Le Deep Learning pas à pas

Manuel Alves et Pirmin Lemberger

PARTIE I - Concepts

Des labos de R&D à la vie quotidienne

L'image ci-dessous vous rappelle bien quelque chose ? On dirait…, mais oui, c'est la *Nuit étoilée* de Van Gogh ? Une *Nuit étoilée* où le Golden Gate Bridge remplace cependant le village bucolique de Saint Remy-de-Provence. Un simple pastiche «à la manière de » qui n'a a priori rien d'extraordinaire, si ce n'est que cette image a été construite numériquement à partir d'une simple photo du célèbre pont de San Francisco et d'une reproduction du chef d'œuvre impressionniste. Ce tour de passe-passe a été réalisé par Artomatix, une start-up irlandaise fondée en 2014 et spécialisée dans la conception de graphismes réalistes pour le cinéma et les jeux vidéo.

En 2012, Google a surpris la communauté du Machine Learning en démontrant que son Google Brain était capable de découvrir, par lui-même, des concepts de haut niveau tel que des visages, des corps humains ou des images de chats, ceci en épluchant des millions d'images glanées sur YouTube. Ce résultat est remarquable car jusque-là les techniques de reconnaissance d'images se basaient sur des approches dite supervisées, chaque image devant être explicitement désignée comme contenant un visage humain, une tête de chat etc. Le tour de force des équipes de Google a été de court-circuiter cette étape de labélisation manuelle (tagging). L'enjeu pour Google est énorme car il s'agit ni plus ni moins que de faire passer à l'échelle ses algorithmes d'apprentissage en tirant parti de la manne d'images disponibles sur le web qui, dans leur immense majorité, ne sont évidemment pas tagguées.

Le point commun entre ces deux applications du Machine Learning ? Les deux utilisent une classe d'algorithmes d'apprentissage automatique que l'on appelle le **Deep Learning**.

Sorti des labos de R&D depuis quelques années, le Deep Learning investit progressivement notre quotidien : la reconnaissance vocale de l'assistant Siri d'Apple, le tagging automatique de morceaux de musique, la synthèse vocale avancée, le légendage automatique d'images et même la conception de nouvelles molécules pharmaceutiques, toutes ces applications mettent aujourd'hui en œuvre des techniques de Deep Learning.

Cet article comprend deux parties. Dans la première partie nous présenterons en guise d'illustration l'une de ces architectures profondes : les Deep Belief Networks. Notre objectif est d'en donner une présentation conceptuelle détaillée et d'expliquer comment certaines avancées récentes sont parvenues à surmonter d'anciennes difficultés inhérentes aux systèmes de neurones. Une deuxième partie abordera les problèmes liés à l'implémentation des architectures profondes en général.

Cet article présuppose une connaissance rudimentaire des concepts du Machine Learning.

Flashback

L'idée de construire des **réseaux de neurones** (RN) artificiels n'est pas neuve, elle remonte à la fin des années 50. Très schématiquement l'idée consiste à s'inspirer du fonctionnement du cortex visuel des animaux. Dans une version élémentaire, chaque neurone i d'un tel réseau possède un niveau d'activation x_i compris entre 0 et 1. Le schéma d'interconnexion entre neurones définit l'architecture du réseau. Une **architecture** classique consiste à organiser les neurones en couches successives avec des interconnexions limitées aux couches adjacentes comme le montre la figure 1 (a).

