# Projet MDI343
# Apprentissage profond, réseau de neurones, *etc.*

Antoine BIARD & Vincent BODIN

---

**Résumé**

Ce projet s'attelle à la question de la représentation des données en *machine learning*. Ces représentations, qui peuvent être utilisées ensuite par un algorithme de décision - régression logistique, SVM... - jouent un rôle primordial car porteuses de l'information. A la méthode classique de création de *features* par des connaissances *a priori*, on oppose des méthodes d'extraction de représentation par des structures dites profondes. Notre projet a consisté essentiellement en deux parties : une première partie de compréhension et d'étude des principales méthodes, et une seconde d'implémentation et de test sur quelques unes de ces méthodes (RBM, MLP, DBN).

# Table des matières

# 1 Introduction : de l'importance de la représentation des données

Ces dernières années se sont imposées massivement des méthodes dites d'apprentissage profond - *deep learning* en anglais. La communauté scientifique a accru son intérêt en la matière au vue de leurs résultats, dans des domaines aussi variés que la vision par ordinateur ou la reconnaissance vocale. Atteignant souvent des performances au moins équivalentes à l'état de l'art, de nombreux algorithmes de type profond ont été créés pour tenter d'améliorer encore la représentation.

La représentation est une clef fondamentale dans l'élaboration d'un processus d'intelligence artificielle, en ceci qu'elle concentre l'information des données. On distingue en effet une première étape d'apprentissage de la représentation, puis une seconde d'apprentissage, à partir de cette représentation, d'une décision - pour de la classification par exemple. Les algorithmes effectuant cette dernière tâche sont hautement dépendants des données fournies en entrée. Une amélioration de la représentation entraîne de fait une meilleure performance dans la prise de décision. Nous renvoyons à la deuxième section de [1] pour des quantifications d'amélioration de la performance avec des représentations plus élaborées.

Traditionnellement, les descripteurs - ou encore *features* en anglais - sont extraits de façon dite supervisée à partir des données. Ceci nécessite une connaissance *a priori* dans le domaine, puisque cela requiert souvent une importante étape de traitement des données pour en extraire l'information - citons parmi tant d'autre en vision les points de Harris et les SIFT ou encore le spectrogramme et les *Mel-frequency cepstrum coefficients* en audio. Cela repose sur l'idée que l'homme a la capacité d'extraire et d'organiser l'information intrinsèque des données. Pour autant, dans un cadre d'intelligence artificielle pure, il serait désirable de s'affranchir de cette hypothèse et d'implémenter des algorithmes capables d'apprendre une représentation convenable sans supervision - non-supervisé. En outre, les types de données ayant tendance à devenir de plus en plus variés - image, son, web... - une telle connaissance dans chaque domaine s'avère illusoire et le manque de généralité de ces méthodes se paye dans la nécessité d'expertise dans chaque domaine. Il s'agit donc d'implémenter des méthodes extrayant des données une représentation abstraite.

Une manière d'atteindre ce but est l'extraction d'une représentation par apprentissage profond - *deep learning*. Une telle représentation s'obtient par l'ajout de couches successives. Une première représentation est obtenue par un processus à partir des données pures, puis cette représentation est elle-même raffinée, et ainsi de suite. Dans l'espoir d'obtenir une représentation abstraite, la non-linéarité devient nécessaire puisque sans elle, l'ajout de deux couches linéaires serait équivalent à une unique couche - le concept de profondeur perdrait sens.

Nous allons par la suite détailler les algorithmes d'apprentissage profond les plus classique, puis nous comparerons les performances de certains d'entre eux.

# 2 Revue élémentaire des méthodes de *deep learning*

## 2.1 Réseau de neurones

Un réseau de neurone peut être vu comme l'utilisation en série et parallèle plusieurs neurones simples, cf. Fig.(1). Un neurone est un nœud, avec un certain nombre d'entrées, et une fonction d'activation, disons $\sigma$ - souvent tangente hyperbolique ou une sigmoïde, l'idée étant d'avoir une fonction proche de la fonction seuil, qui donne une activation *soft*. Pour un neurone ayant $p$ entrées, chacune pondérée par un poids $w_i$, et un biais $b$, la sortie du neurone est :

$$h_{w,b}(x) = \sigma(w^T x) = \sigma\left(\sum_{i=1}^{p} w_i x_i + b\right) \tag{1}$$

Dans le cas d'un réseau de neurones, en utilisant une formule de chaîne où chaque sortie de neurone intermédiaire devient l'entrée du neurone suivant, on obtient un réseau du type de la Fig.(1). On voit ici apparaître naturellement l'idée de profondeur dans l'apprentissage par accumulation de couches de neurone, bien qu'il s'agisse du modèle le plus basique d'apprentissage profond. La fonction de sortie dépend de tous les paramètres $w, b$ à chaque couche et chaque nœud. La difficulté majeure dans l'apprentissage profond réside en l'estimation des paramètres optimaux - dans le sens où la sortie est la représentation la plus adéquate.

