Deep Neural Networks

Bassem JABER - Nour El Houda ZRIBI

5 Février 2020

Table des matières

1 Introduction

Dans ce TP nous nous intéressons à la création d'un réseau de neurones en utilisant des fonctions de base sans faire appel aux librairies spécialement conçues pour l'implémentation de ceux-ci. L'objectif est de comprendre en détail le fonctionnement d'un réseau de neurones simple, qui constitue la base de la majeure partie des réseaux de neurones améliorés trouvables dans les différentes publications disponibles.

2 Données

Dans ce TP nous utiliserons deux bases de données différentes avec des objectifs distincts.

1. Binary AlphaDigits : Il s'agit d'un ensemble de données que l'on utilisera dans un premier lieu pour tester nos programmes et vérifier que tout fonctionne normalement. Les images sont de petites tailles (20x16) donc le calcul sera moins coûteux.
2. MNIST : Il s'agit d'une base de données connues et très utilisée en apprentissage statistique. Les images sont de plus grande taille (28x28) et de nombreux chercheurs se sont déjà penchés sur le problème de reconnaissance des caractères de la base MNIST.
   L'objectif sera donc à cette occasion de comparer les performances de notre réseau en variant les paramètres et par rapport aux modèles déjà existant.

3 Fonctions élémentaires

3.1 Construction d'un RBM et test sur Binary AlphaDigits

3.1.1 Lecture des données

Afin d'avoir en entrée des données lisibles pour notre algorithme, nous transformons les images à notre disposition en vecteurs ligne. Pour une image d'entrée de dimension $p*q$, l'objectif est d'avoir un vecteur ligne de dimension $[1, p*q]$.
Pour ce faire, nous définissons une fonction qui réalise cette procédure pour chaque image d'entraînement à notre disposition. Elle prend en argument la base de données d'entraînement ainsi que la liste des caractères que nous souhaitons faire apprendre à notre RBM.
Elle retourne un vecteur de taille $[n, p*q]$ avec $n = k*N$, N le nombre d'image d'entraînement pour chaque caractère (fixe) et k le nombre de caractères que nous souhaitons apprendre (variable, choix de l'utilisateur)

3.1.2 Initialisation du RBM

Nous souhaitons maintenant définir la structure fondamentale de notre réseau de neurone. Nous définissons notre Restricted Boltzmann Machine (RBM) par la donnée d'une matrice de poids W, un vecteur de biais pour les entrées a et un vecteur de biais pour les sorties b.
Nous définissons donc une classe pour initialiser nos RBM, la fonction qui effectue l'initialisation prend en entrée les deux paramètres p et q, dimensions de nos images d'entraînement, et donne en

sortie un RBM qui possède les composantes définies plus haut, la matrice de poids W, le vecteur de biais des unités d'entrées a et le vecteur des biais des unités de sorties b.

3.1.3 Calcul des sorties en fonction des entrées

Nous souhaitons définir une fonction nous permettant de calculer la valeur des sorties de notre RBM en fonction de ses entrées.
Nous choisissons la fonction sigmoïde pour effectuer ce calcul. Elle est définie par :

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Ce choix est arbitraire et d'autres solutions auraient pu être envisagées, par exemple la fonction $ReLU = max(0, x)$, qui est désormais plus utilisée dans la recherche sur le Machine Learning que la sigmoïde.
Nous définissons donc une fonction prenant en entrée notre RBM et nos données d'entrées (résultat de la lecture des données en 3.1.1) et nous donnant en sortie les valeurs calculées à l'aide de la fonction sigmoïde.

3.1.4 Calcul des entrées en fonction des sorties

Désormais, nous voulons effectuer le chemin inverse. Il nous faut déterminer les entrées que nous avions au début à l'aide des sorties que nous avons en notre possession.
Cela nous servira à déterminer le taux d'erreur de reconstruction de notre RBM et donc de mesurer ses performances.
Nous définissons une fonction dans ce but. Elle prend en entrée le RBM, nos données de sortie et elle retourne les données d'entrée en appliquant la fonction inverse de la sigmoïde.

