Rapport projet long
Équipe HAL9000

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

11 juin 2016

Table des matières

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 thême 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 éléves 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, notament 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 interessés 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

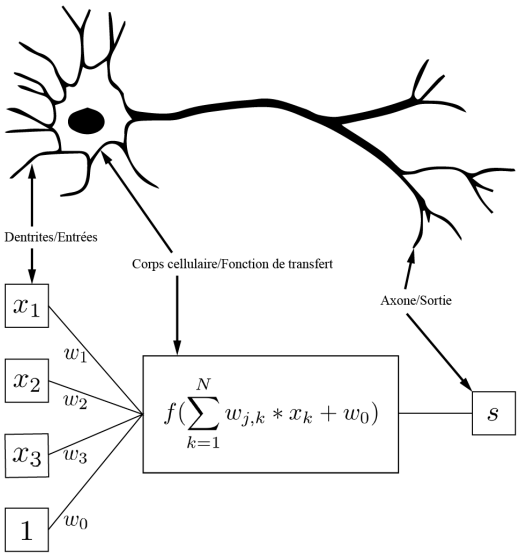

FIGURE 1.1 – Analogie entre neurone biologique et neurone formel.

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 signifiactives 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. Parmis 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.

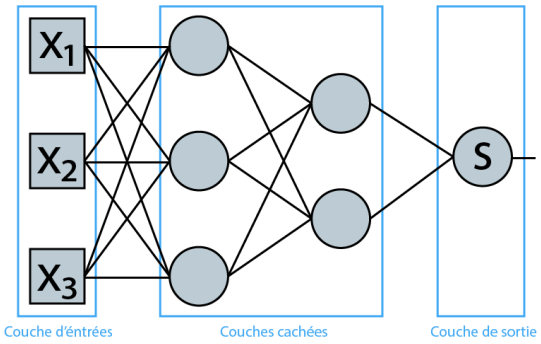

FIGURE 1.2 – Architecture d'un perceptron multi-couches.

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

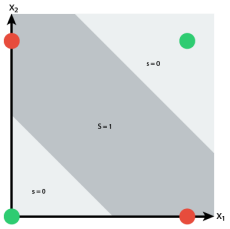

FIGURE 1.3 – Fonction XOR à réaliser par le perceptron.

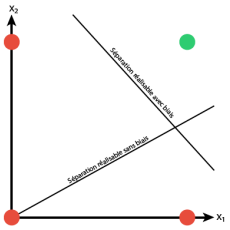

FIGURE 1.4 – Intérêt de l'introduction d'un biais.

$s = 0$ de celui définit par $s = 1$, découle alors directement de l'équation 1.1 :

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

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 prennant 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'appentissage.

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 initailisées au départ 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'enrée 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 vrariations 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 accélerer 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 pourcent et de 2.6 pourcents en test. Deux millions d'exemples sont passés pour entrainer ce perceptron et avoir de tels résultats