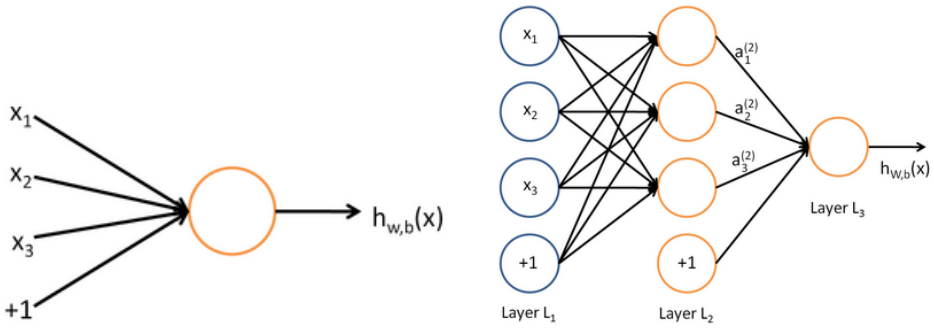


FIGURE 1 – (gauche) Un neurone avec trois entrées $(x_1, x_2, x_3)$ et un *offset* ; (droite) un réseau de neurones de taille $(3, 3, 1)$ avec des *offset* aux deux premières couches.

## 2.2 Multilayer Perceptron (MLP)

Un modèle classique en apprentissage profond est le perceptron multi-couches (MLP). Il s'agit du premier exemple d'algorithme utilisant la méthode dite de *backpropagation*. Bien que peu utilisé seul en pratique, car la fonction d'optimisation n'est pas convexe et il y a de grands risques de trouver un optimum local et non global, le MLP correspond aux prémices du *deep learning* et sera réutilisé pour des modèles plus complexes par la suite.

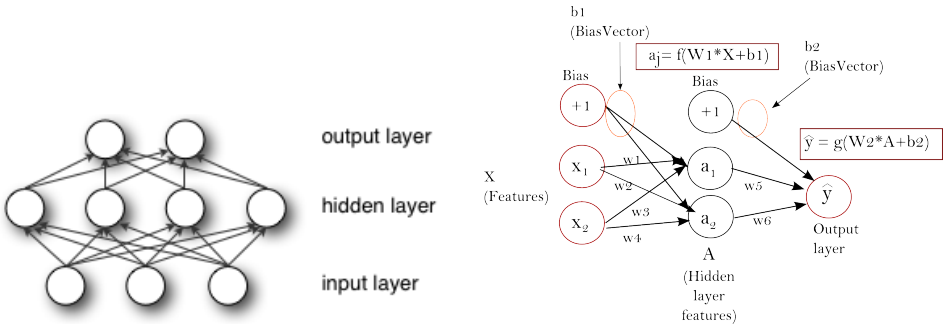


FIGURE 2 – (gauche) Un MLP avec une seule couche de variables cachées ; (droite) structure de MLP avec l'ajout de biais et les poids.

Le MLP possède une fonction d'activation $\sigma$ - tangente hyperbolique ou la fonction logistique souvent - des nœuds de biais $b$ et des poids $W$ associés à chaque arrête. Pour chaque couche, le paramètre est donc $\theta = (W, b)$, et pour une entrée $x$ de dimension $d$ - *i.e.* $d$ nœuds - la sortie de cette couche est :

$$s(x) = \sigma(Wx + b) \tag{2}$$

Ceci forme un vecteur de taille disons $m$ - sur la Fif.(2) par exemple pour la première couche $d = 3$ et $m = 4$ - qui est envoyé comme entrée à la couche suivante.

La nouveauté du MLP comparé au réseau de neurone réside en la manière d'apprendre l'hyper-paramètre $\theta = (W_1, \cdots, W_p, b_1, \cdots, b_p)$ - $p$ est le nombre de couches. L'algorithme en question sera développé plus tard mais nous en donnons un aperçu. L'initialisation s'effectue classiquement de manière aléatoire - bien que dans le cas du DBN on lui envoie des poids définis en entrée. Dans [2], il est expliqué comment tirer les poids initialement, nous y reviendrons. L'algorithme contient ensuite deux étapes :

**Forward pass.** On part des variables visibles $v$ et on remonte le graphe avec les poids de la structure. On en déduit une représentation.

**Backpropagation.** A partir de cette représentation, on effectue le chemin inverse en descendant le graphe et en comptabilisant les erreurs commises. En comparant les résultats de la descente avec les variables visibles, on peut mettre à jour les poids.