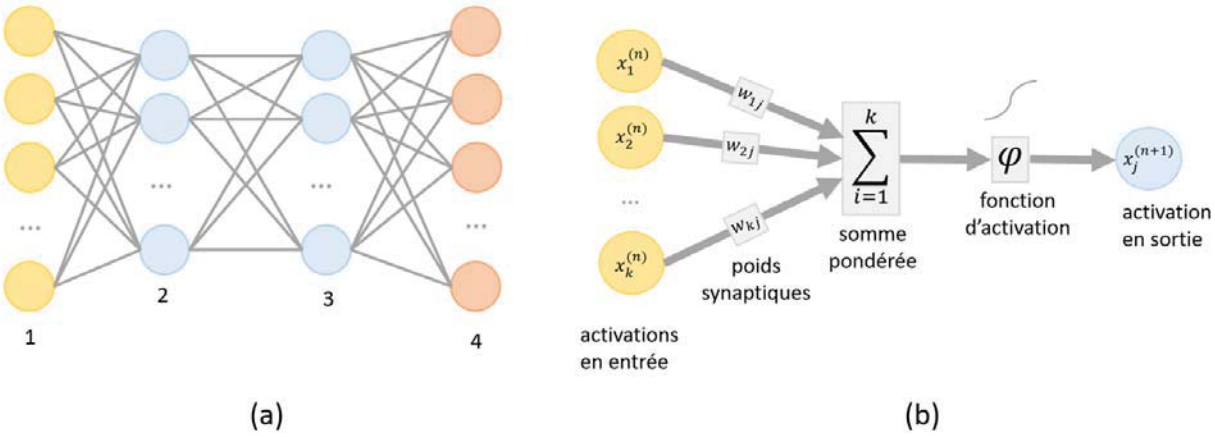

Figure 1 : (a) un RN organisé en 4 couches, (b) le mécanisme d'activation d'un neurone

Dans le cadre d'un **apprentissage supervisé**, un exemple classique d'utilisation d'un RN est celui d'un système chargé de classer des images de chiffres manuscrits. Dans cet exemple les niveaux d'activations $x_i^{(1)}$ des neurones $i = 1, \dots, k$ de la **couche d'entrée** correspondent aux niveaux de gris des pixels de l'image, k étant le nombre de pixels des images. La **couche de sortie** est en l'occurrence constituée de neurones $y_j, j = 0,1, \dots ,9$ qui correspondent aux dix chiffres que l'on peut attribuer à chaque image d'un ensemble d'entraînement. Les niveaux d'activation des neurones sont déterminés récursivement, couche par couche. Ceux de la couche $n + 1$ sont calculés à l'aide d'une **fonction d'activation** φ en fonction des niveaux d'activation des neurones de la couche n pondérés par certains **poids synaptiques** w_{ij} :

$$x_j^{(n+1)} = \varphi\left(\sum_{i=1}^{k} w_{ij}\, x_i^{(n)}\right) \qquad (1)$$

La somme porte sur tous les neurones i de la couche n connectés au neurone j de la couche $n + 1$, voir la figure 1 (b). La fonction d'activation φ est typiquement une fonction telle que la sigmoïde $x \to \varphi(x) \equiv \mathrm{sigm}\,(x)$ représentée dans la figure 2. Elle est croissante, différentiable, non-linéaire et prend ses valeurs entre 0 et 1.

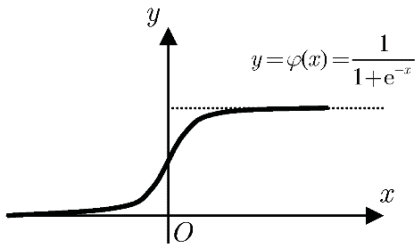

Figure 2 : la fonction d'activation sigmoïde

L'**entraînement** du réseau consiste à trouver des poids synaptiques w_{ij} tels que la **couche de sortie** permette de classer avec précision les images d'un ensemble d'entraînement. On espère naturellement que le RN présentera des capacités de **généralisation** sur des exemples qu'il n'a jamais rencontré.

Une question se pose d'emblée : « De tels poids w_{ij} existent-t-ils toujours quel que soit l'objectif assigné au RN ? ». Par chance, un résultat mathématique connu sous le nom de **théorème d'approximation universel**, garantit que la chose est effectivement possible, même pour un réseau ne comportant qu'une seule couche cachée, à condition toutefois que φ soit non-linéaire et qu'un nombre suffisant de neurones soient mis en jeu en fonction de la marge d'erreur tolérée. En revanche si l'on se limite à des fonctions d'activation φ linéaires, le réseau fonctionnera comme une régression linéaire ordinaire et ne sera par conséquent d'aucune utilité pour prédire des phénomènes non-linéaires complexes.

Reste à construire un algorithme qui fournit une bonne approximation des poids w_{ij} en un temps acceptable.

