation en chaîne s'écrira donc

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial h_j} \frac{\partial h_j}{\partial x_i},$$

où L est vue comme fonction de x dans la partie gauche de l'équation et comme fonction de h dans la partie droite.

Gardons les notations de la Section 2.2, et supposons pour simplifier que la fonction d'activation soit la même sur toutes les couches, on la note g. Notre but est de calculer, pour toutes les couches k du réseau, la dérivée de $\mathcal{L}$ par rapport aux poids $W_{ji}^{[k]}$ et aux offsets b_j^k.

On fait l'hypothèse pour l'instant que la fonction de perte est quadratique

$$\mathcal{L}(W) = \frac{1}{2} \sum_{m=1}^{M} \|\widehat{y}(x^{(m)}, W) - y^{(m)}\|^2.$$

En dérivant, on a pour tout i, j, k

$$\frac{\partial \mathcal{L}}{\partial W_{ji}^{[k]}} = \sum_{m=1}^{M} < \widehat{y}(x^{(m)}, W) - y^{(m)}, \frac{\partial \widehat{y}}{\partial W_{ji}^{[k]}} > . \tag{6}$$

Par exemple, pour la dernière couche $k = l$, on sait que $\widehat{y}(x, W) = g(z^{[l]})$ avec

$$z_j^{[l]} = \sum_i W_{ji}^{[l]} a_i^{[l-1]} + b_j^{[l]}.$$

On peut donc déjà calculer

$$\frac{\partial \widehat{y}}{\partial W_{ji}^{[l]}} = g'(z^{[l]}) \frac{\partial z_j^{[l]}}{W_{ji}^{[l]}} = g'(z^{[l]}) a_i^{[l-1]},$$

en gardant en tête que $z_j^{[l]}$ et $a_i^{[l-1]}$ sont ici des fonctions de $(x^{(m)}, W)$. Ainsi, pour tous les poids de la dernière couche l,

$$\frac{\partial \mathcal{L}}{\partial W_{ji}^{[l]}} = \sum_{m=1}^{M} < \widehat{y}(x^{(m)}, W) - y^{(m)}, g'(z^{[l]}(x^{(m)}, W)) a_i^{[l-1]}(x^{(m)}, W) > .$$

Regardons maintenant comment calculer $\frac{\partial \mathcal{L}}{\partial W_{ji}^{[k]}}$ pour les autres couches du réseau. Grâce à l'équation (6), on sait qu'il suffit de calculer $\frac{\partial \widehat{y}}{\partial W_{ji}^{[k]}}$. Remarquons que la sortie $\widehat{y}(x, W)$ de dépend des poids $W_{ji}^{[k]}$ et de $b_j^{[k]}$ qu'à travers la valeur de $z_j^{[k]}$. En d'autres termes, il existe f telle que

$$\widehat{y}(W) = f(z_j^{[k]}, \{W_{j'i'}^{[k']}, b_{j'}^{[k']}\}_{(i',j',k') \neq (i,j,k)}).$$

On rappelle que

$$z_j^{[k]} = \sum_i W_{ji}^{[k]} a_i^{[k-1]} + b_j^{[k]}.$$

Avec le même abus de notation que plus haut, la dérivée de $\widehat{y}$ par rapport au poids $W_{ji}^{[k]}$ et à l'offset $b_j^{[k]}$ s'écrit donc

$$\frac{\partial \widehat{y}}{\partial W_{ji}^{[k]}} = \frac{\partial \widehat{y}}{\partial z_j^{[k]}} \frac{\partial z_j^{[k]}}{\partial W_{ji}^{[k]}} = \frac{\partial \widehat{y}}{\partial z_j^{[k]}} a_i^{[k-1]}, \quad \text{et} \quad \frac{\partial \widehat{y}}{\partial b_j^{[k]}} = \frac{\partial \widehat{y}}{\partial z_j^{[k]}}.$$

Comme les valeurs $a_i^{[k-1]} = g(z_i^{[k-1]})$ sont connues, ces équations permettent de calculer le gradient de $\widehat{y}$ et donc de $\mathcal{L}$ par rapport aux paramètres du réseau (les W et les b) très simplement si on connaît déjà le gradient de $\widehat{y}$ par rapport aux valeurs $z_j^{[k]}$.

