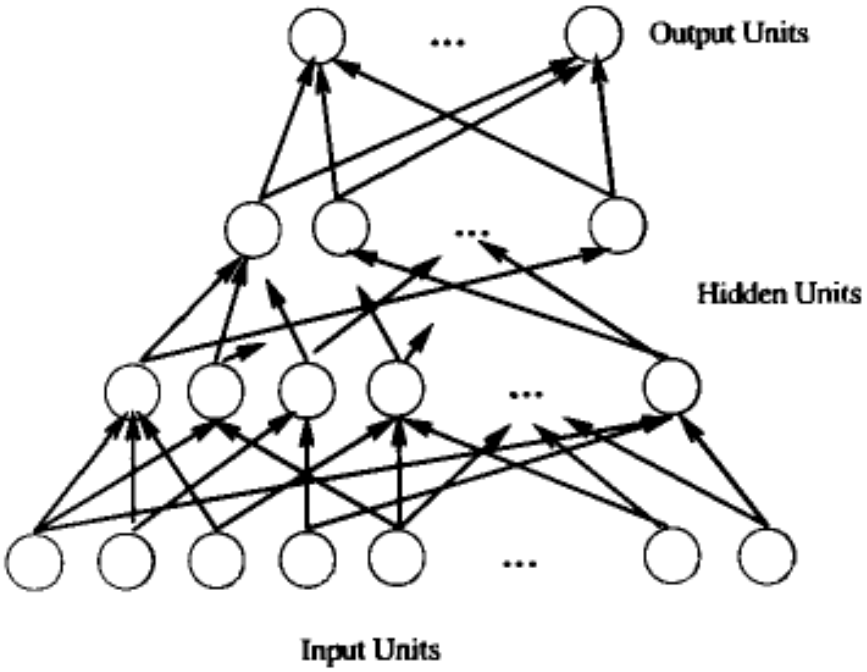
Cours IFT6266,
Réseaux de neurones multi-couches

1 Introduction



Par rapport aux réseaux de neurones à une couche, comme le Perceptron ou la régression logistique, l'idée de base est simplement d'empiler plusieurs réseaux de neurones à une couche [7,8,9].

Ce qui est intéressant avec des réseaux multi-couches c'est que l'on peut montrer que ce sont des approximateurs universels. Cela veut dire que l'on peut théoriquement répresenter n'importe quelle fonction continue avec une erreur d'approximation positive donnée, si on nous permet de choisir le nombre de neurones et leurs paramètres. On appelle neurones cachés (ou unités cachées) ceux qui ne sont ni des entrées ni des sorties de la fonction. Plus on a de neurones cachés et mieux on peut approximer (en supposant qu'on puisse choisir les paramètres). Le désavantage des réseaux multi-couches c'est qu'on perd la convexité que l'on a généralement avec les réseaux à une couche, et l'optimisation devient plus difficile, à cause de la présence de minima locaux dans l'espace des paramètres. Les méthodes d'optimisation et de régularisation des réseaux de neurones (incluant le choix du nombre d'unités

cachées) seront traités ailleurs.

On va se placer ici dans le cadre habituel de l'apprentissage machine supervisé, avec des exemples qui sont des paires (x, y), et avec un critère d'apprentissage que l'on veut minimiser et qui contient une somme sur les exemples d'une fonction de coût

$$L(f, z)$$

avec $z = (x, y)$ notre exemple = (x=entrée, y=sortie désirée). Par exemple on a le coût quadratique pour la régression

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

avec y un vecteur réel, ou bien le coût de log-vraisemblance négative en classification probabiliste

$$L(f, (x, y)) = -\log f_y(x)$$

avec y un indice de classe.

2 Réseau de neurones à une couche cachée

Considérons d'abord le cas le plus simple et le plus fréquemment utilisé, c'est à dire le réseau de neurones à une couche cachée.