Jusqu'à il y a peu l'algorithme phare pour l'entraînement des systèmes de neurones multicouches était l'algorithme dit de **rétro-propagation**. Pour fixer les idées, restons sur l'exemple des digits. Pour une image $\mathbf{x} = (x_1, \dots, x_k)$ en entrée et un ensemble de poids synaptiques $\mathbf{w}$ on commence par définir une **fonction de coût** C qui mesure l'écart entre les prédictions $y_j(\mathbf{x})$ des neurones de sortie et les **valeurs cibles** t_j spécifiées dans l'ensemble d'entraînement :

$$C(\mathbf{x}\,;\mathbf{w}) = \sum_{j=0}^{9} \big(y_j(\mathbf{x}) - t_j\big)^2 \qquad (2)$$

Naturellement $y_j(\mathbf{x})$ est une fonction extrêmement compliquée de la configuration $\mathbf{x}$ en entrée et des poids $\mathbf{w}$, spécifiée par l'itération de (1). Ce que l'on cherche à minimiser en principe c'est la somme ou la moyenne $C_{\text{MSE}}(\mathbf{w})$ de ces erreurs sur toutes les configurations $\mathbf{x}$ de l'ensemble d'entraînement $\mathcal{E}$:

$$C_{\text{MSE}}(\mathbf{w}) = \sum_{\mathbf{x}\in\mathcal{E}} C(\mathbf{x};\mathbf{w}) \qquad (3)$$

Une **descente de gradient** consiste à calculer la direction dans l'espace des poids $\mathbf{w}$ dans laquelle la décroissance de $C_{\text{MSE}}(\mathbf{w})$ est maximale. Cette direction est naturellement donnée par l'opposé du gradient $\nabla C_{\text{MSE}}(\mathbf{w})$. Si tout se passe bien, c.à.d. qu'il n'y a pas de minima locaux, on s'approchera du minimum de $C_{\text{MSE}}(\mathbf{w})$ par itérations successives de petites corrections apportées à $\mathbf{w}$:

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \alpha \nabla C_{\text{MSE}}\big(\mathbf{w}^{(\text{old})}\big) \qquad (4)$$

Le coefficient α permet d'ajuster la vitesse d'apprentissage, le cas échéant dynamiquement. Dans la pratique cependant, le calcul de la somme sur toutes les configurations $\mathbf{x} \in \mathcal{E}$ est impossible car beaucoup trop coûteux en temps de calcul. Pour remédier à cette situation on utilise une **descente de gradient stochastique** (SGD) qui revient à approximer à chaque étape le gradient $\nabla C_{\text{MSE}}(\mathbf{w})$ par le gradient $\nabla C(\mathbf{x};\mathbf{w})$ d'un seul échantillon $\mathbf{x}$ tiré au hasard dans l'ensemble d'entraînement[1] $\mathcal{E}$. Cette stratégie qui peut paraître audacieuse a priori fonctionne car, intuitivement, les erreurs occasionnées par cette approximation se compensent sur le long terme. L'algorithme de SGD a fait ses preuves

[1] Il existe des variantes où l'on calcule le gradient de petites sommes que l'on appelle mini-batch.

dans de nombreux problèmes d'optimisation et des résultats théoriques viennent étayer cette intuition dans certains cas particuliers.

Reste à calculer toutes les composantes $\partial C(\mathbf{x};\mathbf{w})/\partial w_{ij}$ du gradient $\nabla C(\mathbf{x};\mathbf{w})$. La calcul est simple dans son principe puisqu'il ne s'agit ni plus ni moins que de calculer la dérivée d'une fonction composée un peu compliquée définie récursivement par (1) et (2). C'est là qu'intervient l'algorithme de **rétro-propagation**, celui-ci permet d'organiser efficacement ce calcul de dérivée. Il s'avère que les dérivées par rapport aux w_{ij} associés à la couche de sortie sont élémentaires. Le reste du calcul procède par induction car on montre en effet que les dérivées par rapport aux w_{ij} de la couche $n-1$ sont calculables dès lors celles de la couche n ont déjà été calculées. Ce calcul à reculons est à l'origine du nom de l'algorithme.

Tout astucieux qu'il soit l'algorithme de rétro-propagation souffre cependant de deux inconvénients majeurs :

1. l'expérience montre que le temps d'entraînement d'un RN croît rapidement lorsque le nombre de couches augmente. C'est d'ailleurs l'une des raisons pour lesquelles à partir des années 1990 les RN ont été remisés au profit d'autres algorithmes non-linéaires moins gourmand en ressources comme les SVM.
2. les RN n'échappent pas au problème central du Machine Learning : le **surapprentissage**.

Ces résultats empiriques sont en partie corroborés par des résultats en théorie de la complexité qui démontrent que l'entraînement d'un RN est un problème complexe dans un sens précis[2].

Pour progresser, de nouvelles idées sont nécessaires. Après plusieurs décennies de stagnation c'est G. E. Hinton[3] et son équipe qui, en 2006, ont fait la principale percée dans ce domaine.