Cela est répété un certain nombre de fois et cette mise à jour par *backpropagation* permet d'obtenir une représentation convenable. Toutefois ceci est très dépendant de l'initialisation.

Notons que ceci ne fournit qu'une représentation. Pour effectuer une classification, on utilise souvent une base de données étiquetée. Chaque représentation est associée à une étiquette et on se sert des représentations comme *feature* de l'algorithme de décision - régression logistique, SVM...

## 2.3 Restricted Boltzmann Machine (RBM)

### 2.3.1 Machine de Boltzmann

Une machine de Boltzmann est un réseau de variables aléatoires couplées. On se restreint au cas où les données sont binaires, on note les variables visibles $v$ et les cachées $h$. Une machine de Boltzmann est représentée en Fig.(3) à gauche.

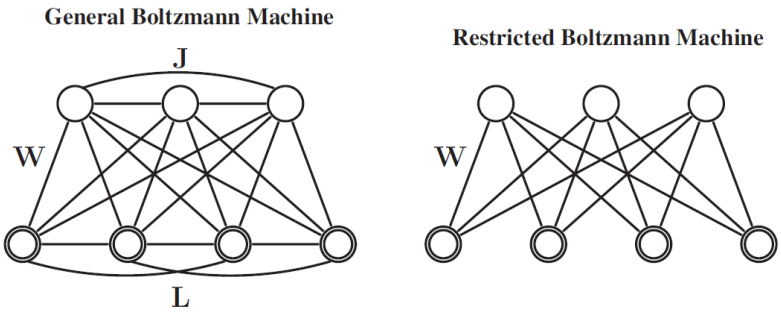


FIGURE 3 – (gauche) Une machine de Boltzmann générale ; (droite) machine de Boltzmann restreinte (RBM).

L'énergie des états $(v, h)$ prend alors la forme :

$$E(v, h; \theta) = -v^T W h - \frac{1}{2} v^T L v - \frac{1}{2} h^T J h \tag{3}$$

(où l'on ne prend pas en compte ici les termes de biais qui correspondent à un recentrage des $(v, h)$). Il s'agit d'un cas de *energy-based models* (EBM). Dans ce cas, la probabilité d'un état $v$ s'écrit :

$$p(v; \theta) = \frac{1}{\mathcal{Z}} \sum_h e^{-E(v,h;\theta)} \tag{4}$$

où $\mathcal{Z} = \sum_{x,h} e^{-E(v,h;\theta)}$ est la fonction de partition. Ces modèles ont la propriété que le gradient de $\log p(v)$ - log-vraisemblance - se décompose en deux termes, une phase positive et une négative. Pour autant la vraisemblance n'est pas calculable car exponentielle à la fois en le nombre de variables cachées et visibles.

### 2.3.2 Restricted Boltzmann Machine

Comme montré dans la Fig.(3), un RBM est une machine de Boltzmann où les interactions inter-couches sont gelées. De par la forme de son graphe, le RBM a des propriétés d'indépendance conditionnelle qui rendent l'inférence facile en pratique :

$$\begin{aligned} p(h|v) &= \prod_i p(h_i|x) \\ p(v|h) &= \prod_j p(x_j|h) \end{aligned} \tag{5}$$

Par suite on a :

$$\begin{aligned} p(h_i = 1|v) &= \sigma\left(\sum_j W_{ji}x_j + d_i\right) \\ p(x_j = 1|h) &= \sigma\left(\sum_i W_{ji}h_i + b_j\right) \end{aligned} \tag{6}$$

où $\sigma$ est la fonction d'activation, et les $(b, d)$ sont les biais. Pour autant, comme la plupart du temps, la fonction de partition reste incalculable en pratique, mais ces formules de factorisation permettent d'effectuer de l'inférence sans avoir à la calculer.

Nous en reparlerons plus en détail dans la partie d'implémentation, mais l'entraînement d'un RBM nécessite d'effectuer des étapes d'échantillonnage - de Gibbs - souvent par des méthodes MCMC. Il existe plusieurs méthodes pour accélérer le processus, nous parlerons de la *Contrastive Divergence* (CD) et du *Stochastic Maximum Likelihood* (SML).

## 2.4 Deep Belief Networks (DBN)

Un DBN est une structure d'apprentissage de représentation qui consiste en un empilement de RBM. Le premier RBM prend en entrée les variables visibles - données du problème - et, par échantillonnage en extrait un ensemble de variables cachées. Celles-ci sont alors utilisées comme entrée de la seconde couche et ainsi de suite, cf. Fig.(4). L'entraînement est donc de type *greedy*. Finalement, la probabilité jointe entre $x$ et $h$ s'écrit :

$$p(x, h^1, \cdots, h^l) = \left(\prod_{k=0}^{l-2} p(h^k|h^{k+1})\right) p(h^{l-1}, h^l) \tag{7}$$

avec $x = h^0$. Nous verrons l'implémentation de l'algorithme en détail plus tard. En modèle graphique, cette décomposition s'interprète comme un graphe dont la dernière couche est non-orientée tandis que toutes les précédentes sont des graphes orientés.

