Rapport projet long
Équipe HAL9000

Hamelain Christian, Hasan Pierre-Yves, Gaspart Quentin, Trestour Fabien

14 juin 2016

Table des matières

I Introduction **3**
0.1 Contexte 4
0.2 Présentation du sujet 4
0.3 Déroulement 4

II Le Perceptron **5**

1 Architecture d'un perceptron **6**
1.1 Introduction 6
1.2 La brique de base : le neurone 6
1.3 Organisation des couches 8

2 Algorithmes d'apprentissage **11**
2.1 L'objectif des algorithmes d'apprentissage 11
2.2 La méthode de rétropropagation du gradient 12
2.3 test 12

3 Influence des paramètres sur l'apprentissage. **14**
3.1 Paramètres perceptron 14

III Machines de Boltzmann **20**

4 Introduction **21**

5 Les machines de Boltzmann restreintes **22**
5.1 Présentation et éléments de preuve 22
5.2 Loi d'apprentissage 23

6 Les Deep Belief Networks **25**
6.1 Principe des Deep Boltzmann Machines 25
6.2 Spécificité de l'apprentissage 25
6.3 Résultats 26

7 Aller plus loin **28**
7.1 La visualisation des spectres du réseau 28
7.2 Discrimination grâce aux RBM 30
7.3 Reconnaissance d'image 32

A La base MNIST **35**
A.1 Présentation de la base de donées 35
A.2 Limites de cette base 36

B Présentation du code. **37**
B.1 Code du perceptron 37
B.2 Code de la RBM et du DBN 37

Première partie

Introduction

0.1 Contexte du projet

CentraleSupélec propose à ses étudiants en deuxième année de participer à des projets longs qui se réalisent en équipe, tout au long de l'année sur un thème mêlant plusieurs compétences que l'ingénieur se doit de posséder.

Dans notre cas, c'est l'élaboration de réseaux neuronaux et les méthodes de Deep Learning que 12 élèves ont étudié, répartis en 3 groupes, dont HAL9000.

Ce rapport présente les avancement de ce dernier groupe sur le sujet tout au long de l'année.

0.2 Présentation du sujet

Ce projet aborde le sujet du Deep Learning. Cette méthode spécifique de machine learning est dérivée du concept de réseau de neurones.

Les réseaux de neurone sont une modélisation simple du fonctionnement cérébral à l'échelle cellulaire. Cette approche a vu le jour avec les études de McCulloch et Pitts dès la fin des années 50. Malgré certains écueils dans les années 70, l'approche connexioniste des réseaux de neurones a su se développer et devenir un sujet de recherche populaire dans les dernières années, notamment grâce aux capacités d'adaptabilité et de généralisation de ces réseaux qui en font d'excellents candidats pour des applications telles que la reconnaissance d'image ou la classification.

0.3 Déroulement du projet

Tout d'abord, il s'agissait pour nous de comprendre le fonctionnement de ces structures et commencer à coder des structures élémentaires afin de comprendre les enjeux du deep learning. Nous avons pour cela travaillé avec la base de données MNIST de Yann LeCun et en JAVA. Afin de travailler en groupe de manière efficace, nous avons aussi utilisé l'outil GIT.

Par la suite nous nous sommes intéressés aux architectures profondes, chaque groupe se penchant sur une architecture spécifique. Notre équipe s'est en particulier intéressée aux machines de Boltzmann, en commençant par les machines de Boltzmann restreintes (RBM), puis la structure de Deep Learning associée, les Deep Belief Networks.

Cette étude a été encadrée par Joanna Tomasik et Arpad Rimmel, enseignants à CentraleSupélec, campus de Gif-sur-Yvette.

Deuxième partie

Première approche du problème : le Perceptron

Chapitre 1

Architecture d'un perceptron

1.1 Introduction

Ici, l'objectif était la reconnaissance des caractères de la base de données MNIST grâce à un perceptron.

Le perceptron fait partie des architectures de réseaux neuronaux les plus simples. Son étude nous a donc permis de s'introduire à la problématique du machine learning avant d'approfondir en étudiant des structures plus complexes.

1.2 La brique de base : le neurone

Le neurone est le composant élémentaire des réseaux neuronaux. Il est une modélisation du fonctionnement des neurones du système nerveux humains.

Chaque neurone reçoit un signal via une entrée, qui correspond aux dendrites des systèmes biologiques. Le neurone prend en compte la valeur de toutes ses entrées et en déduit la valeur de sortie. Cette sortie est ensuite propagée par le biais d'un axone vers un autre neurone.

La sortie du j$^{\text{ième}}$ neurone est donnée par la formule :

$$s_j(x) = f(\sum_{k=1}^{N} w_{j,k} * x_k + w_0) \tag{1.1}$$

où :
- s est la valeur de la sortie
- f est la fonction d'activation.
- N est la dimension du vecteur d'entrée.
- $w_{j,k}$ est la k$^{\text{ième}}$ composante du vecteur de poids w_j du j$^{\text{ième}}$ neurone. w_0 est le biais du neurone. Cette valeur correspond au poids d'une entrée fictive valant toujours 1.
- x_k est la k$^{\text{ième}}$ composante du vecteur d'entrée x.

L'entrée x est un vecteur défini par un ensemble de caractéristiques que l'on choisit pour représenter les données d'entrées. Dans le cas d'une image par exemple, ce vecteur d'entrée peut être composé de la valeur de tous les pixels

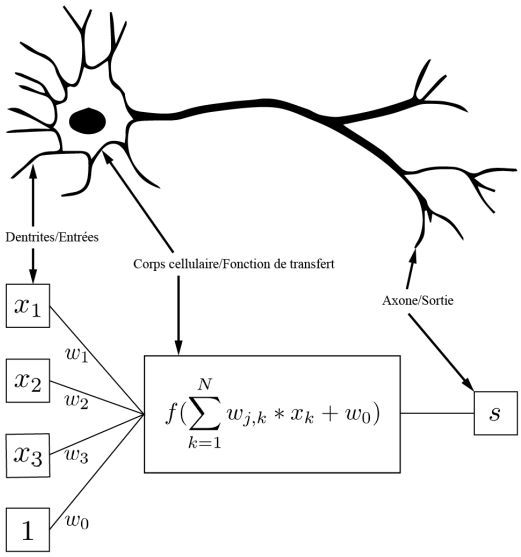

FIGURE 1.1 – Analogie entre neurone biologique et neurone formel.

de l'image. D'autres représentations moins triviales peuvent être choisies afin de réduire la quantité d'information à traiter et de s'approcher au mieux des données significatives de la problématique traitée.

La fonction d'activation permet de définir le comportement du neurone. Selon la définition de cette fonction on aura une sortie à valeurs discrètes ou continues, centrées en 0 ou en 0,5. Le choix de la fonction est donc étroitement lié avec le problème à traiter. Il existe différents types de fonctions d'activation. Parmi les plus utilisées figurent :

— La fonction échelon

$$f(x) = \mathbf{1}_{\mathbb{R}_+^*} \tag{1.2}$$

— Les fonctions linéaires

$$f(x) = \alpha * x + \beta \tag{1.3}$$

— La fonction sigmoïde

$$f(x) = \frac{1}{1 + e^{-\lambda * x}} \tag{1.4}$$

— La fonction tangente hyperbolique

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.5}$$

Le vecteur de poids du j$^{\text{ième}}$ neurone représente la pondération de chacune des entrées de ce neurone. C'est ce paramètre qui permet de modifier le réseau de neurone sans avoir à modifier son architecture. Toutes les propriétés d'un réseau neuronal sont donc issues de ce vecteur de pondération et de ses variations.

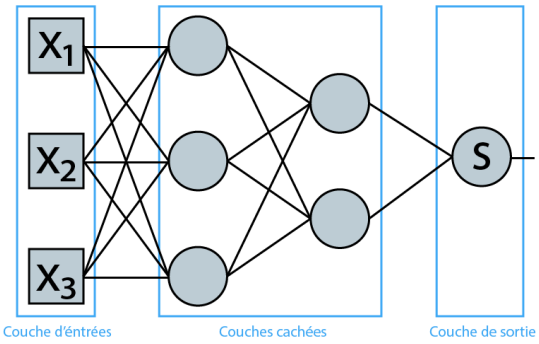

FIGURE 1.2 – Architecture d'un perceptron multi-couches.