Les idées nouvelles de Hinton

L'article fondateur de Hinton contient une série d'idées et de stratégies innovantes. Ce paragraphe les présente sur un plan intuitif, le suivant rentre dans les détails.

Un modèle génératif

Dans un contexte de **classification supervisée**, où l'on essaie d'apprendre une relation entre des observations $\mathbf{x}$ en entrée (les images de chiffres) et des labels y en sortie (les chiffres '0' à '9'), la stratégie la plus courante consiste à construire un **modèle discriminant**, c.à.d. un modèle pour les probabilités conditionnelle $p(y|\mathbf{x})$ d'observer un label y lorsqu'on connait l'entrée $\mathbf{x}$. La régression logistique rentre par exemple dans ce cadre. Par contraste, la stratégie de Hinton et al. consiste à élaborer un **modèle génératif**, c.à.d. un modèle pour la distribution de probabilité conjointe $p(\mathbf{x},y)$ des images $\mathbf{x}$ et des labels y. Autrement dit, on cherche à construire un RN capable d'apprendre puis de générer simultanément les images $\mathbf{x}$ et les labels y avec une distribution de probabilité proche de celle observée dans un ensemble d'entraînement.

Une fois le RN entraîné on pourra l'utiliser d'une part pour reconnaître des images, c.à.d. associer un chiffre y à une image $\mathbf{x}$ (on cherche le y qui maximise $p(y|\mathbf{x})$) ou, inversement, pour générer des images $\mathbf{x}$ correspondant à un chiffre y (on échantillonne $p(\mathbf{x}|y)$). A ce stade, nous invitons vivement le lecteur à jouer avec l'application mise en ligne par Hinton qui illustre ces deux processus de manière visuelle et dynamique. Comme le veut l'adage : une image vaut mille mots, alors une vidéo…

[2] Il a été démontré que la recherche de poids optimaux dans un RN multicouches est un problème NP-complet.
[3] G.E. Hinton est professeur émérite à l'université de Toronto et chercheur chez Google.

Une initialisation rapide du RN puis un fine-tuning lent

L'architecture de RN utilisée pour réaliser le modèle génératif de Hinton et al. s'appelle un **Deep Belief Network** (DBN). Elle contient deux parties représentées dans la figure 3.