Montrons maintenant que calculer les dérivées partielles $\frac{\partial \widehat{y}}{\partial z_j^{[k]}}$ se fait également très simplement en propageant les valeurs de la couche de sortie vers la couche d'entrée. Remarquons que $\widehat{y}$ ne dépend de $z_j^{[k]}$ qu'à travers les valeurs $z_l^{[k+1]}$ des neurones de la couche $k+1$ à laquelle le neurone $z_j^{[k]}$ est connecté. Or,

$$z_l^{[k+1]} = \sum_i W_{li}^{[k]} g(z_i^{[k]}).$$

On en déduit que

$$\frac{\partial \widehat{y}}{\partial z_j^{[k]}} = \sum_l \frac{\partial \widehat{y}}{\partial z_l^{[k+1]}} \frac{\partial z_l^{[k+1]}}{\partial z_j^{[k]}} = \sum_l \frac{\partial \widehat{y}}{\partial z_l^{[k+1]}} W_{lj}^{[k+1]} g'(z_j^{[k]}).$$

Ainsi, si on connaît toutes les dérivées partielles $\frac{\partial \widehat{y}}{\partial z_l^{[k+1]}}$ pour la couche $k+1$, on peut calculer toutes les dérivées partielles $\frac{\partial \widehat{y}}{\partial z_j^{[k]}}$ pour la couche k par simple combinaison linéaire. L'algorithme 1 est linéaire en le nombre de neurones du réseau. Le même type d'algorithme peut être utilisé pour calculer des dérivées d'ordre supérieur dans le réseau (comme des hessiennes par exemple).

4 Régularisation

Le nombre d'entrées et de sorties dans un réseau est généralement déterminé par les données d'apprentissage, mais le nombre total M de neurones des couches intermédiaires et le nombre de ces couches est un paramètre qui doit être ajusté pour éviter à la fois l'over-fitting et l'under-fitting.

Algorithm 1 Algorithme de *Backpropagation*

1. Pour un vecteur d'entrée $x^{(m)}$ de la base d'apprentissage, appliquer l'algorithme de *forwardpropagation* pour calculer toutes les valeurs des neurones du réseau.
2. Pour k décroissant de K à 1, calculer les dérivées partielles de $\mathcal{L}$ par rapport aux noeuds de la couche k.
3. En déduire les dérivées partielles de $\mathcal{L}$ par rapport à chacun des paramètres du réseau.

Si le nombre M est choisi très grand, le risque d'over-fitting l'est aussi. Une manière d'éviter l'over-fitting est alors d'ajouter à la fonction de perte $\mathcal{L}$ un terme de régularisation sur les poids. Typiquement, on peut choisir une régularisation quadratique ou L^1.

4.1 Sur et sous-ajustement (*Overfitting* et *underfitting*)

Le but de l'apprentissage automatique est d'inférer un modèle à partir d'une base d'apprentissage, de manière à ce que ce modèle soit également bien adapté à de nouvelles données si elles suivent les mêmes lois que les données d'apprentissage. Cette capacité, appelée capacité de généralisation, est mesurée par l'erreur de généralisation de la méthode considérée.

Notons $\mathcal{D}_{train} = \{(x^{(m)}, y^{(m)})\}_{m=1...,M}$ et $\mathcal{D}_{test} = \{(x^{(m)}, y^{(m)})\}_{m=M+1...,M_2}$ deux bases d'entraînement et de test, dont on suppose qu'elles sont composées de réalisations indépendantes de la même distribution $p(x, y)$. Le réseau est entraîné pour minimiser l'erreur d'entraînement

$$\mathcal{L}_{train}(W) = \frac{1}{M} \sum_{m=1}^{M} \mathcal{L}(\hat{y}(x^{(m)}; W), y^{(m)}) \tag{7}$$

avec $\hat{y}(x; W)$ la sortie du réseau au point x pour les paramètres W. On définit l'erreur de test par

$$\mathcal{L}_{test}(W) = \frac{1}{M_2 - M + 1} \sum_{m=M+1}^{M_2} \mathcal{L}(\hat{y}(x^{(m)}; W), y^{(m)}). \tag{8}$$

Le but est donc de choisir les paramètres W afin que $\mathcal{L}_{train}(W)$ soit petit et que $\mathcal{L}_{test}(W)$ soit du même ordre de grandeur (voir la Figure 7).