1.3 Organisation des couches

Il existe de nombreuses architectures de réseau de neurone. Elles peuvent être récurrentes ou non, entièrement connectées ou seulement partiellement, organisées en couches, ... De nombreuses propriétés permettent de caractériser une architecture de réseau de neurone.

Le Perceptron est un modèle assez élémentaire de réseau. Il est constitué en couches totalement connectées.

On peut distinguer trois types de couches : la couche d'entrée, les couches cachées et la couche de sortie.

La couche d'entrée est assez élémentaire. C'est tout simplement une couche dont la valeur sera le vecteur d'entrée x. Cette couche doit donc être constituée d'autant de neurones que le vecteur d'entrée a de dimensions. Les neurones de cette couche ont pour fonction d'activation l'identité.

Les couches cachées sont celles qui effectuent les calculs. Les principales propriétés du réseau sont héritées de cet empilement de couches. On comprend alors mieux l'intérêt actuel pour le Deep Learning, c'est-à -dire l'apprentissage des réseaux avec un grand nombre de couches cachées. Le nombre de neurones dans chaque couche et les fonctions d'activation utilisées par les neurones sont choisies par le concepteur du réseau selon le problème traité par le réseau.

Afin d'expliciter l'importance de l'architecture du réseau, abordons l'exemple usuel du ou exclusif.

Essayons de réaliser avec un unique neurone la fonction logique XOR. Ce neurone aura deux entrées x_1 et x_2 à valeurs dans $\{0\,;1\}$ et une sortie s, elle aussi à valeurs dans $\{0\,;1\}$. On prendra comme fonction d'activation la fonction de Heaviside, c'est à dire $\mathbf{1}_{\mathbb{R}_+^*}$. Ce neurone aura par conséquent un fonctionnement totalement binaire. Il s'agit donc ici de déterminer les pondérations w_1 et w_2, respectivement associées aux entrées x_1 et x_2, qui conviennent pour obtenir en sortie la valeur $x_1 \oplus x_2$.

Pour un tel problème, le neurone agit comme un séparateur linéaire de l'espace des entrées. L'équation de la droite séparant le demi-espace définit par

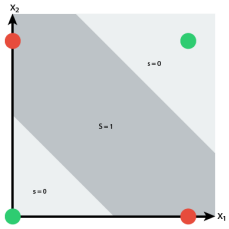

FIGURE 1.3 – Fonction XOR à réaliser par le perceptron.

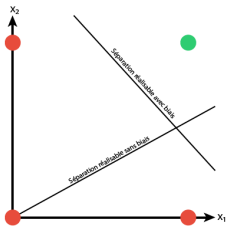

FIGURE 1.4 – Intérêt de l'introduction d'un biais.

$s = 0$ de celui définit par $s = 1$, découle alors directement de l'équation :

$$w_2 * x_2 + w_1 * x_1 + w_0 = 0 \tag{1.6}$$

On constate ici l'intérêt du biais, qui permet de réaliser des séparations affines de l'espace des entrées et pas seulement des fonctions linéaires.

Les limites d'un neurone seul sont alors évidentes : toutes les séparations de l'espace des entrées ne sont pas affine, et le cas du XOR en est déjà une illustration.

Il est alors nécessaire d'introduire la structure de réseau. En prenant une couche d'entrée, une couche cachée et une couche de sortie, respectivement composées de deux, deux et un neurone, on peut contourner le problème sus-cité. Cette structure permet en fait de générer deux séparations de l'espace des entrées.

La séparation que l'on cherche à effectuer est en effet de la forme :

$$f(x_1, x_2) = 0 \Leftrightarrow \begin{cases} x_1 + x_2 - 0,5 & < & 0 \\ x_1 + x_2 - 1,5 & > & 0 \end{cases} \tag{1.7}$$

Ce premier exemple simpliste manifeste donc la nécessité d'adapter la structure du réseau au problème traité.

De manière plus générale, le nombres de neurones du réseau fait varier le nombre de poids du réseau, et donc la capacité du réseau à séparer des ensembles

multiples et complexes. Ainsi, un réseau sous-dimensionné mène à des résultats pas assez précis et un réseau sur-dimensionné mène à du sur-apprentissage. On a donc environ la loi suivante :

$$N < \frac{1}{10} * T * dim(s) \tag{1.8}$$

— N est le nombre de neurones dans le réseau.
— T est le nombre de vecteurs de la base d'apprentissage.
— s est le vecteur de sortie.

De plus, le nombre de neurones dans une couche doit être de moins de trois fois celui de la couche précédente.

Ces deux approximations donnent donc une première idée de la structure à adopter pour un réseau de neurones.

Chapitre 2

Algorithmes d'apprentissage

2.1 L'objectif des algorithmes d'apprentissage

L'objectif des algorithmes d'apprentissage est simple : optimiser les poids du réseau de neurones afin qu'il "apprenne".

Les poids sont en effet la seule variable disponible après avoir définit l'architecture du réseau (il existe des algorithmes permettant de modifier l'architecture durant l'apprentissage mais on ne les étudiera pas ici). On va donc les modifier progressivement, après avoir présenté au réseau un nombre fixé d'exemples, afin que le réseau réagisse au mieux au stimulations qui lui sont présentées.

Il existe en réalité deux types d'algorithmes d'apprentissage.

Le premier cas est celui de l'apprentissage supervisé. Il nécessite de connaître "l'étiquette" des exemples entrés dans le réseau pour effectuer l'apprentissage.

On observe ainsi quelle est la réponse proposée par le réseau après propagation de l'entrée et on la compare à l'étiquette, correspondant à la réponse attendue. On peut alors "récompenser" les comportements positifs en renforçant les connections mises en jeu lors de cette propagation, ou "punir" les mauvais comportements en affaiblissant ces dernières.

On s'appuie alors sur une classification a priori des données : les classes sont crées par le concepteur du problème et du réseau. Cela est à double tranchant. En effet, les classes choisies seront a priori plus pertinente vis-à-vis du résultat attendu mais pas nécessairement vis-à-vis des données et de la structure du réseau.

Le second cas est celui de l'apprentissage non supervisé. Comme l'indique son nom, ce genre d'apprentissage n'a pas besoin d'une intervention extérieure pour apprendre. En pratique cela se manifeste par l'absence d'étiquettes dans les bases d'apprentissage.

Ce genre d'apprentissage revient plus ou moins à mettre en concurrence les différentes classes de données à chaque présentation d'exemple. Ainsi, la classe qui correspond la mieux à l'entrée devient la sortie et la classe est modifiée pour ressembler encore plus à l'entrée qui a permis l'activation.

Au final, dans ce cas, la définition de chaque classe n'est pas explicite : c'est

le réseau qui détecte lui même une structure de classe et il l'amplifie durant l'apprentissage afin d'améliorer la classification qu'il a détecté. Les classes ne sont alors pas forcément celles que l'on attend mais plutôt les classes les plus "naturelles" étant données la base d'apprentissage et l'architecture du réseau. En ce sens les algorithmes d'apprentissage s'approchent de la notion de "clustering".

2.2 La méthode de rétropropagation du gradient

La rétropropagation du gradient correspond à une descente de gradient dans l'espace des poids.

Le principe de l'algorithme est en fait d'assigner à chaque neurone sa responsabilité dans l'erreur commise dans le calcul de la sortie. Les poids individuels des neurones sont ainsi corrigés en fonction de cette responsabilité.

2.3 test

2.3.1 Initialisation des poids

Les valeurs sont initialisées de manière aléatoire, avec des valeurs centrées en 0.

Le choix de l'intervalle possible est très important : un intervalle trop petit va beaucoup ralentir l'apprentissage tandis qu'un intervalle trop grand donne des données trop éparses qui rendent plus difficile l'apprentissage.