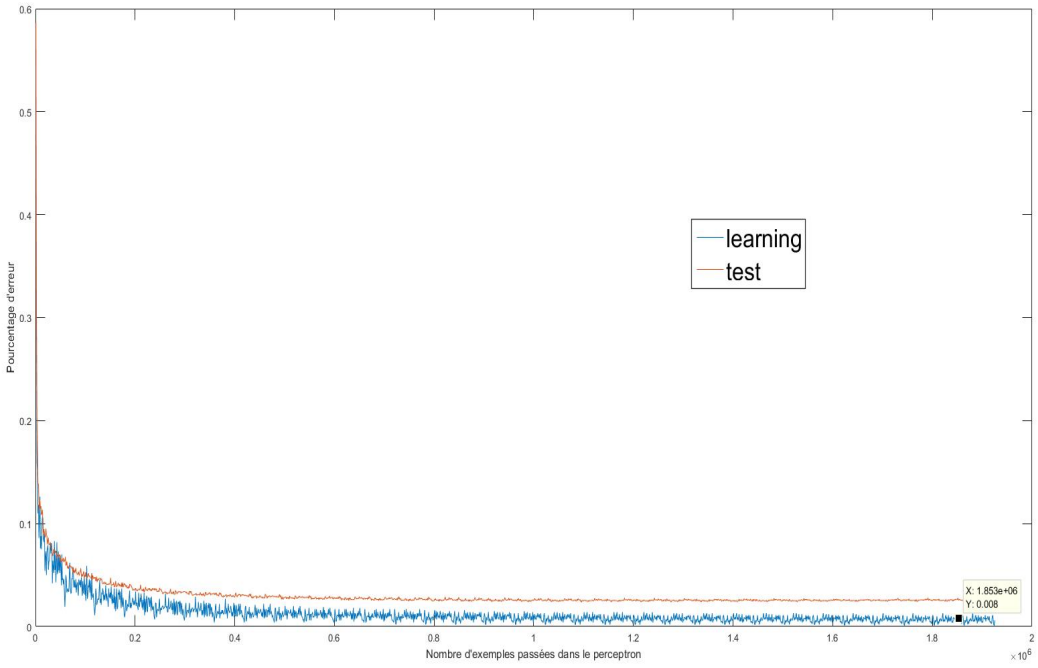

FIGURE 3.1 – Réalisation longue du perceptron 784-90-10 neurones sur Mnist nombre d'exemples vs pourcentage d'erreurs

On peut s'apercevoir que l'erreur de learning continue légèrement à décroite tandis que l'erreur de test qui elle ne dimunie 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)

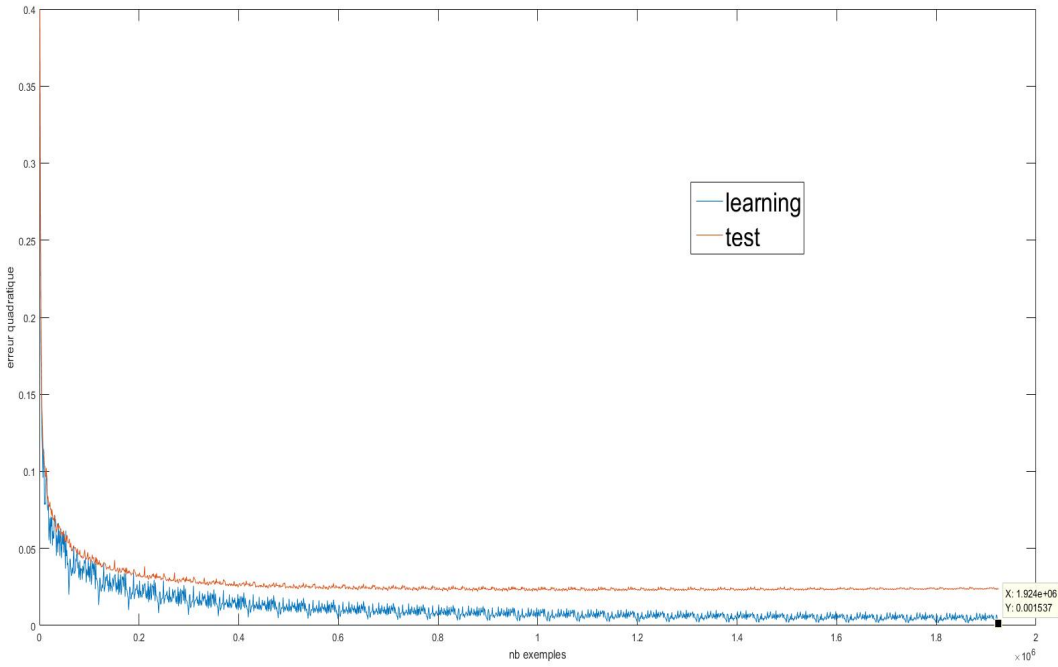

FIGURE 3.2 – Réalisation longue du perceptron 784-90-10 neurones sur Mnist-nombre d'exemples vs erreur quadratique

étude de la vaiation 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 suffisament bon en termes de rapidité de calcul mais ce perceptron reste à 3.4 pourcents d'errurs en test soit près de 1 pourcent 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 suffisament intéressant pour continuer à élever le nombre de couches.