On construit une classe de fonctions de la forme suivante:

$$\mathcal{F} = \{f : \mathbb{R}^d \mapsto \mathbb{R}^m, t.q. f(x) = G(b + W \tanh(c + Vx))\}$$

où $y = \tanh(x) \Leftrightarrow y_i = \tanh(x_i)$ s'applique élément par élément, $\tanh(x) = 2\text{sigmoid}(x) - 1$, $\text{sigmoid}(x) = 1/(1 + e^{-x})$, $b \in \mathbb{R}^m$, $W \in \mathbb{R}^{m \times h}$, $c \in \mathbb{R}^h$, $V \in \mathbb{R}^{h \times d}$ sont les paramètres $\theta = (b, W, c, V)$. La fonction G est fixée d'avance selon l'application. Pour la régression on prend habituellement G l'identité. Pour la prédiction de probabilités conditionnelles, on prend généralement G la *softmax*:

$$\text{softmax}_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

De manière générale, si on voulait prédire les paramètres ω de la loi de Y (conditionnellement à X), la fonction G transforme des nombres sans contraintes (les $b + W \tanh(c + Vx)$) en ω. Par exemple si ω_i est une variance on voudrait que

G_i soit toujours positif, que l'on pourrait obtenir en prenant $G_i(x) = x_i^2$ ou bien (pour éviter des symmétries)

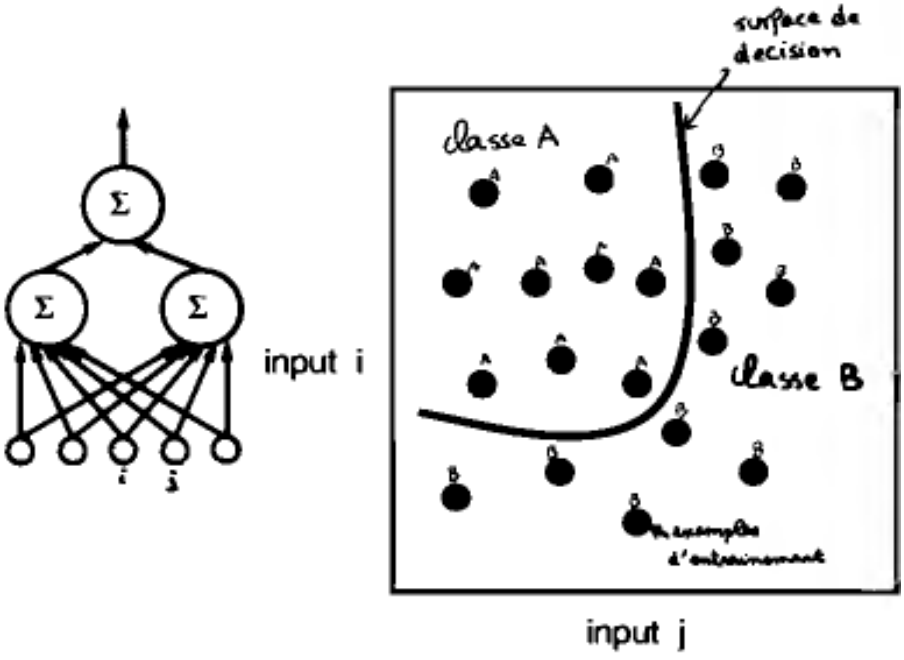
$$G_i(x) = \text{softplus}(x_i)$$

où la fonction softplus(.) est monotone croissante est amène $\mathbb{R}$ à $\mathbb{R}_+$:

$$\text{softplus}(x) = \log(1 + e^x)$$

qui est asymptotiquement x pour $x \to \infty$ et 0 pour $x \to -\infty$. Ce n'est pas toujours le cas que G a autant d'entrées que de sorties. Par exemple pour la classification probabiliste, on aurait besoin de seulement $m - 1$ entrées pour obtenir les m probabilités (à cause de la contrainte de somme à 1).