L'entraînement du DBN repose en outre sur un *fine-tuning* des paramètres obtenus par apprentissage par couches sur les RBM. Le *fine-tuning* s'effectue soit selon un *proxy* de la log-likelihood - cas non-supervisé - ou d'un critère d'erreur dans le cas supervisé. Nous nous plaçons dans le cas supervisé, et la méthode de classification appliquée est le MLP.

Nous pouvons donner ici une justification rapide du *pre-training* par couche. Dans le cas de deux couches par exemple, il a été montré [3] que :

$$\log p(x) = \mathrm{KL}(Q(h^{(1)}|x)\|p(h^{(1)}|x)) + \mathrm{H}_{Q(h^{(1)|x)}} + \sum_h Q(h^{(1)}|x)\left(\log p(h^{(1)}) + \log p(x|h^{(1)})\right) \tag{8}$$

où KL est la divergence de Kulbac-Leibler, H l'entropie, $Q(h^{(1)}|x)$ la distribution *a posteriori* du premier RBM (appris seul) et $p(h^{(1)}|x)$ celle après l'apprentissage entier du DBN. Si l'initialisation est $W^{(2)} = W^{(1)T}$ alors $Q(h^{(1)}|x) = p(h^{(1)}|x)$ et la divergence est nulle. Fixer ensuite $W^{(1)}$ et optimiser selon $W^{(2)}$ ne peut qu'augmenter la log-vraisemblance.

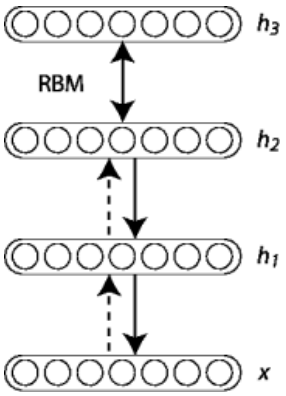


FIGURE 4 – Un *deep belief network* (DBN), la dernière couche est non orientée, tandis que toutes les autres le sont.

## 2.5 Deep Boltzmann Machine (DBM)

Un DBM est une structure profonde qui, en première approche tout au moins, ressemble fortement à un DBN. De la même manière que le DBN, le DBM extrait une représentation abstraite des données. Également, un faible nombre de données étiquetées est nécessaire à la tâche de classification, ce qui peut être intéressant dans certains cas. Il est dit dans [4] que l'inférence dans les DBM est plus robuste que dans les DBN car les entrées ambiguës ont moins d'impact.

Un DBM est constitué de RBM empilés comme dans le cas des DBN. L'étape de *pre-training* se résume à un entraînement couche par couche - *greedy* - de ces RBM. L'idée pour autant est de modifier un peu cette étape pour éviter le double passage quand on fait les étapes de haut vers le bas et bas vers le haut. Ainsi, pour le niveau le plus bas les entrées sont doublées avec les poids visibles-cachés inchangés, cf. Fig.(5). Une étape d'évaluation du DBM basée sur des méthode de Monte Carlo - *Annealed Importance Sampling* - a ensuite lieu. Finalement, on termine comme dans le cas du DBN par du *fine-tuning* avec un MLP par exemple - qui converti la représentation en étiquette.

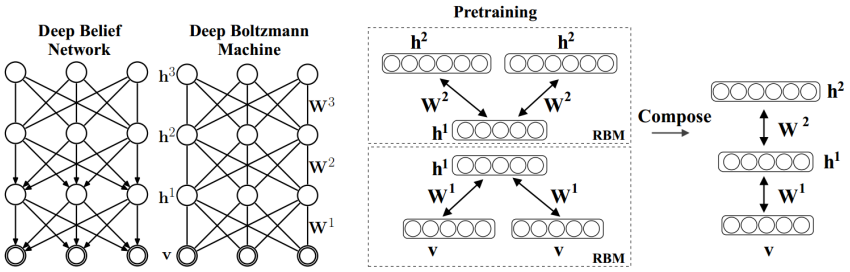


FIGURE 5 – (extrême gauche) Un DBN avec sa dernière couche non orientée ; (gauche) graphe de DBM, toutes les couches sont non orientées ; (droite) *pre-training* des RBM couche par couche ; (extrême droite) composition des couches pour former le DBM.