Ainsi, l'intervalle choisi est les suivant :

$$\forall i, w_i \in [\frac{-2.4}{N_i}, \frac{-2.4}{N_i}] \tag{2.1}$$

où N_i est le nombre d'entrées du neurone.

2.3.2 Calcul des variations de poids

Les exemples sont présentés consécutivement au perceptron, et à chaque nouvel exemple on calcul les nouvelles variations de poids.

Une fois l'entrée propagée dans le réseau, on obtient une sortie $y^{obtenue}$. Notons par ailleurs la sortie théorique $y^{theorique}$ et h le produit scalaire entre les poids w et les entrées x. Nous pouvons alors calculer l'erreur commise par le réseau, pour chaque neurone de sortie :

$$\forall i, e_i^{sortie} = f'(h_i^{sortie}) * (y_i^{theorique} - y_i^{obtenue}) \tag{2.2}$$

On propage ensuite l'erreur de la couche n vers la couche $n-1$ en assignant à chaque neurone sa responsabilité dans l'erreur commise e^{sortie} :

$$\forall j, e_j^{(n-1)} = f'(h_j^{(n-1)}) * \sum_k w_{i,j} * e_i^{(n)} \tag{2.3}$$

Une fois le calcul de l'erreur retropropagé pour toutes les couches du réseau, il ne reste plus qu'à assigner les différentes variations de poids selon un taux d'apprentissage λ :

$$\Delta w_{i,j}^{(n)} = \lambda e_i^{(n)} x_j^{(n-1)} \tag{2.4}$$

2.3.3 Potentielle modification de l'algorithme

L'une des modifications possibles de cet algorithme est l'ajout d'une inertie à la modification des poids.

La variation de poids devient alors :

$$\Delta w_{i,j}^{(n)}(t) = \lambda e_i^{(n)} x_j^{(n-1)} + \alpha \Delta w_{i,j}^{(n)}(t-1) \tag{2.5}$$

Le coefficient α est le coefficient d'inertie. Il est dans l'intervalle [0;1] et ajoute ainsi une influence des variations de poids précédentes sur les variations de poids actuelles.

Cette technique permet d'éviter certains minima locaux et de se stabiliser dans des configurations plus optimales. Cela peut aussi accélérer la convergence de l'apprentissage si le coefficient α est bien choisit.

Chapitre 3

Influence des paramètres sur l'apprentissage.

3.1 Paramètres perceptron

3.1.1 la méthode de la quantité de mouvement

La méthode de la quantité de mouvement est une méthode utile pour éviter que l'algorithme de backpropagation ne tombe dans un minimal local et n'en sorte pas. Un apprentissage efficace passera le minimum local et doit converger vers les poids idéaux de toutes les synapses pour que le réseau fasse le moins d'erreur possible. Cette méthode est donc parfois nécessaire pour améliorer les performances d'apprentissage du réseau. Cependant, elle ne devra pas être trop élevé afin de ne pas causer de trop grandes oscillations de valeur qui rendront la convergence impossible. L'équation d'update des poids est la suivante :

$$\Delta w_{ij}[n] = \Delta w_{ij}[n] + \alpha \Delta w_{ij}[n-1] \tag{3.1}$$

Ici l'étude de cette méthode n'a pas été étudiée, néanmoins, il reste la possibilité dans notre code de pouvoir modifier la valeur du facteur α qui correspond à la variable `momentumfactor`

3.1.2 le learning rate

Le learning rate est le paramètre qui va le plus déterminer la tendance ou non d'un perceptron d'évoluer en fonction des exemples que l'on va lui mettre en entrée. Plus le learning rate sera grand, plus les poids des synapses vont évoluer de façon rapide. Encore une fois il y a un compromis à faire entre vitesse d'apprentissage et la stabilité d'apprentissage. Dans notre code, nous avons choisi un learning rate de 0,1 qui semble être un bon compromis.

3.1.3 Structure du réseau

Perceptron optimum dans notre cas Une des meilleures structure du réseau que nous avons trouvé pour le perceptron qui satisfait en même temps des conditions de rapidité et de justesse est un perceptron à 3 couches dont le nombre de neurones décroit de façon géométrique au fur et à mesure des couches. Notre structure est donc constituée de 3 couches de 784, 90, 10 neurones respectivement. On peut donc s'apercevoir qu'il y a un facteur 9 entre le nombre de neurones de couches consécutives. A la figure suivante, nous avons la courbe d'une réalisation de ce perceptron pour la base de données MNIST, Nous constatons que nous arrivons à une erreur de learning juste en dessous de 1% et de 2.6% en test. Deux millions d'exemples sont passés pour entrainer ce perceptron et avoir de tels résultats

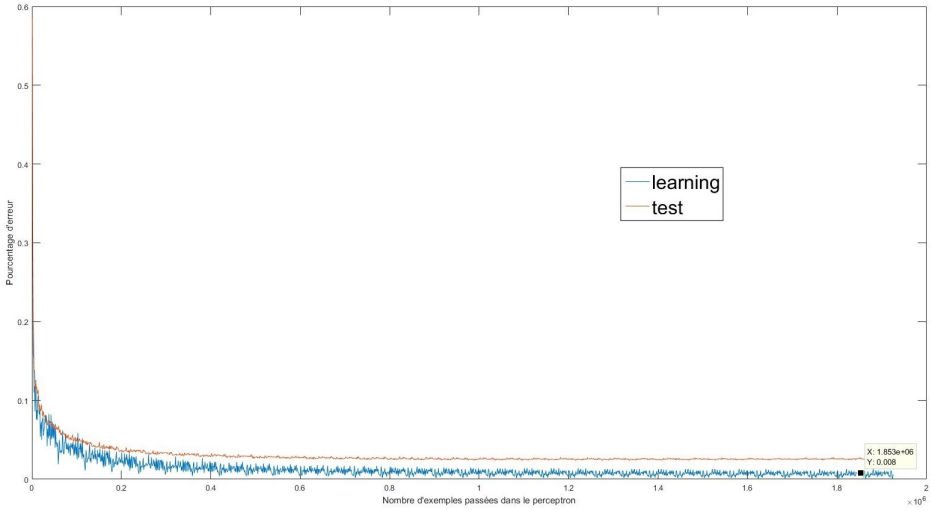

FIGURE 3.1 – Réalisation longue du perceptron 784-90-10 neurones sur MNIST nombre d'exemples vs pourcentage d'erreurs

On peut s'apercevoir que l'erreur de learning continue légèrement à décroitre tandis que l'erreur de test qui elle ne diminue plus et sature très vite à partir de 600.000 exemples. On peut confirmer cette impression par le graphe de l'erreur quadratique de test et de learning où le même phénomène se reproduit plus clairement. (Voir figure 2)

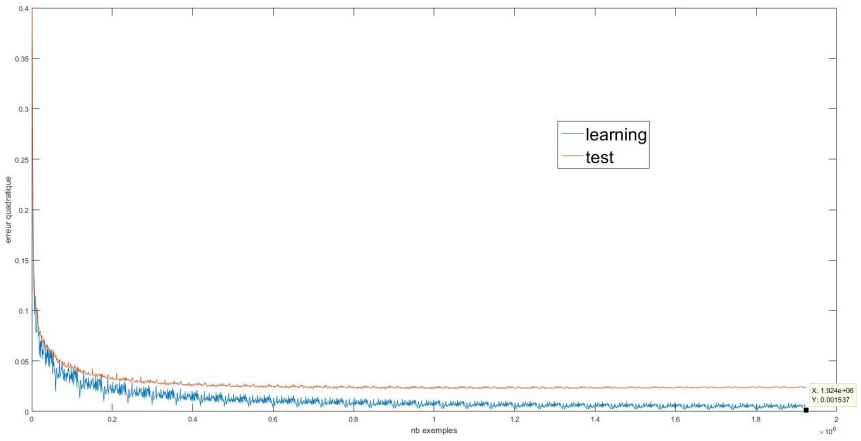

FIGURE 3.2 – Réalisation longue du perceptron 784-90-10 neurones sur MNIST - nombre d'exemples vs erreur quadratique

étude de la variation de la structure Nous avons essayé de faire varier le nombre de couches. Avec 2 couches et donc un perceptron à 784 puis 10 neurones pour la deuxième couche, nous obtenons des résultats suffisamment bon en termes de rapidité de calcul mais ce perceptron reste à 3.4% d'erreur en test soit près de 1% de plus qu'un perceptron à 3 couches (voir figure 3). Nous avons aussi essayer un perceptron à 4 couches mais le temps de calcul était trop élevé et les résultats pas suffisamment intéressant pour continuer à élever le nombre de couches.