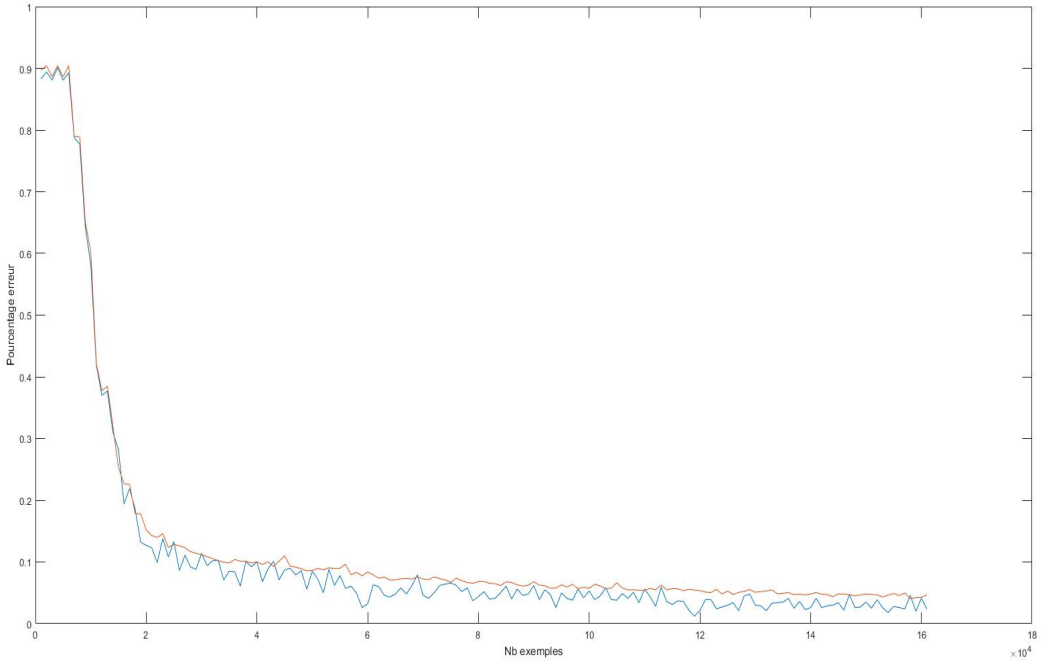

FIGURE 3.3 – Control with both motors(100%-100%)

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 intialisation plus favorable fait jeu égal avec celui à 180 neurones pour un temps de calcul plus faible.

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

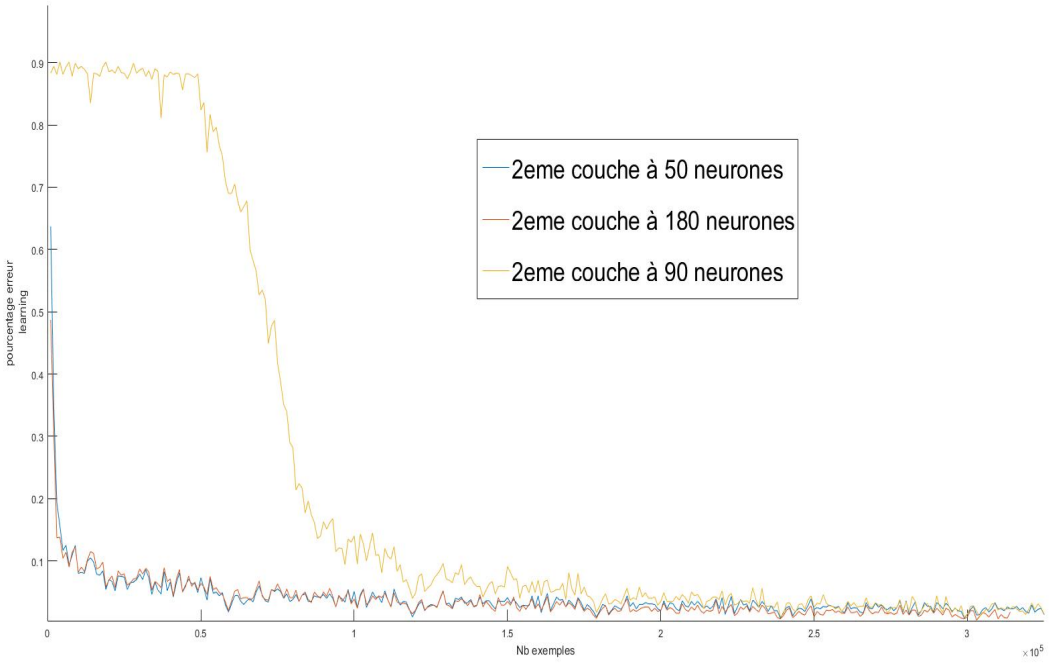

FIGURE 3.4 – Pourcentage d'erreurs de learning pour 3 couches intermédiaires différentes.