On appelle h le **nombre d'unités cachées**. C'est un hyper-paramètre important car il contrôle la richesse de la classe de fonctions $\mathcal{F}$ (aussi appelée la *capacité*), donc plus h est grand plus grand sera la capacité (le nombre d'exemples que l'on peut théoriquement apprendre par coeur, quelle que soit leur étiquette).



Si on écrit notre réseau de neurones à une couche en scalaires on obtient (dans le cas où G est l'identité)

$$f_i(x) = b_i + \sum_{j=1}^{h} W_{ij} \tanh(c_j + \sum_k V_{jk} x_k).$$

On voit que les quantités $\tanh(c_j + \sum_k V_{jk} x_k)$ ne dépendent pas de i et qu'il serait donc stupide de les recalculer pour chaque i. On peut donc les calculer une fois pour toutes, en créant des variables intermédiaire que l'on va appeler **les sorties des unités cachées**:

$$u_j = \tanh(c_j + \sum_k V_{jk} x_k).$$

3 Calcul des gradients dans un réseau à une couche cachée

Pour calculer les gradients dans un réseau à une couche cachée, de la même manière qu'on veut factoriser les calculs pour obtenir $f(x)$, on veut factoriser les calculs pour obtenir $\frac{\partial L(f,(x,y))}{\partial \theta}$. De même qu'on avait besoin pour le calcul efficace de $f(x)$ d'utiliser des calculs intermédiaires u_j ci-dessus, on va utiliser le calcul de quantités intermédiaires $\frac{\partial L(f,(x,y))}{\partial u_j}$ pour rendre le calcul du gradient efficace. Comme la fonction G peut introduire des dépendences entre les intrants i de G et les sorties $j \neq i$ de G, il est aussi utile de nommer ces intrants (appelons les s_i) et de les considérer comme quantité intermédiaire. Le calcul de $f(x)$ pourrait donc se faire selon les étapes suivantes:

1. Pour $j = 1$ à h, calculer $u_j = \tanh(c_j + \sum_k V_{jk} x_k)$.
2. Pour $i = 1$ à m, calculer $s_i = b_i + \sum_j W_{ij} u_j$.
3. Calculer $f(x) = G(s)$.
4. Quand ou si y est observé, calculer le coût $Q = L(f, (x, y))$.

Cela donnerait l'algorithme suivant pour le calcul des gradients (si y est observé). Pour simplifier notons $Q = L(f, (x, y))$. Notez que l'on fait les calculs dans l'ordre inverse, d'où le nom de **rétro-propagation**:

1. Calculer $\frac{\partial Q}{\partial s}$.
2. Pour $i = 1$ à m, assigner $\frac{\partial Q}{\partial b_i} = \frac{\partial Q}{\partial s_i}$.
3. Pour $j = 1$ à h
   - Pour $i = 1$ à m, calculer $\frac{\partial Q}{\partial W_{ij}} = \frac{\partial Q}{\partial s_i} u_j$

- Calculer $\frac{\partial Q}{\partial u_j} = \sum_{i=1}^{m} \frac{\partial Q}{\partial s_i} W_{ij}$
- Calculer $\frac{\partial Q}{\partial c_j} = \frac{\partial Q}{\partial u_j}(1 - u_j^2)$

4. Pour $j = 1$ à h
   - Pour $k = 1$ à d, calculer $\frac{\partial Q}{\partial V_{jk}} = \frac{\partial Q}{\partial c_j} x_k$

4 Réseau à plusieurs couches cachées

En exploitant le principe général des **graphes de flot** (voir notes `gradient.pdf`) on remarque que l'algorithme ci-haut est une application particulière du principe de calcul du gradient dans les graphes de flot. Cela nous amène de manière naturelle à pouvoir généraliser à des réseaux de neurones avec une connectivité arbitraire (mais toujours formant un graphe dirigé acyclique), et en particulier au cas des réseaux à plusieurs couches cachées.