— Si $\mathcal{L}_{train}(W)$ est grand, on dit que le réseau est sous-ajusté, insuffisamment complexe pour représenter les données ;

— Si la différence $|\mathcal{L}_{test}(W) - \mathcal{L}_{train}(W)|$ est grande, le réseau est sur-ajusté, il colle trop aux données d'entraînement et peu généralisable.

La régularisation est introduite dans l'algorithme pour réduire l'erreur de généralisation sans perturber l'erreur d'apprentissage. Différents types de régularisation peuvent être envisagées :

1. La pénalisation d'une norme des paramètres dans la fonction de perte $\mathcal{L}_{train}(W)$. On pénalise seulement les poids W, pas les offsets b. On détaille quelques exemples dans les sections suivantes.
2. La restriction de l'espace des paramètres : on peut par exemple imposer certaines symétries sur W.

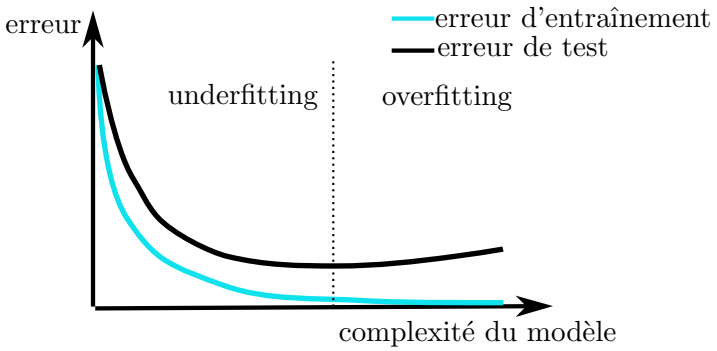

FIGURE 7 – Erreur d'entrainement et de test au fur et à mesure de l'entraînement du modèle.

3. *Early stopping* : consiste à entraîner le réseau en utilisant à la fois une base d'entraînement et une base de test, et à stopper l'entraînement lorsque $\mathcal{L}_{test}(W)$ se met à ré-augmenter
4. *Dropout* : consiste à désactiver certains neurones à chaque étape de la descente de gradient, pour éviter le sur-apprentissage
5. Augmentation de données : consiste à augmenter la taille de la base d'apprentissage en lui ajoutant des données obtenues par transformations (ajout de bruit, transformations géométriques, etc) des données de la base de départ.

4.2 Régularisation quadratique des poids

Voyons ce qui se passe lorsqu'on ajoute à la fonction de perte $\mathcal{L}(W)$ un terme pénalisant la norme l^2 des poids. La fonction devient

$$\mathcal{L}_R(W) = \mathcal{L}(W) + \frac{\lambda}{2}\|W\|_2^2.$$

Remarque : on a

$$\nabla\mathcal{L}_R(W) = \nabla\mathcal{L}(W) + \lambda W,$$

donc la descente de gradient devient

$$W_{t+1} = W_t + \alpha\nabla\mathcal{L}_R(W_t) = (1-\alpha\lambda)W_t - \alpha\mathcal{L}(W_t).$$