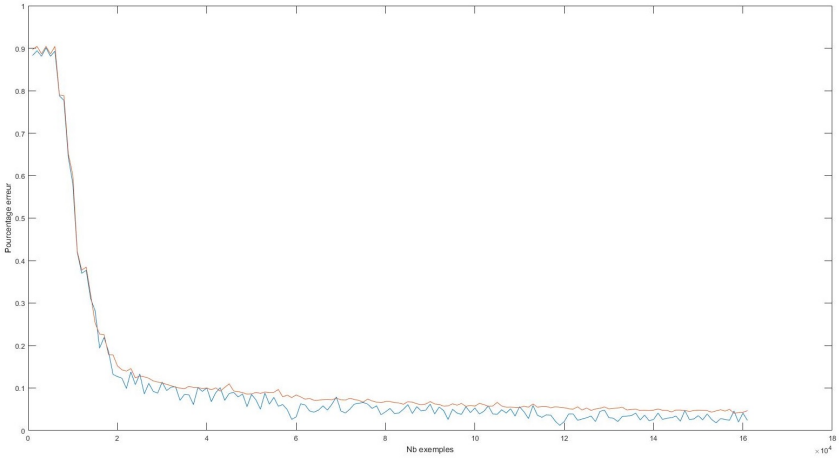

FIGURE 3.3 – Control with both motors(100%-100%)

En revenant au perceptron à 3 couches et vu que le nombre de neurones d'entrée et le nombre de neurones de sorties sont fixés par le problème :

— 784 neurones d'entrées vu que c'est le nombre de pixel d'un image de la librairie MNIST.

— 10 neurones de sorties vu que c'est le nombre de classes.
Nous avons décidé de faire varier le nombre de neurones de la couche intermédiaire. Nous avons tracé le pourcentage d'erreur de learning au fur et à mesure de l'apprentissage (figure 4) pour une couche intermédiaire de 180 neurones (en orange), de 90 neurones (en jaune), de 50 neurones (en bleu). On remarque qu' à la fin plus la courbe à de neurones intermédiaires, meilleur est le taux d'erreur. L'erreur du perceptron avec 50 neurones est le plus élevé même si cela reste très faible. Celui avec 90 neurones intermédiaires, malgré une initialisation peu favorable arrive à avoir un taux d'erreur entre les 2. Finalement le perceptron avec le plus de neurones intermédiaires à le meilleur taux d'erreur mais pour un temps de calcul accrus. On gardera donc celui avec 90 neurones intermédiaires au vu du fait qu'avec une initialisation plus favorable fait jeu égal avec celui à 180 neurones pour un temps de calcul plus faible.

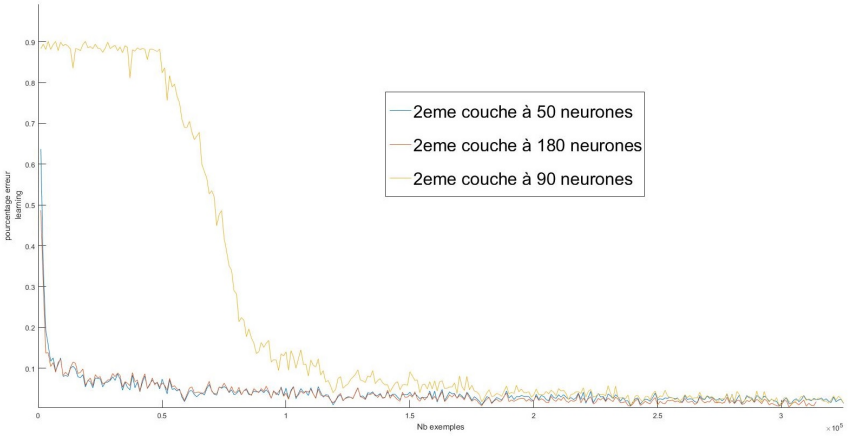

FIGURE 3.4 – Pourcentage d'erreurs de learning pour 3 couches intermédiaires différentes.

Il est donc à noter que plus on augmente le nombre de neurones par rapport à la loi géométrique déterminée précédemment, plus on augmente le temps de calcul sans améliorer nécessairement les résultats. Tandis que si on diminue le nombre de neurones intermédiaires en dessous de 50, plus on dégrade nos résultats, ce qui est forcément peu à notre avantage.

On peut tout de fois se demander pourquoi ne pas garder la couche intermédiaire à 50 neurones puisque les résultats sont plus au moins similaires que celui à 90 neurones. Pour répondre à cette question on peut regarder la figure 5 qui nous montre que au fur et à mesure de l'apprentissage l'erreur de test à tendance à remonter légèrement. Cela nous est confirmé à la figure 6, car nous voyons l'erreur quadratique de test augmenter plus visiblement alors que celle de learning reste constante. On préférera donc la structure que nous avons choisie pour son meilleur comportement lors de l'apprentissage.

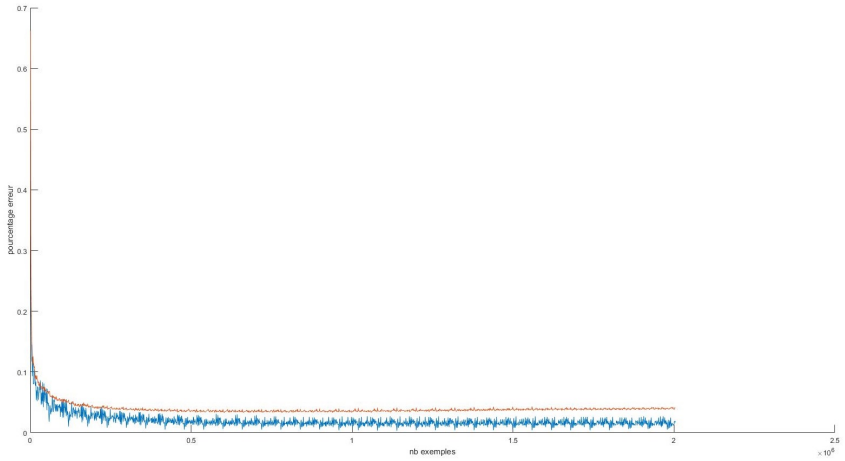

FIGURE 3.5 – pourcentage d'erreur de learning(bleu) et de test(orange) lors de l'apprentissage

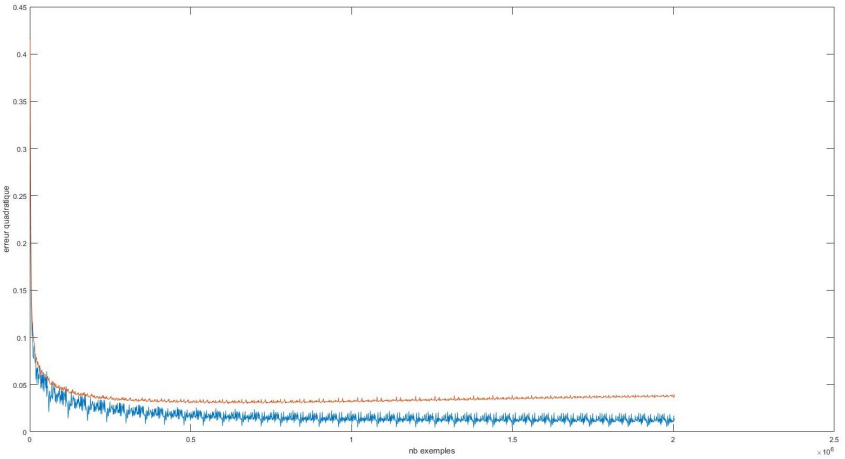

FIGURE 3.6 – erreur quadratique de learning(bleu) et de test(orange) lors de l'apprentissage

étude supplémentaire possible Une étude statistique de l'apprentissage aurait pu être intéressante, malheureusement le temps nous a manqué pour la réaliser. A la figure 7, on peut apercevoir une courbe moyennés sur 4 trajectoires de l'apprentissage de notre perceptron. A partie d'un plus grand nombre de réalisation, nous aurions pu calculer les écarts types, les intervalles de confiances, les moyennes des taux d'erreurs,... Nous constatons sur cette même figure que les courbes ont déjà tendance à se lisser.