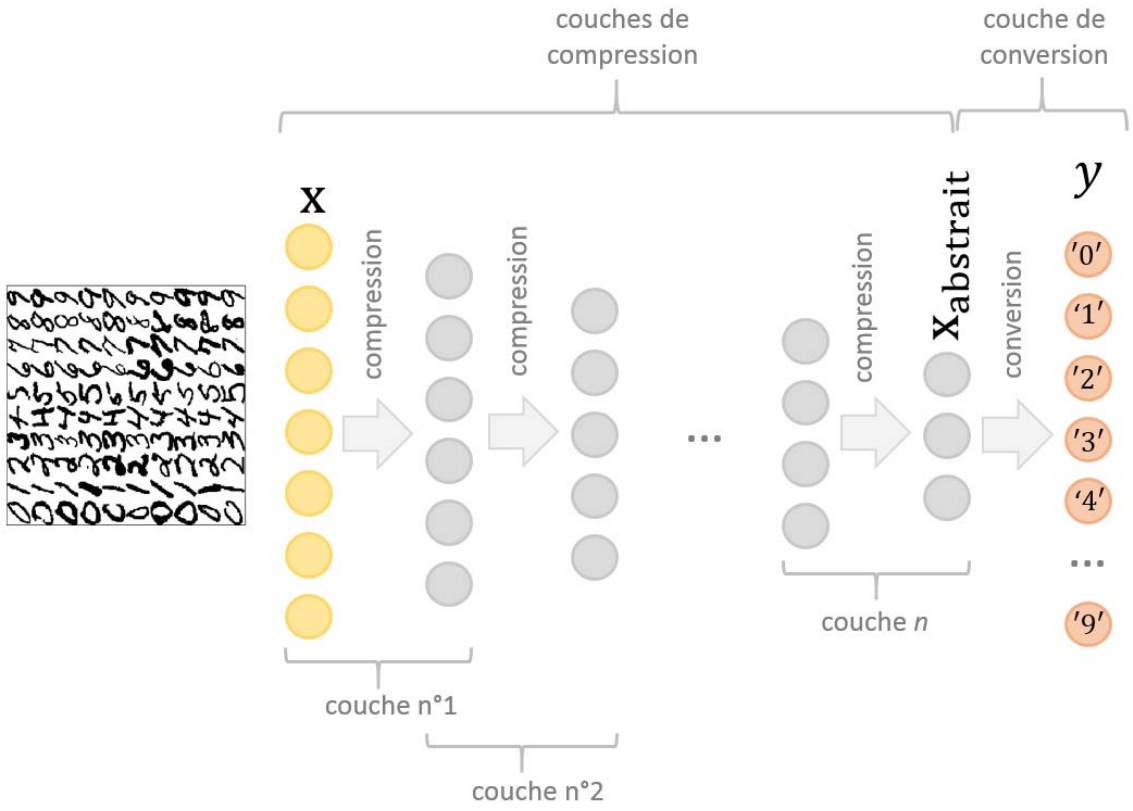

Figure 3 : Un Deep Belief Network qui identifie des images de digits. La première partie du DBN est constituée d'un ensemble de couches de compression qui convertissent les données entrées $\mathbf{x}$ en une représentation abstraite $\mathbf{x}_{abstrait}$. La seconde partie convertit cette représentation en labels de classification y.

La première partie a pour objectif d'apprendre la distribution des données $\mathbf{x}$ présentées en entrée sans tenir compte des labels y. Elle est constituée d'une succession de couches dont chacune contiendra une représentation plus abstraite (ou compressée) que la précédente. Pour ancrer l'intuition considérons un RN chargé de classifier des images. La première couche stockera les niveaux de gris de l'image (l'équivalent de la rétine), la seconde contiendra, par exemple, un encodage des lignes ou des zones de contraste de l'image, la troisième détectera l'existence de certaines formes géométriques simples comme des cercles, la quatrième identifiera certains agencements particuliers de ces figures comme celles qui représentent un '8' formé de deux cercles juxtaposés et ainsi de suite. Ainsi on automatise en quelque sorte le processus de **feature engineering** manuel du machine learning. Hinton et al. ont par ailleurs découvert un algorithme rapide pour entraîner cette première section en procédant couche par couche (nous y reviendrons). Une fois entraînée, cette première section du RN contiendra une **représentation hiérarchique** des données en entrée, la dernière couche encodant la représentation $\mathbf{x}_{\text{abstrait}}$ la plus abstraite et aussi, c'est l'idée, la plus utile. Cette première phase peut être conçue comme une **initialisation** efficace du DBN, on l'appelle aussi **pré-entraînement**.

Le rôle de la seconde partie du DBN est de convertir la représentation abstraite et obscure $\mathbf{x}_{\text{abstrait}}$ en labels y utilisables par exemple dans le cadre d'un apprentissage supervisé. Dans l'exemple des digits cette représentation sera constituée d'une couche de sortie de dix neurones, un neurone par

digit. Cette conversion peut être réalisée au moyen d'une **couche logistique** entraînée par une SGD classique.

L'entraînement du DBN est considéré comme achevé lorsque la performance du DBN évaluée sur un ensemble de validation distinct de l'ensemble d'entraînement ne progresse plus significativement. Cette seconde étape est appelée le **fine-tuning**, elle est généralement beaucoup plus lente que l'initialisation.

Les RBM comme briques de base

Pour mener à bien le programme esquissé dans les deux paragraphes précédents il faut disposer d'une brique de compression (c.à.d. un morceau de RN) qui implémente les couches de la première section. La principale caractéristique attendue de cette brique est sa capacité à apprendre rapidement une distribution de probabilité spécifiée empiriquement par un jeu d'exemples. Il existe pour cela différentes alternatives[4] qui permettent peu ou prou d'implémenter la même philosophie mais nous nous restreindrons dans cet article à la description des **Restricted Boltzman Machines** (RBM) utilisés par Hinton et al. dans leur travail pionnier de 2006.

Le paragraphe suivant explique ce que sont les **RBM** et comment l'algorithme dit de **Contrastive Divergence** de Hinton et al. exploite astucieusement leurs propriétés.

Zoom sur les RBM

Deux propriétés qui simplifient la tâche

Les RBM sont des RN **stochastiques**, qui peuvent être entraînés sur un mode non supervisé pour générer des échantillons selon une distribution de probabilité complexe spécifiée par des exemples (l'ensemble d'images de digits p.ex.).