### 2.6 Auto-Encoder

La dernière méthode que nous présentons succinctement est celle des *auto-encoder*. Partant d'un ensemble d'apprentissage $x^{(t)}$ on extrait des descripteurs $h^{(t)}$ à l'aide d'une fonction paramétrique $f_\theta(x)$. Une autre fonction $g_\theta$ est appelé le décodeur et permet de faire la transformation inverse, $r = g_\theta(h)$ s'appelle la reconstruction. Une erreur de reconstruction est définie $L(x^{(t)}, r)$. Finalement, l'entraînement de l'auto-encodeur consiste à trouver le paramètre $\theta$ vérifiant :

$$\mathcal{J}_{\mathrm{DAE}}(\theta) = \sum_t L(x^{(t)}, g_\theta(f_\theta(x^{(t)}))) \tag{9}$$

La forme la plus usuelle pour les fonctions $f_\theta$ et $g_\theta$ est :

$$\begin{array}{rcl} f_\theta(x) & = & s_f(b + Wx) \\ g_\theta(h) & = & s_g(d + W'h) \end{array} \tag{10}$$

où typiquement $s_f$ et $s_g$ sont des sigmoïdes ou des tangentes hyperboliques.

## 3 Implémentation et résultats

Dans cette partie nous allons présenter les différents choix que nous avons pris afin d'implémenter un DBN. Puis nous présenterons l'implémentation complète du graphe, et enfin nous commenterons les résultats obtenus.

### 3.1 Données et algorithmes élémentaires

Dans la littérature il existe un grand nombre d'algorithmes afin d'optimiser un DBN. Nous allons ici présenter rapidement les briques qui composent classiquement les algorithmes d'apprentissage de DBN.

#### 3.1.1 Les données

Afin de tester notre algorithme, nous avons décidé d'utiliser la base de données MNIST. Cette base de données a l'avantage d'être une référence et nous avons ainsi une très bonne quantification des performances de notre implémentation. Elle permet aussi de réaliser un apprentissage multiclasse. De plus, cette base a été le témoin de la puissance de l'apprentissage profond (réseau de neurones à convolution). La structure des chiffres permet en effet d'envisager une représentation à avec plusieurs niveau d'abstraction. Par exemple, la différence entre un 1 et un 7 ou un 6 et un 4 peut sembler ambiguë, et un classifieur prenant en entrée les vecteurs des pixels risque de ne pas être très pertinent. Pour autant, construire des structures profondes extrait des représentations linéairement séparables de ces chiffres.

Le temps de calcul nécessaire à ces algorithmes est très long. Nous avons donc choisi de réaliser la plupart de nos tests sur une base réduite extraite de la base MNIST que nous avons chargée grâce à la fonction `sklearn.datasets.load_digits` de scikit-learn. Cette base contient 1797 images de chiffres de dimension $8 \times 8$. Afin d'augmenter la taille de la base, nous avons utilisé une technique classique qui consiste à réaliser des translations de chaque chiffre afin d'augmenter la base par un facteur 5.

Nous avons aussi essayé de lancer notre algorithme sur la véritable base de données dont les images sont de taille $28 \times 28$ et est constituée de plus de 50000 exemples. Nous présenterons les résultats obtenus sur cette base.

### 3.1.2 Implémentation du RBM

La structure essentielle de notre rapport est le RBM. La principale difficulté de cette structure est le calcul du gradient qui peut se calculer analytiquement. Nous rappelons que le principe des RBM est de minimiser la log-vraisemblance qui s'exprime de cette manière :

$$\log\left(P(x;\theta)\right) = \log \sum_h \mathrm{e}^{-E(x,h;\theta)} - \log \sum_{x,h} \mathrm{e}^{-E(x,h;\theta)} \tag{11}$$

Cette expression se dérive par rapport à $\theta$ :

$$\begin{aligned}
\frac{\partial \log\left(P(x;\theta)\right)}{\partial \theta} &= \frac{\log \sum_h \mathrm{e}^{-E(x,h;\theta)}}{\partial \theta} - \frac{\partial \log \sum_{x,h} \mathrm{e}^{-E(x,h;\theta)}}{\partial \theta} \\
&= -\frac{1}{\sum_h \mathrm{e}^{-E(x,h;\theta)}} \sum_h \mathrm{e}^{-E(x,h;\theta)} \frac{\partial E(x,h;\theta)}{\partial \theta} \\
&\quad + \frac{1}{\sum_{x,h} \mathrm{e}^{-E(x,h;\theta)}} \sum_{x,h} \mathrm{e}^{-E(x,h;\theta)} \frac{\partial E(x,h;\theta)}{\partial \theta} \\
&= -\sum_h P(h|x) \frac{\partial E(x,h;\theta)}{\theta} + -\sum_{x,h} P(x,h) \frac{\partial E(x,h;\theta)}{\partial \theta}
\end{aligned} \tag{12}$$

La première partie du gradient est appelée phase positive, la seconde est la phase négative. Afin de calculer la première partie il faut échantillonner selon la loi $P(h|x)$. Comme présentée dans l'article [?]. cette partie est calculée à partir d'un exemple $x$ de la base d'entrainement. La seconde partie en revanche demande d'échantillonner selon la loi jointe de $x$ et $h$. Elle ne peut être calculée que approximativement, et on aimerait que l'échantillonnage soit réalisé à partir du modèle courant. Il existe principalement deux méthodes afin d'échantillonner selon la loi jointe $P(x,h)$. Toutes sont des dérivées des méthodes MCMC trop longues pour être appliquées à chaque étape.