Si W^* est un minimum local de $\mathcal{L}$, on peut écrire au voisinage de W^*,

$$\mathcal{L}(W) \simeq \mathcal{L}(W^*) + \frac{1}{2}(W-W^*)^t H_{\mathcal{L}}(W^*)(W-W^*)$$

avec $H_{\mathcal{L}}$ la matrice hessienne de $\mathcal{L}$. Ainsi, au voisinage de W^*, le gradient $\nabla\mathcal{L}$ est bien approché par $H(W-W^*)$ et

$$\nabla\mathcal{L}_R(W) \simeq H(W-W^*) + \lambda W,$$

donc le minimum local W_R^* correspondant devrait vérifier (en diagonalisant $H = Q\Lambda Q^t$ avec $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$)

$$W_R^* \simeq (H+\lambda Id)^{-1}HW^* = Q(\Lambda+\lambda Id)^{-1}Q^tW^*.$$

En première approximation, le minimum local W_R^* est donc une version de W dans laquelle on a réduit les coefficients dans la base Q par les quantités $\frac{\lambda_i}{\lambda_i+\lambda}$.

4.3 Régularisation l^1 des poids

On peut ajouter à la fonction de perte $\mathcal{L}(W)$ un terme pénalisant la norme l^1 des poids. La fonction devient

$$\mathcal{L}_R(W) = \mathcal{L}(W) + \lambda \|W\|_1.$$

Remarque : on a

$$\nabla \mathcal{L}_R(W) = \nabla \mathcal{L}(W) + \lambda \text{sign}(W),$$

donc la descente de gradient devient

$$W_{t+1} = W_t + \alpha \nabla \mathcal{L}_R(W_t) = W_t - \alpha \mathcal{L}(W_t) - \alpha \text{sign}(W_t).$$

Si W^* est un minimum local de $\mathcal{L}$, on peut toujours écrire au voisinage de W^*,

$$\mathcal{L}_R(W) \simeq \mathcal{L}(W^*) + \frac{1}{2}(W - W^*)^t H_{\mathcal{L}}(W^*)(W - W^*) + \lambda \|W\|_1.$$

Pour donner un peu d'intuition à cette régularisation, on peut supposer (c'est évidemment faux en général) que la hessienne $H_{\mathcal{L}}(W^*)$ est diagonale avec des coefficients H_{ii} sur la diagonale. On peut alors réécrire

$$\mathcal{L}_R(W) \simeq \mathcal{L}(W^*) + \sum_i \left(\frac{1}{2} H_{ii}(W_i - W_i^*)^2 + \lambda |W_i| \right).$$

Ainsi, le minimum local W_R^* correspondant devrait vérifier

$$(W_R^*)_i = \begin{cases} \max(W_i^* - \frac{\lambda}{H_{ii}}, 0) & \text{si } W_R^* \geq 0 \\ \max(W_i^* + \frac{\lambda}{H_{ii}}, 0) & \text{si } W_R^* < 0. \end{cases}$$

La régularisation par une norme l^1 revient donc dans ce cas à appliquer un seuillage doux aux coefficient de W^*, et favorise donc les résultats parcimonieux.

5 Invariances et réseaux convolutionnels (CNN)

Dans de nombreuses applications, on peut souhaiter que les prédictions du réseau soient invariantes sous certaines transformations des données d'entrée (translation, rotation, etc). C'est par exemple le cas en traitement d'images. Si la base d'apprentissage est suffisamment importante, le réseau peut apprendre cette invariance au moins approximativement. Mais cette approche n'est pas toujours la meilleure, surtout si la base d'apprentissage est de taille limitée, ou s'il y a beaucoup d'invariances à apprendre. Plusieurs alternatives existent :

- enrichir la base d'apprentissage artificiellement avec de nouveaux exemples créés en appliquant des transformations aux exemples de la base de départ ;
- ajouter dans la fonction de perte un terme qui pénalise un changement dans le résultat du réseau si la donnée d'entrée est transformée
- extraire des données d'entrée des caractéristiques invariantes et utiliser un réseau qui prend ces caractéristiques en entrée ;
- construire un réseau qui par sa structure respecte intrinsèquement ces invariances. C'est le cas par exemple des réseaux convolutionnels utilisés en vision par ordinateur.

Références

[1] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4) :303–314, 1989.

[2] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2) :251–257, 1991.