Alors que les neurones des RN évoqués dans la section Flashback ont des niveaux d'activation x_i déterministes compris entre 0 et 1, les neurones des RBM sont des variables aléatoires binaires. Elles sont réparties sur deux couches. L'une de ces couches est appelée la **couche visible**, c'est elle qui encodera les exemples de l'ensemble d'entraînement $\mathcal{E}$. Notons collectivement $\mathbf{v} = (v_1, \ldots, v_k)$ les niveaux d'activation de ces k neurones. L'autre est appelée **couche cachée**, c'est elle qui stockera une forme compressée ou abstraite des données de la couche visible. Notons $\mathbf{h} = (h_1, \ldots, h_l)$ les niveaux d'activation correspondants.

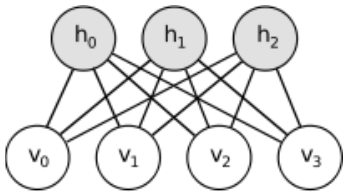

Figure 4 : Une RBM est constituée d'une couche de neurones visibles et d'une couche de neurones cachés

Par définition, une RBM est définie par la distribution de **probabilité conjointe** suivante[5] sur $\mathbf{v}$ et $\mathbf{h}$:

$$p(\mathbf{v}, \mathbf{h}) = \frac{\mathrm{e}^{-E(\mathbf{v}, \mathbf{h})}}{Z} \quad \text{où} \quad E(\mathbf{v}, \mathbf{h}) = \sum_{i=1}^{k} a_i \, v_i + \sum_{j=1}^{l} b_j \, h_j + \sum_{i=1}^{k} \sum_{j=1}^{l} w_{ij} \, v_i \, h_j \qquad (5)$$

[4] Les principaux sont les auto-encodeurs et les réseaux de convolution, voir cet article p.ex.

[5] La constante Z garantit que $p(\mathbf{v}, \mathbf{h})$ est bien une distribution de probabilité, on l'appelle la fonction de partition.

Les grandes valeurs de $E(\mathbf{v},\mathbf{h})$ correspondent à de probabilités $p(\mathbf{v},\mathbf{h})$ faibles. La physique statistique utilise des formules similaires pour attribuer des probabilités $p(\mathbf{x})$ à certaines configurations d'un système thermique. Dans un tel cadre $E(\mathbf{x})$ s'interprète essentiellement comme l'énergie de $\mathbf{x}$. L'intuition derrière la formule (5) est donc que les configurations $(\mathbf{v},\mathbf{h})$ les plus probables sont celles dont l'**énergie** $E(\mathbf{v},\mathbf{h})$ est minimale. Remarquons que les poids w_{ij} ne connectent que les h_i et les v_j mais pas les h_i entre eux ni les v_j entre eux, c'est là l'origine de l'adjectif « restricted », voir la figure 4. Les paramètres $\mathbf{a}$, $\mathbf{b}$ et $\mathbf{w}$ sont à optimiser durant l'entraînement. Pour simplifier l'écriture nous les désignerons globalement par θ.

Ce qui nous intéresse au premier chef c'est la **distribution marginale** $p_\theta(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v},\mathbf{h})$ des variables visibles $\mathbf{v}$ de la RBM (la somme porte sur toutes les configurations $\mathbf{h}$ possibles des neurones cachés et l'indice θ rappelle la dépendance de p). L'optimisation de θ a pour objectif de maximiser la ressemblance entre la distribution marginale $p_\theta(\mathbf{v})$ générée par la RBM et la distribution expérimentale observée au sein de l'ensemble d'entraînement $\mathcal{E}$. Pour cela on va maximiser la **vraisemblance** $\prod_{\mathbf{v}\in\mathcal{E}} p_\theta(\mathbf{v})$ des valeurs observées $\mathbf{v}$. Pour reformuler le problème comme la recherche du minimum d'une fonction de coût qui est une somme de contributions associées à chaque $\mathbf{v} \in \mathcal{E}$ on passe au logarithme et on change le signe :

$$C_{\mathrm{NLL}}(\theta) = -\sum_{\mathbf{v}\in\mathcal{E}} \log p_\theta(\mathbf{v}) \qquad (6)$$

L'acronyme NLL signifie *Negative Log Likelyhood*. On peut alors appliquer à C_{NLL} la même stratégie d'optimisation par SGD qu'avec C_{MSE}. Il suffit pour cela de savoir calculer la dérivée de chaque terme dans (6). Pour simplifier les expressions qui suivent on introduit généralement, en parallèle avec l'énergie $E(\mathbf{v},\mathbf{h})$, la notion d'**énergie effective** $F(\mathbf{v})$ définie par $p_\theta(\mathbf{v}) \equiv \mathrm{e}^{-F(\mathbf{v})}/Z$. En utilisant les définitions on trouve :

$$-\frac{\partial}{\partial\theta}\log p_\theta(\mathbf{v}) = \frac{\partial F(\mathbf{v})}{\partial\theta} - \sum_{\mathbf{u}\in\mathcal{E}} p_\theta(\mathbf{u})\frac{\partial F(\mathbf{u})}{\partial\theta} \qquad (7)$$