On peut considérer chaque couche comme un module M_i qui calcule $a_i = f_i(a_{i-1}, \theta_i)$, où a_i est le vecteur de sortie du $i$ème module, avec des paramètres $\theta_i = (w_i, b_i)$ (w_i est une matrice de poids $w_{i,j,k}$ et b_i un vecteur de biais $b_{i,j}$).

Le gradient $\frac{\partial Q}{\partial a_L}$ de la fonction de coût pour un exemple par rapport aux sorties $f(x) = a_L$ de la dernière couche peuvent être calculées facilement. Par exemple, si $Q(f, z) = 0.5(f(x) - T(y))^2$, alors

$$\frac{\partial Q}{\partial a_L} = (a_L - T(y)).$$

Étant donné le gradient sur le vecteur de sortie d'un module quelconque, $\frac{\partial Q}{\partial a_i}$, ce module peut calculer le gradient sur ses entrées, $\frac{\partial Q}{\partial a_{i-1}}$, et sur ses paramètres, $\frac{\partial Q}{\partial \theta_i}$. Par exemple si $a_{i,j} = \tanh(b_{i,j} + \sum_k w_{i,j,k} a_{i-1,k})$

$$\begin{aligned}
\frac{\partial Q}{\partial a_{i-1,k}} &= \sum_j \frac{\partial Q}{\partial a_{i,j}} (1 - a_{i,j}^2) \, w_{i,j,k} \\
\frac{\partial Q}{\partial w_{i,j,k}} &= \frac{\partial Q}{\partial a_{i,j}} (1 - a_{i,j}^2) \, a_{i-1,k} \\
\frac{\partial Q}{\partial b_{i,j}} &= \frac{\partial Q}{\partial a_{i,j}} (1 - a_{i,j}^2)
\end{aligned} \tag{1}$$

On peut ainsi propager l'information sur l'erreur (comment changer les a_i de manière à réduire Q) de la couche M_i à la couche M_{i-1} récursivement (d'où le terme **rétropropagation de l'erreur**). D'une manière plus générale, nous pouvons entraîner sur le même principe n'importe quel séquence de modules (ou plus généralement, réseau de modules). Chaque module a un vecteur d'entrée, un vecteur de sortie et un vecteur de paramètres, et peut faire deux opérations:

1. calculer sa sortie en fonction de son entrée et de ses paramètres,
2. calculer le gradient par rapport à ses entrées et paramètres, étant donné le gradient par rapport aux sorties (ainsi que l'entrée et les paramètres).

5 Références

1. Gluck, M. and Rumelhart, D. (1990). *Neuroscience and connectionist theory*. Lawrence Erlbaum, London.
2. McCulloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5.
3. P. M. Lewis II and C. L. Coates, *A realization procedure for threshold gate networks*, Proceedings of the Third Annual Symposium on Switching Circuit Theory and Logical Design (Chicago, Illinois), American Institute of Electrical Engineers, 7–12 October 1962, pp. 159–168.
4. Rumelhart, D., McClelland, J., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge.
5. F. Rosenblatt, *The perceptron — a perceiving and recognizing automaton*, Tech. Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, N.Y., 1957.
6. F. Rosenblatt, *Principles of neurodynamics*, Spartan, New York, 1962.
7. Y. LeCun, *Une procédure d'apprentissage pour réseau à seuil assymétrique*, Cognitiva 85: A la Frontière de l'Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences (Paris 1985), CESTA, Paris, 1985, pp. 599–604.

8. D.B. Parker, *Learning logic*, Tech. Report TR–47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

9. D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature **323** (1986), 533–536.

10. B. Widrow, *Generalization and information storage in networks of adaline "neurons"*, Self-Organizing Systems 1962 (Chicago 1962) (M.C. Yovits, G.T. Jacobi, and G.D. Goldstein, eds.), Spartan, Washington, 1962, pp. 435–461.