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 simulaires 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.

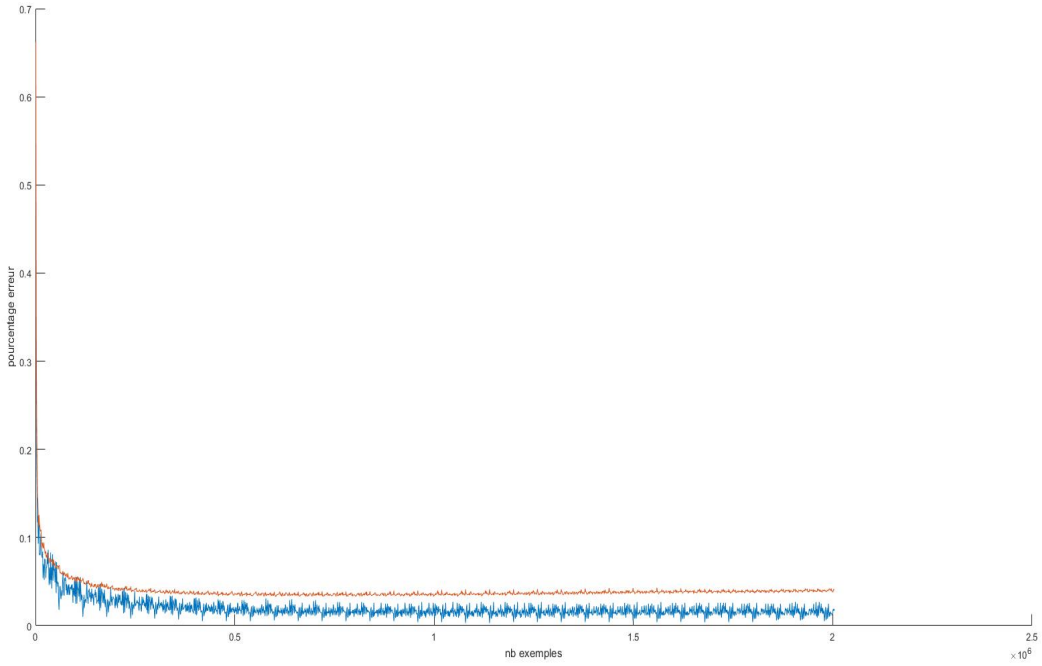

FIGURE 3.5 – pourcentage d'erreur de learning(bleu) et de test(orange) lors de l'apprentissage

étude supplémentaire possible Une étude statistique de l'apprentissage aurait pu être intéressane, 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.