**Contrastive Divergence.** Cette méthode consiste à échantillonner $x$ et $h$ grâce à un très court échantillonneur de Gibbs initialisé par un exemple $x$ de la base d'entrainement. La pratique a montré que le CD-1 (une seule étape) est souvent meilleure. L'idée serait que les échantillons risquent d'être très biaisés, mais qu'ils sont échantillonnés selon le modèle et qu'ils permettent au gradient d'être stable.

**Persistent Contrastive Divergence.** Cette méthode est très proche de la *Contrastive Divergence*. En effet, à chaque calcul d'un nouvel échantillon, plutôt que de prendre un exemple de la base d'entraînement, l'algorithme récupère l'échantillon $x$ obtenu par échantillonnage de l'étape précédente. Ainsi l'algorithme implémente un échantillonneur de Gibbs à travers les différentes mises à jour du RBM. Cette méthode permet de limiter le biais de l'échantillonneur, mais permet comme pour la *Contrastive Divergence* de conserver une continuité entre les échantillons.

Après plusieurs tentatives d'implémentation peu efficace en python (lenteur et mémoire), nous avons essayé la librairie de Theano qui n'est cependant pas très rapide uniquement sous CPU - et l'option GPU n'est pas entièrement stable sous Windows. Nous avons ainsi utilisé l'implémentation de scikit-learn `sklearn.neural_network.BernoulliRBM` qui implémente une *Persistent Contrastive Divergence*.

### 3.1.3 Implémentation du MLP

Nous avons implémenté l'algorithme le plus classique dans le cas de l'entraînement du MLP : la *backpropagation*. Cet algorithme se décompose en deux étapes, une première de propagation, et une seconde de mise à jour des poids.

**Initialisation.** L'initialisation est cruciale dans ces algorithmes de *backpropagation*. De plus, comme nous allons utiliser le MLP dans le cas de poids initialisés par empilement de RBM - pour le DBN - notre implémentation doit prendre en compte deux possibilités. Soit on lui envoie les poids, soit on utilise le MLP à part entière et dans ce cas là les poids doivent être initialisés. Dans [2], il est expliqué que les poids doivent vérifiés (si $n_1$ et $n_2$ indiquent le nombre d'éléments au couches d'entrées et de sorties respectivement) :

$$\begin{cases} \left[-\sqrt{\frac{6}{n_1+n_2}}, \sqrt{\frac{6}{n_1+n_2}}\right] & \text{pour activation en tanh} \\ \left[-4\sqrt{\frac{6}{n_1+n_2}}, 4\sqrt{\frac{6}{n_1+n_2}}\right] & \text{pour activation en sigmoïd} \end{cases} \tag{13}$$

**Propagation.** La propagation correspond à une étape de montée du graphe pour générer les sorties correspondantes puis à une étape de descente qui génère des $\delta$ accumulant les erreurs commises. Ceci nécessite une étiquette $y$ que l'on compare avec la variable cachée finale $h$ - éventuellement ceci peut être vectoriel. Nous verrons dans Alg.(1) l'algorithme précisément.

**Mise à jour des poids.** Pour chaque synapse on met à jour les poids avec les $\delta$ calculés par propagation. Ceci nécessite un paramètre, le *learning rate* qui influence fortement la qualité et la vitesse de l'apprentissage.

La prédiction dans le cas d'un MLP consiste uniquement à, une fois les poids établis, utiliser les données et monter le graphe pour retrouver leur représentation. En ce sens, on considère un MLP comme un pur outils de représentation - vision non-paramétrique. On peut toutefois lui rajouter une régression logistique à l'issue - lorsque l'on dispose de données supervisées - qui permet d'en faire un outils de classification supervisée. Nous avons testé le MLP sur des exemples simples - bases de données naïves créées par nous-même et étiquette associé. Les résultats étaient toujours très concluants et rapide.

On peut lancer le MLP seul comme processus de représentation sur la base de données MNIST - la petite base de données. Les résultats sont fournis en Fig.(**??**). On voit que les taux de classifications ne sont pas absurdes mais restent relativement faibles, nous verrons que le *pre-training* par RBM change ceci - dans le bon sens.