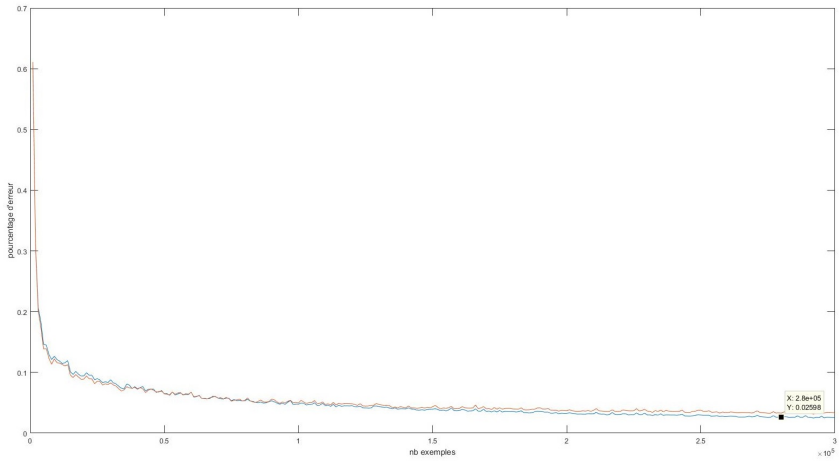

FIGURE 3.7 – Courbes du pourcentage d'erreurs de test et de learning moyennées sur 4 exécutions

Troisième partie

Approfondir le sujet : les machines de Boltzmann.

Chapitre 4

Introduction

Les réseaux de Boltzmann sont des réseaux qui ont un fonctionnement particulier que nous allons présenter dans ce chapitre. Les machines de Boltzmann constituent des réseaux neuronaux. La principale différence entre les réseaux neuronaux comme les réseaux neuronaux de convolution (CNN) et les réseaux de Boltzmann va être le fait qu'un réseau de Boltzmann n'est pas *feed-forward* ; comme l'était le perceptron, précédemment présenté dans ce rapport.

Ce rapport traitera d'abord des machines de Boltzmann restreintes en les présentant et en expliquant les lois qui régissent leur fonctionnement pour ensuite s'intéresser aux Réseaux de Boltzmann Profond.

Chapitre 5

Les machines de Boltzmann restreintes

Les machines de Boltzmann restreintes, ou RBM, ont d'abord été pensées par P. Smolensky (1986) mais réellement mise en œuvre et étudiées par G. Hinton (2005) Nous allons présenter dans cette partie leur constitution et leurs fonctionnement.

5.1 Présentation et éléments de preuve

Les RBM sont des réseaux neuronaux qui donnent une probabilité d'activation de chaque neurone.

Une RBM possède deux couches de neurones : l'une sera qualifiée de couche visible et l'autre de couche cachée. La couche visible correspond à l'entrée de notre réseau, ce sont les neurones de la couche d'entrée. On va leur assigner les valeurs des exemples de notre jeu de données pour entrainer le réseau.

Au sein d'une même couche, aucun neurone n'est relié à un autre, c'est ce caractère restreint qui donne sa particularité et son nom à ce type de réseau de Boltzmann. Cependant, chaque neurone de la couche visible (respectivement cachée) est connecté à tous les neurones de la couche cachée (respectivement visible).

On forme alors un graphe, non orienté, qui est un modèle de la distribution de probabilité. Chaque neurone est une variable aléatoire qui prend sa valeur dans $0; 1$. Le fait qu'aucun neurone v ne soit connecté aux autres de la couche traduit le fait que les variables aléatoires associées sont indépendantes.

La probabilité qu'un neurone v_i de la couche visible soit activé est

$$p(v_i = 1 \mid H) = \sigma(\sum_{j=1}^{m} w_{i,j} * h_j + b_i) \tag{5.1}$$

En notant w_i le poids de la synapse reliant les neurones v_i à h_j et b_i le biais du neurones v_i.

Puisque la probabilité d'activation du neurone v_i d'une couche dépend uniquement de la valeur des états des neurones de l'autre couche et uniquement au moment où l'on souhaite échantillonner, l'ensemble des probabilité d'activation des neurones aux instants t forme un champ de Markov en prenant pour variables aléatoires la valeur de l'état de chaque neurones.

On introduit maintenant l'échantillonnage de Gibbs, qui est un algorithme de la classe des Metropolis-Hasting. Cet algorithme est un Monte-Carlo qui permet d'échantillonner un probabilité jointe dont l'idée principale est d'échantillonner les états des variables en se basant sur la valeur des états des autres variables grâce à la probabilité conditionnelle qui les relie. C'est une marche aléatoire sur les chaînes de Markov. Les neurones d'une même couche sont indépendants on peut donc mettre à jour une couche entière d'un coup ce qui rend l'échantillonnage plus rapide et plus simple à réaliser.

Par ailleurs on peut démontrer que tout les neurones peuvent prendre les valeur 0 ou 1 en un seul échantillonnage et que toutes les combinaisons d'état des neurones de la RBM sont atteignables en un nombre fini d'échantillonnage ce qui montre que la chaîne est irréductible. Par ailleurs la la chaîne de Markov est apériodique. Ces deux propriété d'apériodicité et de irréductibilité permettent de prouver que le RBM possède une distribution stationnaire vers laquelle elle converge.
Une preuve plus rigoureuse à été proposé par Fischer et Igel (*An Introduction to Restricted Boltzmann Machines* (2012)).

La particularité de ce réseau est donc le fait qu'il n'est pas *feed-forward*, même si l'on considère la couche cachée comme entrée du réseau, le réseau arrive à une distribution stationnaire qu'après des allers-retours (c'est-à-dire des échantillonnages) entre les deux couches.

5.2 Loi d'apprentissage

Pour faire l'apprentissage, on va entraîner le réseau sur les différents exemple, la première différence avec l'implémentation du perceptron est que l'on va échantillonner les états des neurones à chaque fois, on travaille avec des données binaires et non réelles, nous avons décidé d'affecter, dans le cas de la base de données MNIST, la valeur 1 si le pixel n'était pas blanc. On va ensuite faire des pas de Gibbs pour obtenir la distribution stationnaire de probabilité qui correspond à l'exemple et aux paramètres (poids et biais) fixés.

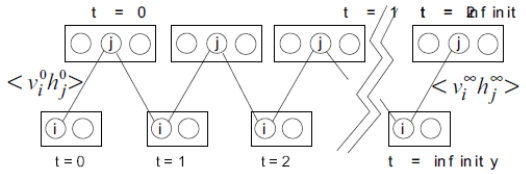

FIGURE 5.1 – Pas de Gibbs.

Une fois la distribution stationnaire atteintes on obtient deux ensembles de valeurs : l'état des couches V et H qui correspond à l'*input*, que l'on peut appeler $< v_i h_j >_{data}$ ou $< v_i^0 h_j^0 >$, et l'état des couches V et H une fois la distribution stationnaire atteinte notée $< v_i h_j >_{model}$ ou $< v_i^\infty h_j^\infty >$. A partir de ces deux distribution, on peut modifier les poids et les biais du réseau selon la loi :

$$\Delta w_{i,j} = \alpha(< v_i h_j >_{data} - < v_i h_j >_{model}) \tag{5.2}$$

où α est le *learning rate*

Un point critique de l'implémentation d'une RBM est comme bien souvent la loi d'apprentissage. Si l'on trouve dans la littérature plusieurs solutions, la forme générale du calcul est celle présentée ci-dessus.