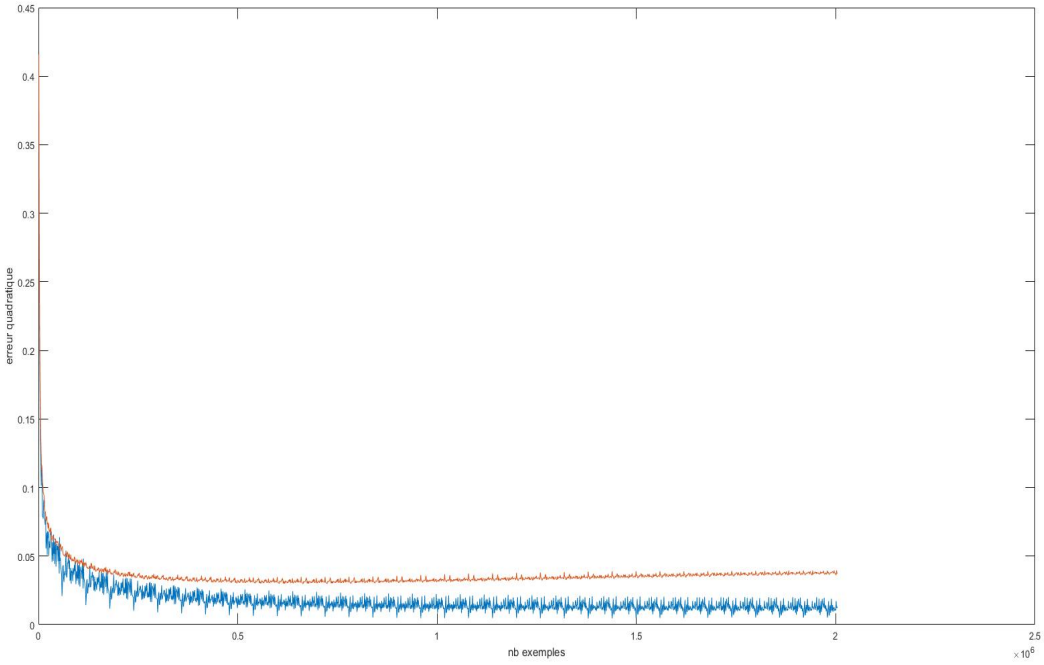

FIGURE 3.6 – erreur quadratique de learning(bleu) et de test(orange) lors de l'apprentissage

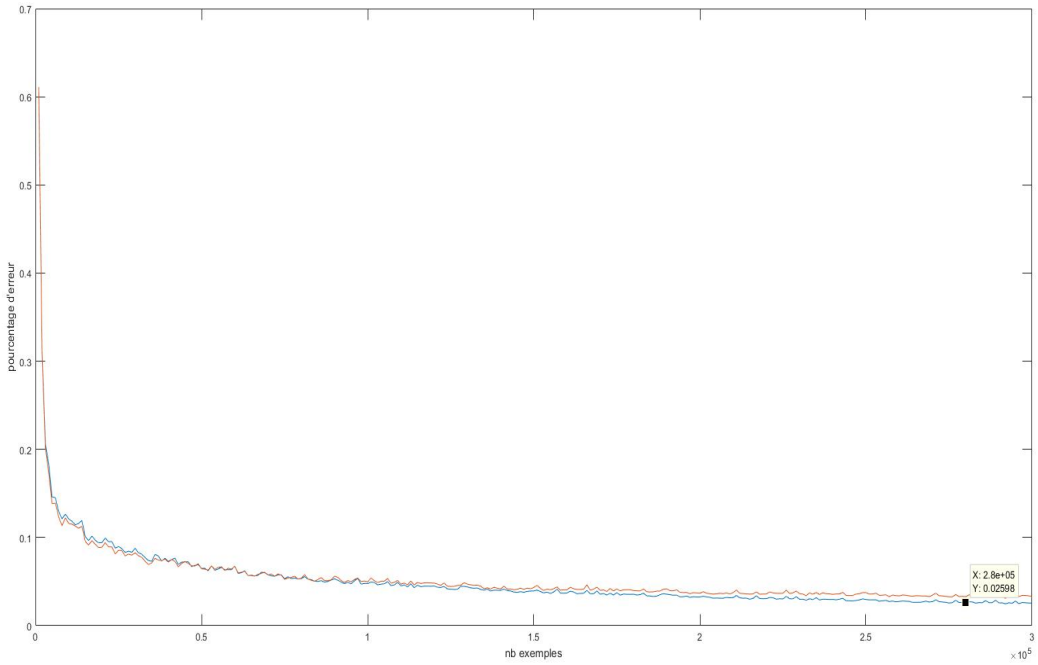

FIGURE 3.7 – Courbes du pourcentage d'erreurs de test et de learning moyennés sur 4 éxecutions

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 prinipale différence entre les réseaux neuronnaux comme les réseaux neuronnaux 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 oeuvres et étudiées par G. Hinton(2006). Nous allors présenter dans cette partie leur constitution et leurs fonctionnement.