**Algorithm 1** Entraînement du Multi-Layer Perceptron (MLP)

**Input:** $X$ : matrice d'entrainement, $y$ : étiquettes d'entraînement, $l_r$ : *learning_rate*, *epochs* : nombre d'itérations, ($W$ : liste de matrice de poids)
**Output:** $W$ : liste de matrice de poids optimaux

```
Rajoute une colonne de 1 à la matrice X (biais),
for k ∈ range(epochs) do
    states = [X[i]] où i est un indice de ligne tiré aléatoirement,
    for j monte le graphe (propagation) do
        states.append(states[j] · W[j])
    end for
    error = y[i] - states[end]
    δ = [error * ∂σ(states[end])]
    for j descend le graphe (backpropagation) do
        deltas.append((deltas[end] · W[j]^T) × ∂σ(states[j]))
    end for
    δ.revese()
    for i monte le graphe (mise à jour) do
        W[i] += l_r(vecteur de nombres de composant^T · δ)
    end for
end for
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.90 | 0.96 | 0.93 | 174 |
| 1 | 0.67 | 0.58 | 0.62 | 184 |
| 2 | 0.86 | 0.87 | 0.86 | 166 |
| 3 | 0.75 | 0.74 | 0.75 | 194 |
| 4 | 0.86 | 0.85 | 0.86 | 186 |
| 5 | 0.75 | 0.80 | 0.77 | 181 |
| 6 | 0.90 | 0.94 | 0.92 | 207 |
| 7 | 0.84 | 0.90 | 0.87 | 154 |
| 8 | 0.80 | 0.65 | 0.72 | 182 |
| 9 | 0.69 | 0.78 | 0.73 | 169 |
| avg / total | 0.80 | 0.80 | 0.80 | 1797 |

FIGURE 6 – Résultats avec le MLP seul et la petite base MNIST - méthode de classification SVM

## 3.2 Le processus complet

Le processus est décrit par le pseudo code 2. Il consiste dans un premier temps en l'entrainement des différents RBM. Le premier est entrainé à partir de la base d'entrainement. Pour les RBM suivants, on échantillonne des variables cachées à partir de l'état du précédent RBM. Dans notre processus, nous avons utilisé pour chaque échantillon de la nouvelle base d'entrainement une chaine de Gibbs de longueur 1. Ce procédé est discutable, mais il semble cohérent avec les méthodes couramment utilisées. Une fois l'intégralité des RBM entrainés, nous initialisons le MLP grâce aux poids des RBM puis nous l'entrainons. Après cette étape de *fine-tuning*, nous entraînons la régression logistique. Des tests de classifications sont alors réalisés, ils sont décrits dans la section suivante.

**Algorithm 2** Entrainement d'un DBN