Dans ce calcul, on distingue l'échantillonnage de la machine à partir des données (data), de l'échantillonnage de la machine après la reconstruction par pas de Gibbs, $< v_i h_j >_{model}$.

Pour extraire des informations à partir de l'*input*, il est important de ne pas reconstruire les deux couches depuis les *inputs*. Ainsi le calcul du produit $< v_i h_j >_{data}$ s'obtient par l'état de l'entité visible v_i et l'évaluation de l'entité cachée sous forme réelle : $p(h_j|v^{(0)})$.

Une chaîne de Markov peut prendre du temps à converger et dans le cas des RBM qui peuvent avoir plusieurs centaines de neurones par couche, les pas de Gibbs, c'est-à-dire l'échantillonnage des états des neurones au sein d'une même couche, peut être très calculatoire.

Il est alors proposé par Hinton en 2005 une méthode pour approximer la distribution stationnaire : au lieu de faire un nombre fini de pas de Gibbs jusqu'à avoir cette stationnarité, on se limite qu'à peu de pas de Gibbs. Cette méthode s'est révélée efficace lors de l'apprentissage des RBM. On se limite à quelques pas de Gibbs (de l'ordre de l'unité). Hinton propose d'ailleurs de ne faire qu'un seul pas de Gibbs.

Chapitre 6

Les Deep Belief Networks

6.1 Principe des Deep Boltzmann Machines

Nous avons présenté dans le chapitre précédent les *restricted boltzmann machines* avec leur utilisation parallélisée. Nous verrons dans ce chapitre une nouvelle façon d'utiliser ses machines, en série. Après la description d'une configuration intéressante pour l'apprentissage, nous présenterons nos résultats obtenus sur la base MNIST.

En effet, le concept de *Deep Boltzmann Machine*(DBM) repose sur l'utilisation en cascade des RBM(figure 6.1). La couche cachée de la première machine correspond à la couche visible de la suivante, et ainsi de suite. Nous avons vu qu'une RBM permet l'extraction des informations pertinentes de l'*input*, ainsi l'intérêt majeur de cette structure est l'extraction successive des *features* de l'*input*, afin d'obtenir une identification de plus en plus fine.

6.2 Spécificité de l'apprentissage

L'apprentissage d'une DBM se fait en deux étapes : d'abord non supervisé puis supervisé. Contrairement au perceptron présenté précédemment, nous ne disposons pas pour une *Deep Boltzmann Machine* d'une sortie nous donnant le label attendu. Il est donc nécessaire d'ajouter au bout des machines de Boltzmann en cascade une ou plusieurs *layers* en *feed-forward* menant vers une *output layer* de 10 neurones.

L'apprentissage non-supervisé est destiné à entraîner les machines de Boltzmann. Pour se faire, il s'agit de faire l'apprentissage sur la première machine isolément, puis construire la couche cachée de la machine avec un pas de Gibbs depuis l'input. Ainsi, on obtient l'input nécessaire à l'apprentissage pour la machine suivante, et par itérations successives, il est possible de faire l'apprentissage sur toutes les machines.

L'apprentissage supervisé fait office de *fine-tuning* : il entraîne le perceptron situé à la fin de la chaîne, après les machines de Boltzmann. Il s'opère par rétro-propagation, dont nous avons décrit plus haut le fonctionnement.

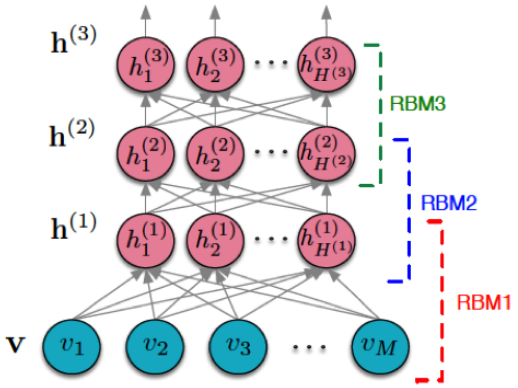

FIGURE 6.1 – Mise en cascade des RBM.

6.3 Résultats

Le réseau que nous avons entraîné sur la base MNIST avait une configuration comportant 5 layers, dont l'output layer. Les entités sont réparties en 784, 500, 500, 2000 et enfin 10 entités, soit 3 RBM et 1 perceptron à deux couches.

Les paramètres choisis sont les suivants :

Learning rate :	0.1
Gibbs steps :	2
Number of sets of training :	5 (soit 300.000 exemples)

Tous les poids et les biais du réseau ont été initialisés à zéro, à l'exception des biais de l'*input layer*, initialisés à $log[p_i/(1-p_i]$, avec p_i la proportion d'*inputs* pour lesquels l'entité visible v_i est activée.

Nous avons ensuite observé les résultats de l'apprentissage au fur-et-à-mesure de l'apprentissage et sur une partie d'exemples de test. Les résultats figurent sur la figure 6.2.

On obtient un pourcentage d'erreur autour de 11%, avec une décroissance rapide initialement, puis qui tend à diminuer après quelques époques. L'allure est conforme au résultat attendu, même si on l'on aurait pu souhaiter une meilleure convergence.

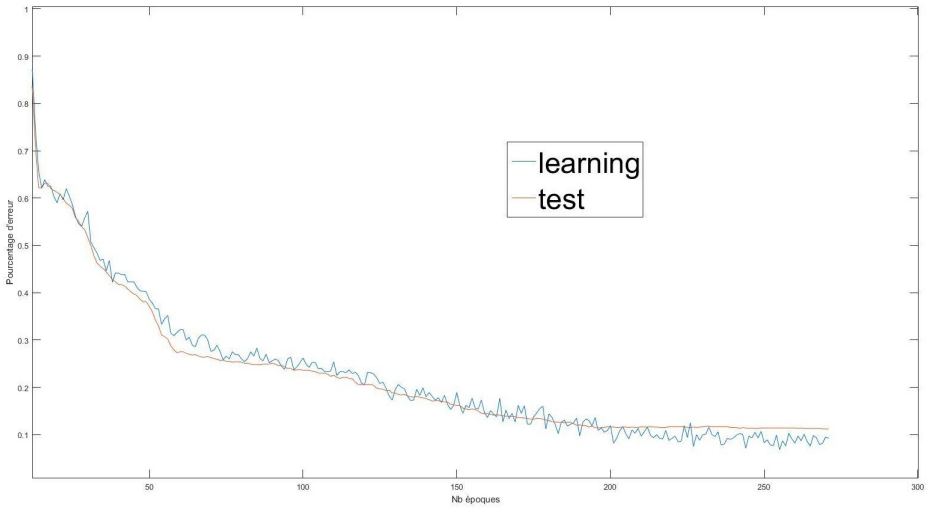

FIGURE 6.2 – Courbe d'apprentissage de la Deep Boltzmann Machine.

Chapitre 7

Aller plus loin

7.1 La visualisation des spectres du réseau

7.1.1 Cas de la RBM

Lors du développement de la RBM, nous avons étés confrontés au problème de la quantification de l'apprentissage. Comment savoir si oui ou non le réseau s'adapte aux exemples qu'on lui a présenté ?

Suite à l'absence de réelle mesure de cette notion, l'idée nous est venue d'implémenter une mesure bien plus qualitative du phénomène : l'objectif était de visualiser l'importance que le réseau accorde à chaque pixel de l'image.

Dans le cas de la RBM, cela correspond tout simplement à l'ensemble des pondérations des connections entre la couche visible et la couche cachée du réseau. Ainsi, à chaque neurone de la couche cachée, on peut associer une image correspondant aux dites pondérations. Ce sont ces images que l'ont qualifie de "spectres".

Ces spectres représentent les caractéristiques que le réseau essaie de détecter dans une image pour y reconnaître un caractère. Selon le nombre de neurones dans la couche cachée, ces caractéristiques sont assez différentes. En effet, plus il y a de neurones cachés, plus le réseau pourra détecter de nombreuses caractéristiques précises, qui une fois assemblées formeront la représentation du caractère complet. Ces caractéristiques peuvent être, par exemple, des angles ou des boucles. A contrario, quand le réseau a peu de neurones cachés, ces spectres sont beaucoup plus proches de la forme "usuelle" du caractère associé. Ainsi, avec un seul neurone caché, on obtient très exactement la représentation d'un chiffre idéal pour ce réseau.