Deux propriétés des RBM liées à la forme spécifique de l'énergie $E(\mathbf{v},\mathbf{h})$ définie en (5) vont grandement nous simplifier la vie. Ce sont elles qui motivent en définitive le choix des RBM comme briques élémentaires des DBN.

---

1. L'**énergie effective** $F(\mathbf{v})$ qui intervient dans (7) **est calculable facilement** au moyen d'une formule explicite :

$$F(\mathbf{v}) = \sum_{i=1}^{k} a_i v_i - \sum_{j=1}^{l} \log\left[1 + \exp\left(-b_j - \sum\nolimits_{i=1}^{k} w_{ij} v_i\right)\right] \qquad (8)$$

   Sans le secours de la formule (8) il faudrait calculer une énorme somme sur toutes les configurations $\mathbf{h}$, une somme dont le nombre de termes croit exponentiellement avec le nombre l de neurones cachés. Rappelons en effet que $\mathrm{e}^{-F(\mathbf{v})} = \sum_{\mathbf{h}} \mathrm{e}^{-E(\mathbf{v},\mathbf{h})}$.

2. Les **neurones d'une couche sont conditionnellement indépendants lorsque les activités des neurones de l'autre couche sont fixés**, explicitement (par économie on supprime l'indice θ de p_θ) :

$$p(\mathbf{v}|\mathbf{h}) = \prod_{i=1}^{k} p(v_i|\mathbf{h}) \qquad (9)$$

$$p(\mathbf{h}|\mathbf{v}) = \prod_{j=1}^{l} p(h_j|\mathbf{v}) \qquad (9')$$

Un calcul simple montre par ailleurs que les probabilités conditionnelles individuelles sont données par la fonction sigmoïde :

$$p(v_i = 1|\mathbf{h}) = \text{sigm}\left(-a_i - \sum_{j=1}^{l} w_{ji} h_j\right) \qquad (10)$$

$$p(h_j = 1|\mathbf{v}) = \text{sigm}\left(-b_j - \sum_{i=1}^{k} w_{ji} v_i\right) \qquad (10')$$

On retrouve donc la fonction d'activation usuelle mais avec une interprétation stochastique. Comme on le verra les propriétés (9,9') et (10,10') sont cruciales pour échantillonner efficacement la distribution p_θ qui intervient dans la somme dans (7).

Le calcul de la somme dans (7) s'avère cependant impraticable pour des grands ensembles d'entraînement[6] car cela prendrait trop de temps. Il faudra par conséquent nous contenter de l'approximer en n'utilisant qu'un nombre limité de termes échantillonnés selon une distribution proche de $p_\theta(\mathbf{v})$. C'est là que les propriétés d'indépendance conditionnelle précédentes viennent à la rescousse.

L'algorithme de Contrastive Divergence

Pour échantillonner une distribution de probabilité multivariée $p(\mathbf{x}) = p(x_1, \ldots, x_r)$ les méthodes dites **Monte-Carlo Markov-Chain** (MCMC) sont souvent indiquées. Elles permettent de générer itérativement une succession infinie d'échantillons $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}$ … qui, à la longue, seront distribués selon une distribution $p(\mathbf{x})$ souhaitée. Pour les RBM on utilise une version particulière de MCMC dite avec **échantillonnage de Gibbs**. Concrètement cela signifie que l'échantillon $\mathbf{x}^{(n+1)}$ de la MCMC est généré à partir de l'échantillon $\mathbf{x}^{(n)}$ en r étapes (où r est le nombre de variables que contient $\mathbf{x}$). A chaque étape on génère une nouvelle composante x_i en utilisant la probabilité conditionnelle $p(x_i|\mathbf{x}_{-i})$ où $\mathbf{x}_{-i}$ désigne $\mathbf{x}$ dont on a retiré la variable x_i. Dans le contexte des RBM qui nous intéresse $\mathbf{x} = (\mathbf{h},\mathbf{v})$ et les probabilités conditionnelles sont données par les expressions (10) et (10'). Tout l'intérêt de cette propriété d'indépendance conditionnelle propre aux RBM est qu'elle permet de **paralléliser** l'échantillonnage de Gibbs. On va ainsi échantillonner simultanément tous les h_i connaissant les v_j puis, dans un second temps, simultanément tous les v_j connaissant les h_i, explicitement :