```
H ← digits
for currentRBM in RBMs do
    Entrainer currentRBM à partir de H
    if currentRBM n'est pas le dernier then
        Créer une base d'entrainement en échantillonnant des variables cachées
H à partir du modèle de previousRBM et de H
    end if
    previousRBM ← currentRBM
end for
MLP.weights ← RBMs.weights
Entrainement du MLP à partir de digits
Entrainement de la regression logistique
Test à partir d'une base de test
```

## 3.3 Les résultats

Nous présentons ici les résultats obtenus sur différentes bases de données.

### 3.3.1 Comparaison RBM et DBN

Par *GridSearch*, les paramètres pour l'apprentissage d'un RBM seul nous donne un *learning rate* égal à 0.04, et un nombre d'itérations égal à 20. Nous utilisons 100 variables cachées. Les résultats de ce RBM seul donnent les résultats présentés dans le tableau 3.3.1.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 0.99 | 0.99 | 174 |
| 1 | 0.92 | 0.95 | 0.93 | 184 |
| 2 | 0.95 | 0.98 | 0.97 | 166 |
| 3 | 0.97 | 0.91 | 0.94 | 194 |
| 4 | 0.97 | 0.95 | 0.96 | 186 |
| 5 | 0.93 | 0.93 | 0.93 | 181 |
| 6 | 0.98 | 0.97 | 0.97 | 207 |
| 7 | 0.95 | 1.00 | 0.97 | 154 |
| 8 | 0.90 | 0.88 | 0.89 | 182 |
| 9 | 0.91 | 0.93 | 0.92 | 169 |
| avg / total | 0.95 | 0.95 | 0.95 | 1797 |

FIGURE 7 – Résulats obtenus après apprentissage d'un RBM puis d'une régression logistique sur la petite base de données des digits.

On peut comparer les résultats obtenus avec ceux que l'on obtient avec notre apprentissage d'un DBN à deux couches (figure 3.3.1).

Le fait que nos résultats de DBN ne soit pas aussi bons que ceux du RBM peut venir de deux facteurs. D'une part, l'optimisation du RBM est complète - avec l'outil de `scikit` - tandis que celle du DBN demande beaucoup plus d'attention car le nombre de paramètre est beaucoup plus grand et tous nécessite d'être optimiser conjointement ce qui rend - du moins sur nos machines - la possibilité de *grid search* limitée. D'autre part, notre échantillonnage pour la

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 0.99 | 0.99 | 174 |
| 1 | 0.94 | 0.92 | 0.93 | 184 |
| 2 | 0.93 | 0.98 | 0.95 | 166 |
| 3 | 0.94 | 0.91 | 0.92 | 194 |
| 4 | 0.97 | 0.91 | 0.94 | 186 |
| 5 | 0.94 | 0.91 | 0.92 | 181 |
| 6 | 0.98 | 0.95 | 0.96 | 207 |
| 7 | 0.93 | 0.98 | 0.96 | 154 |
| 8 | 0.86 | 0.92 | 0.89 | 182 |
| 9 | 0.89 | 0.93 | 0.91 | 169 |
| avg / total | 0.94 | 0.94 | 0.94 | 1797 |

FIGURE 8 – Résulats obtenus après apprentissage d'un DBN à deux couches puis d'une régression logistique sur la petite base de données des digits.

base d'entraînement du deuxième RBM est probablement trop biaisé - on utilise une CD-1 alors que l'on pourrait faire une chaîne de Gibbs plus longue.

### 3.3.2 Apprentissage sur la vraie base

L'apprentissage sur la vraie base de données est très long. Ainsi nous n'avons pas pu optimiser les paramètres nous-mêmes. Nous avons utilisé l'ordre de grandeur des paramètres obtenus par *GridSearch* sur www.deeplearning.net. Pour les RBMs on prend un *learning rate* égal à 0.01, et nous les initialisons avec 1000 variables cachées chacune. L'apprentissage a duré environ 6h, et nous obtenons les résultats de la figure 3.3.2.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0.0 | 0.97 | 0.98 | 0.98 | 1312 |
| 1.0 | 0.99 | 0.98 | 0.98 | 1604 |
| 2.0 | 0.95 | 0.96 | 0.95 | 1348 |
| 3.0 | 0.94 | 0.95 | 0.95 | 1427 |
| 4.0 | 0.97 | 0.97 | 0.97 | 1362 |
| 5.0 | 0.96 | 0.93 | 0.94 | 1280 |
| 6.0 | 0.97 | 0.98 | 0.97 | 1397 |
| 7.0 | 0.96 | 0.95 | 0.96 | 1461 |
| 8.0 | 0.94 | 0.96 | 0.95 | 1390 |
| 9.0 | 0.94 | 0.94 | 0.94 | 1419 |
| avg / total | 0.96 | 0.96 | 0.96 | 14000 |

FIGURE 9 – Résulats obtenus après apprentissage d'un DBN à deux couches puis d'une régression logistique sur la grande base de données des digits.

On voit que ces résultats sont concluants. Obtenir un taux de 0.96 avec une méthode linéaire telle que la régression logistique montre une propriété fondamentale de notre représentation apprise : elle a tendance à rendre linéairement séparables les groupes de *digits* dans l'espace des représentation. Notons tout de même que nous sommes loin de l'état de l'art (atteint avec des *convolutionnal network*), mais peut-être que l'optimisation des paramètres - nécessitant des

machines plus puissantes que les nôtres - aurait pu faire augmenter significativement nos résultats.

# 4 Conclusion

Dans un environnement où l'intelligence artificielle doit faire face à des données de plus en plus hétéroclites et où l'homme n'a pas de connaissance *a priori*, la représentation abstraite semble prometteuse. Nous avons vu ici une méthode parmi tant d'autre, celle de l'apprentissage profond. Certaines représentations - comme les *convolution network* qui nécessitent une structure de signal - sont des alternatives à ce que nous avons étudié.

En première partie, nous avons étudié les méthodes principales de *deep learning*, et avons essayé de donner une idée intuitive de comment cela fonctionnait. Nous nous sommes ensuite intéressé un peu plus en profondeur à certains algorithmes, nous avons en particulier étudié les RBM, MLP et DBN. Les résultats fournis nous montrent que ces représentations ne sont pas absconses, en particulier qu'elles ont tendance à rendre les représentations linéairement séparables ce qui trivialise les algorithmes de classification. Pour autant elles nécessitent une grosse phase d'apprentissage et de réglage des paramètres. Ceci peut être rédhibitoire dans certains cas et surtout ne supporte pas les calculs *on-line*.

## Références

[1] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning : A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[2] Yoshua Bengio and Xavier Glorot. Understanding the difficulty of training deep feedforward neural networks. 9 :249–256, May 2010.

[3] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, pages 504–507, 2006.

[4] Ruslan Salakhutdinov and Geoffrey Hinton. Deep Boltzmann machines. 5 :448–455, 2009.