L'intérêt d'une telle initiative est simple : utiliser l'intuition visuelle qu'on a des chiffres pour juger de la qualité de l'apprentissage.

Au début de l'apprentissage, l'image sera exclusivement composée de bruit blanc (c'est une conséquence de l'algorithme d'apprentissage), tandis qu'à la fin elle ressemblera au négatif d'un des exemples de la base de donnée (toujours dans le cas d'un réseau à un seul neurone caché).

Pour faire cette visualisation, on cherche uniquement à connaître les neurones

FIGURE 7.1 – Spectre d'une RBM entraînée à vingt neurones cachés.

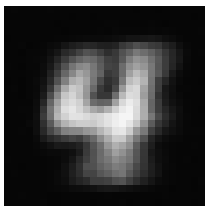

FIGURE 7.2 – Spectre d'une RBM ayant apprise la "forme" 4.

qui activent le neurone de la couche caché. Une méthode de visualisation serait donc de chercher les valeurs binaires des neurones de la couche visible pour activer le neurone désiré. Puisque le réseau n'est pas orienté, on peut directement affecter la valeur 1 au neurone h_j de la couche caché qui correspond au filtre que l'on veut voir et 0 aux autres. On fait alors un pas de Gibbs vers la couche visible. Pour obtenir un meilleur résultat, on peut directement travailler avec la valeur de la sigmoïde des neurones de la couche visible.

7.1.2 Cas du DBN

Dans le cas du DBN, il est plus compliqué d'obtenir une image du fait de l'architecture profonde. On va donc choisir le neurone de la couche dont on veut observer le filtre ou ce qu'on pourrait appeler la carte d'activation. Pour cela on assigne la valeur 0 à tout les neurones du réseau et on va faire un unique pas de Gibbs jusqu'à la couche d'entrée du réseau en échantillonnant la valeur des neurones des couches par lesquelles on passe.

Cette méthode a donc un inconvénient : on a une chance de perdre de l'information. Nous avons alors testé deux options : la première est de changer la nature du réseau et de travailler avec les probabilités comme valeur de l'état du réseau. Cette méthode n'a pas été concluante. Nous sommes alors venus à considérer une autre méthode, dans l'esprit du réseau.

Cette méthode est la suivante : on initialise les valeurs des neurones du réseau entraîné à 0 sauf pour le filtre que l'on souhaite visualiser qui est à l'état 1. On va ensuite effectuer plusieurs pas de Gibbs au sein de la **même** RBM. Ainsi on va obtenir une distribution des états qui correspond à l'activation du filtre désiré mais aussi d'autres filtres. Ce procédé permet de réduire le nombre de neurones de la couche précédente qui ne serait pas dan l'état voulu. On procède ainsi jusqu'à la première RBM où l'on utilise la méthode exposée dans la partie correspondante.

Cette méthode est simplement expérimentale mais produit des résultats plus probant. Dans la littérature les méthodes de visualisation sont peu détaillées mais semble être le résultat d'un apprentissage à partir d'une entrée quelconque afin d'avoir le neurone dont on veut observer la carte d'activation activé. C'est à dire que l'on ferait un apprentissage supervisé mais on ne modifierait que la valeur d'entrée du réseau.

7.2 Discrimination grâce aux RBM

La discrimination est un principe qui consiste à construire autant de Restricted Bolztmann machine que de classes à différencier. Par exemple, pour la base de données MNIST, on fonctionne avec 10 RBM vu qu'il y a 10 chiffres à reconnaitre.

Après l'initialisation des 10 RBM, on entraine les machines indépendamment les unes des autres. c'est à dire que chaque RBM ne sera entrainé que par les

exemples d'une seule classe. Chaque machine se voit donc une classe avec les exemples correspondant à cette classe.

Maintenant que les machines sont entrainées, il reste à savoir comment on détermine à quelle classe appartient un exemple. Pour cela, on passe l'exemple dans chaque machine sans l'entrainer. On considérera dés lors que l'exemple appartient à la classe associée à la machine qui aura l'énergie libre la plus basse. Plus la différence entre l'énergie libre minimum et les autres est grande, plus on sera certain de cette prédiction.

7.2.1 Résultats

Après une campagne de calcul lancé sur ordinateur, nous avons obtenu une courbe de l'erreur de learning en fonction du learning rate. On a exécuté 20 fois le programme pour un learning rate donné. On obtient la courbe à la figure 8.

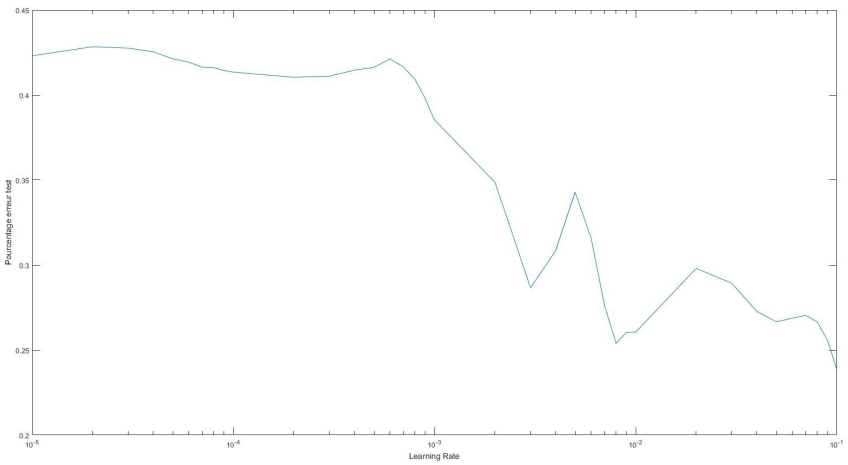

FIGURE 7.3 – erreur de learning en fonction du learning rate

On aperçoit donc un plateau aux alentours des 42% au début de la courbe. On remarque que le learning rate a un effet plus important par la suite. Cependant on obtient des comportements étranges avec des minimums à différent endroits (en 8 10^{-3} et en 0.1). On tourne alors aux alentours des 25% d'erreur de learning. On peut donc dire que cette méthode est moins avantageuse que celle du perceptron.

7.3 Reconnaissance d'image

Dans le cadre de la présentation des projets longs, dont fait partie ce projet, aux élèves de première année, nous avons proposé une application ludique de notre code : la reconnaissance d'image. En partant de notre réseau déjà entraîné, on place en entrée une image de même type que les exemples qui ont servi à entrainer le DBN : des images de 28×28 pixel en niveaux de gris. Ces images sont tracées à l'aide de l'application Paint avec un pinceau de taille 1. En utilisant une dalle tactile, on peut obtenir des exemples de chiffres proches de ceux qu'aurait tracé la même personne avec un stylo.
Proposer de nouveaux exemples au réseau est une manière de le tester et de prendre connaissance des limites de l'entraînement : si la base de données est peu importantes, les résultats sur des exemples qui n'ont pas été traités comme ceux des ensembles d'apprentissage ou de test vont être peu probant.

Les résultats n'ont pas été très concluant, le réseau a montré des difficultés à reconnaître des chiffres pourtant pu ambigus. En observant les erreurs, on a pu s'apercevoir que certaines de ces erreurs sont dues à la base de données. Parmi ces erreurs on a mettre en avant le fait que des pays différents peuvent avoir des graphies bien distinctes sur certains chiffres comme les 1 ou les 7. Cet aspect est aussi développé dans l'annexe A.

Une autre limitation pour cette application est aussi apparue : les exemples tracés à la main étaient parfois trop différents de ceux des ensembles de la base de données mais cette fois-ci en terme de position et de taille. Les fichiers de la base MNIST étant donné en byte, nous n'avons pas cherché à les afficher pour prendre connaissance des exemples.
Ainsi lorsque l'on traçait un chiffre on pouvait être confronté au fait que le réseau ne le reconnaîtrai pas à cause de sa position : si le chiffre n'est pas centré, la RBM n'étant pas invariable par translation des pixels ne peut reconnaître l'image. De même si le chiffre ne prend pas une place suffisante, ou au contraire est top imposant, le réseau ne parvient pas à le reconnaître.