$$h_j^{(n+1)} \sim p(h_j|\mathbf{v}^{(n)}) \qquad (11)$$

$$v_i^{(n+1)} \sim p(v_i|\mathbf{h}^{(n+1)}) \qquad (11')$$

Bonne nouvelle, au lieu d'avoir $r = k + l$ étapes il n'y en a donc finalement que deux étapes parallélisées !

Hélas, en dépit de cette bonne nouvelle du parallélisme, il reste un problème de taille. Notre objectif, rappelons-le, est d'optimiser par itération successives les paramètres θ de la RBM au moyen d'une

[6] Pour le problème de reconnaissance des digits $|\mathcal{E}|$=50'000 à titre d'exemple.

SGD. Or chaque itération exige le calcul du gradient (7) qui à son tour demande, du moins en principe, d'attendre que la convergence de l'échantillonnage MCMC vers p_θ s'installe, ce qui est évidemment irréaliste.

Hinton et al. utilisent **deux astuces** pour surmonter cette difficulté. Elles constituent l'algorithme de **Contrastive Divergence**[7] (CD). Comme la distribution p_θ est sensée être proche de la distribution expérimentale dans $\mathcal{E}$ on peut accélérer la convergence de la MCMC en l'initialisant avec un échantillon tiré de $\mathcal{E}$. Ensuite, plutôt que d'attendre patiemment la convergence de la MCMC, on arrête les itérations après un petit nombre d'étapes. Et même, en étant un peu audacieux, après une seule étape ! Ce faisant on commet évidement une erreur puisqu'on ne génère pas exactement p_θ. L'expérience montre cependant que les itérations de la SDG finissent par lisser cette erreur et que cette approximation grossière reste suffisante pour faire converger la fonction de coût $C_{\mathrm{NLL}}(\theta)$ vers un minimum approximatif utile.

Empilement et fine-tuning

Nous voici donc équipé pour entraîner efficacement une RBM. Un DBN consiste comme nous l'avons expliqué en un empilement de RBM et une couche logistique supplémentaire qui converti le contenu de la couche cachée du dernier RBM en labels de classification. L'entraînement d'un DBN complet procède alors selon les étapes suivantes :

1. la RBM n°1 est entraînée par CD avec les échantillons $\mathbf{v}$ de l'ensemble d'entraînement $\mathcal{E}$ présentés sur sa couche visible ;
2. une fois entraînée, cette première RBM fera office de convertisseur/compresseur. A chaque échantillon $\mathbf{v}$ de l'ensemble d'entraînement $\mathcal{E}$ elle associe une configuration $\mathbf{h}$ de neurones de la couche cachée. On construit $\mathbf{h}$ en échantillonnant les distributions $p_\theta(h_i|\mathbf{v})$ avec les paramètres θ appris à l'étape précédente. Les configurations $\mathbf{h}$ ainsi obtenues par transformation sont alors utilisées comme échantillons d'entraînement présentés à la couche visible de la RBM n°2 ;
3. on itère 1. et 2. sur toutes les RBM du DBN. Une fois toutes les RBM entraînées, la **phase d'initialisation** est terminée. On dispose alors d'un mécanisme complexe de conversion des $\mathbf{v}$ en une représentation abstraite/compressée sur la couche cachée de la dernière RBM. C'est une forme de **feature engineering automatisé** ;
4. on peut alors utiliser cette représentation abstraite en entrée d'une couche logistique que l'on entraînera de manière classique avec une SGD supervisée basée sur une fonction de coût C_{NLL} de type maximum de vraisemblance. C'est l'étape dite de **fine-tuning**.

Ce paragraphe clos la partie conceptuelle de cet article. Cependant le phénomène récent du Deep Learning n'est uniquement le résultat de progrès conceptuels. Comme nous le verrons dans un prochain article, de nombreuses avancées technologiques ainsi que la mise à disposition de nouveaux outils de programmation, promus par les grands acteurs de l'IT, alimentent également cet engouement.

[7] L'origine de cette terminologie de divergence de Kullback Leibler qui mesure le degré de dissimilarité entre deux distributions de probabilité