5.1 Principe

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 On introduit maintenant l'échantillonage de Gibbs, qui est un algorithme de la classe des Metropolis-Hasting. Cet algorithme est un Monte Carlo d'une chaine de Markov dont l'idée principale est d'échantillonner les variables en se basant sur la valeur des autres variables. C'est une marche aléatoire sur les chaînes de Markov.

A expliquer : MRF et les propriété –> chaine de Markov Particularité et donc convergence –> contrastive divergence.

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.

Chapitre 6

Les Deep Belief Networks

Chapitre 7

Aller plus loin

7.1 La visualisation des spectres du réseau

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).

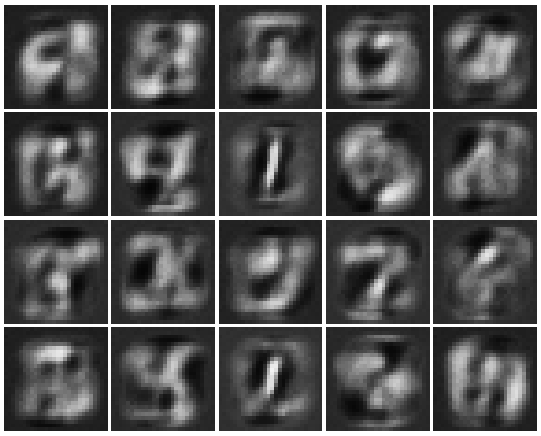

FIGURE 7.1 – Spectre d'une RBM entraînée à vingt neurones cachés.

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 indépendemment 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'exmple 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 éxécuter 20 fois le programme pour un learning rate donné. On obtient la courbe à la figure 8

On aperçoit donc un plateau aux alentours des 42 pourcents 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 pourcents d'erreur de learning. On peut donc dire que cette méthode est moins avantageuse que celle du perceptron.

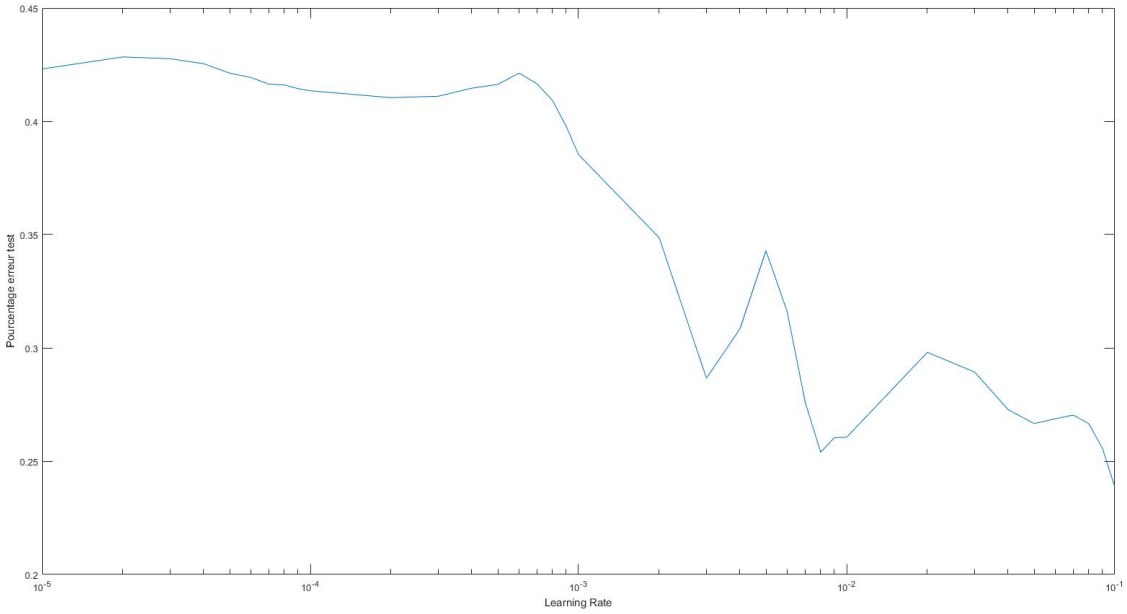

FIGURE 7.2 – erreur de learning en fonction du learning rate

Annexe A

La base MNIST.

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 défault. 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 fesant 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.