Bibliographie

[1] Hopfield Hopfield. Neural Networks and Physical System with Emergent Collective Computational Abilities. 1982.

[2] Paul Smolensky. Information Processing in Dynamical Systems : Foundations of Harmony Theory. 1986.

[3] Rumelhart, Hinton, and Williams. Learning Internal Representations by Error Propoagation. 1986.

[4] Radford Neal. Probalistic Inference Using Markov Chain Monte Carlo Methods. 1992.

[5] Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus-Robert Müller. Efficient BackProp. 1998.

[6] Gardner and Dorling. Artificial Neural Network (The Multilayer Perceptron) - A Review Of Applications In The Atmospheric Sciences. 1998.

[7] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. 2005.

[8] Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the dimmensionality of data with Neural Networks. 2006.

[9] Geoffrey Hinton. NIPS Tutorial on : DBN, 2007.

[10] Yoshua Bengio and Le Roux Le Roux. Representational Power of Restricted Boltzmann Machines and Deep Belief Networks. 2007.

[11] Florin Leon and Violeta Sandu. Recognition of Handwritten Digits Using Multilayer Perceptrons. *Universitatea Tehnică „Gheorghe Asachi" din Iaşi*, 55(59), 2009.

[12] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7) :579–588, 2009.

[13] Yoshua Bengio. Learning Deep Architectures for AI. 2009.

[14] Abdel-Rahman Mohamed, Geoffrey Hinton, and George Dahl. Deep Belief Network for phone recognition. 2009.

[15] Ruslan Salakhutdinov and Geoffrey Hinton. Deep Boltzmann Machine. 2009.

[16] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Vidualizing Higher-Layer Feature of a Deep Network. 2009.

[17] Geoffrey Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1) :926, 2010.

[18] Ruslan Salakhutdinov. Learning Deep Generative Models, 2010.

[19] George Dahl and Kit La Touche. Learning words with Deep Belief Networks. 2010.

[20] Aaron Courville, James Bergstra, and Yoshua Bengio. A Spike and Slab Restricted Boltzmann Machine. 2011.

[21] Asja Fischer and Christian Igel. An Introduction to Restricted Boltzmann Machines. 2012.

[22] Asja Fischer and Christian Igel. Training Restricted Boltzmann Machines : An Introduction. 2014.

[23] Leon Gatys, Alexander Ecker, and Mathias Bethge. A Neural Algorithm of Artistic Style. 2015.

Annexe A

La base MNIST

A.1 Présentation de la base de donées

La base de données MNIST, Mixed Mixed National Institute of Standards and Technology, est un ensemble de chiffres manuscrits extraits d'une base antérieure, NIST. Lea base est organisée de la manière suivante :

— La base d'apprentissage est constituée de 60000 images.
— La base de test est constituée de 10000 images.
— Chaque image est constituée de 28*28 pixels en nuances de gris.
— Chaque image est normalisée.

Cette base est aujourd'hui devenue une référence permettant de tester les algorithmes d'apprentissages ou de reconnaissance de forme divers et variés. Cet aspect facilite ainsi la comparaison de nos résultats avec ceux d'autres personnes ayant fait la même démarche que nous.
On peut trouver cette base à l'adresse : `http://yann.lecun.com/exdb/mnist/`

FIGURE A.1 – Exemples de chiffres de la base de donnée MNIST.

A.2 Limites de cette base

Cette base, malgré tous ses avantages, n'est cependant pas parfaite.

En effet, une première limite apparaît assez rapidement quand on travaille avec MNIST : cette base n'est pas "universelle". Les caractères utilisés dans la base de donnée sont fortement connotés anglosaxons. On peut constater cet aspect dans les spectres réalisés par nos RBM.

L'image idéale de chaque chiffre a toujours une prédominante américaine, et quand il s'agît ensuite d'entrer des données européenne, la limitation de la base se révèle.

(a) "1" en français

(b) "1" anglosaxon

FIGURE A.2 – Exemple d'une situation de litige entre notation anglosaxone et française.

Une autre limite apparaît aussi : c'est celle de la simplicité de la base de donnée.

La reconnaissance de caractères est en effet une tâche simpliste en comparaison avec certains des problèmes auxquels peuvent être confrontés les réseaux issus du deep learning. Cela se constate par les résultats très bons obtenus pour des réseaux peu complexes, tel que notre perceptron qui atteignait les 3% d'erreur sur MNIST.

Il devient alors nécessaire de trouver un test plus discriminatoire de la qualité des réseaux.

Faute de temps nous n'avons pas eu la possibilité d'approfondir ce point de vue mais semble nécessaire pour tester les réseaux dans un domaine plus complexe.

Annexe B

Présentation du code.

B.1 Code du perceptron

La classe Test La classe Test.java réalise la courbe d'apprentissage sur MNIST d'un perceptron dont la structure en terme de neurones a été construite grâce à un tableau de int

la classe Perceptron Son constructeur est le suivant `public Perceptron(int[] inputData, boolean randomWeight)`. Le tableau de int spécifie le nombre de neurones de chaque couche, le boolean quant à lui veut savoir si les poids des synapses sont initialisés aléatoirement sinon ils auront tous une valeur par défaut. Il est à noté que un perceptron ne peut par marcher si toutes les synapses ont un même poids. Il vous sera donc nécessaire d'attribuer vous-même le poids des synapses avec la fonction `public void setWeight(double w)` de la classe Synapse.java

Pour faire passer un exemple : Il suffit d'utiliser la fonction `public void setNormalizedInputs (double[] x, double max)` pour mettre les entrées normalisées à l'entrée du perceptron puis d'utiliser la fonction `fire` pour calculer les sorties liées à l'exemple.

Pour l'apprentissage : Il suffit de lancer la fonction `public void launch(NeuralNetwork N, double learningRate, Input I)` de la classe BackPropagation sur le perceptron N, pour un learningRate donné, l'apprentissage se faisant sur l'input I.

Pour déterminer la classe à laquelle appartient votre input : Vous devez utiliser la fonction `public double[] getOutputs()` de Perceptron.java et déterminer par la méthode du maximum de vraisemblance le résultat.

B.2 Code de la RBM et du DBN

La classe Entity : C'est la classe qui définit le neurone, qu'il soit d'entrée ou de sortie, ses attributs sont son état, l'erreur correspondant au neurone, son biais et son id.

La classe Sigmoid : Singleton de la fonction sigmoïde.

La classe Restricted Boltzmann Machine : Son constructeur est le suivant `public RestrictedBoltzmannMachine(Entity[] visibleEntities, Entity[] hiddenEntities, double weightWide, double learningRate)`. Elle fait correspondre à un tableau de neurone un tableau de synapse. L'attribut `weightWide` correspond à la valeur maximum que peut prendre un poids, il définit l'intervalle dans lequel les poids prennent leur valeurs à l'initialisation par symétrie avec 0.

Pour faire l'apprentissage d'une RBM : On utilise la methode `public double[][] unsupervisedLearning( int pasDeGibbs, int[] exemple)` puis on applique les fonctions `public double getLogProbabilityDerivativeSum(double[][] RBM)` et `public void applyLearningGradients()`.

La Classe Deep Belief Network : Son constructeur est `public DeepBeliefNetwork(int[] inputData, int rbmLayerNumber, double weightWide, double biasWide, double learningRate, double backPropLearningRate)`, le tableau `inputData` correspond au nombre d'entités par *layer*.

Pour faire l'apprentissage d'un DBN : On utilise la méthode `public void singleUnsupervisedLearning(int pasDeGibbs, int[] exemple, boolean instantLearning)` . Qui va faire l'apprentissage de chaque RBM qui compose le DBN en commençant par l'entrée et en remontant dans le réseau.
Le booléen `instantLearning` permet de faire un apprentissage par batch.