3.1.5 Apprentissage non supervisé du RBM

Nous avons désormais assez d'outils à notre disposition pour nous atteler à l'apprentissage de notre RBM.
Nous définissons une fonction qui prend en entrée notre RBM, un nombre d'itérations à effectuer pour notre algorithme de descente de gradient, un learning rate, une taille de mini-batch et nos données d'entrée. Elle nous retourne notre RBM entraîné de manière non supervisée à l'aide de l'algorithme de Contrastive Divergence.
C'est dans cette étape que nous pouvons mesurer la qualité de notre RBM en fonction du taux d'erreur de reconstruction des données d'entrée par les données de sortie.

3.1.6 Génération d'images à l'aide du RBM

Nous disposons désormais d'un RBM entraîné à détecter certains caractères.
Nous pouvons désormais lui faire générer des images qui ressemblent aux images qu'il a apprise dans la base de données d'entraînement. Nous définissons une fonction qui réalise cette tâche. Elle prend en entrée notre RBM entraîné, le nombre d'itération de notre échantillonneur de Gibbs (c'est l'algorithme que l'on utilise pour générer notre échantillon en fonction des probabilités retenues lors de l'apprentissage) et le nombre d'images que l'on souhaite générer dans notre échantillon d'images.

On peut estimer la qualité de l'apprentissage à l'oeil nu en plus de la RMSE en observant la ressemblance de notre échantillon aux caractères appris.

Nous avons désormais terminé notre travail sur les RBM, nous avons toutes les fonctions à notre disposition pour l'entraînement, l'évaluation de la performance et la génération d'images ressemblant aux entrées étudiées. Il est donc temps de passer à la construction d'une réseau d'apprentissage profond. Cela consiste en "l'empilement" de différents RBM qui représenteront nos différentes couches cachées pour notre réseau de neurone. C'est très pratique étant donné que cela nous permet de réutiliser notre code initial en partie pour coder notre DBN.

3.2 Construction d'un DBN et test sur Binary AlphaDigits

3.2.1 Initialisation du DBN

Pour construire notre DBN, nous commençons par l'initialiser. Pour ce faire, nous avons besoin de la taille que nous souhaitons donner à notre réseau, c'est à dire une liste d'entier, chaque élément de la liste correspond à une couche de notre réseau et la valeur de cet élément désigne la quantité de neurones que nous utilisons dans cette même couche. Cette fonction d'initialisation nous retourne une instance de la classe DNN, caractérisée par une matrice de poids W, une liste de biais d'entrée A et une liste de biais de sortie B. Nous remarquons que ce choix de structure imite une liste de RBM, ce qui facilite la réutilisation des méthodes et fonctions définies précédemment.

3.2.2 Apprentissage non supervisé du DBN

Après avoir initialisé notre DBN, nous passons maintenant à l'apprentissage de celui-ci, de manière non supervisé, pour lui permettre une première approche de notre ensemble d'entraînement.

Pour ce faire, nous définissons une fonction qui nous retournera notre DBN entraîné. Elle prend en argument notre DBN initialisé, le nombre d'itération pour notre méthode de descente de gradient, un taux d'apprentissage, une taille de batch de données et nos données d'entraînement.

Notre méthode de descente de gradient n'effectue pas ses calculs sur l'ensemble des données mais uniquement sur un batch de taille définie par l'utilisateur. Bien que cela réduit la performance de la recherche du minimum de notre fonction de coût, cela permet de réduire drastiquement le temps de calcul pour une assez bonne approximation de l'estimation du minimum par la descente du gradient sur l'ensemble des données pour des batchs de taille raisonnable, par exemple 50 données.

Le nombre d'itération dans cette méthode permet aussi de jouer sur le compromis que nous souhaitons effectuer entre la qualité de l'estimateur et le temps de calcul.

Pour finir, le taux d'apprentissage caractérise l'importance que nous donnons aux nouvelles informations apprises à chaque itération de batch de données.
Un taux proche de 1 signifie que les nouvelles données apprises sont très importante par rapport aux anciennes données déjà connues et qu'il faut donc donner peu de poids à ces dernières.
Au contraire, un taux proche de 0 signifie que l'on a une grande confiance dans les données déjà

connues et que l'on suppose que l'apprentissage est bénéfique mais qu'il faut lui accorder une importance plus faible.
La valeur du taux d'apprentissage est donc très dépendante de la situation dans laquelle nous travaillons et il est difficile de lui donner une valeur de manière générale sans effectuer des test préliminaires pour évaluer son impact sur la qualité de nos prédictions.

3.2.3 Génération d'images par le DBN

Sur le même principe que la générations d'image par notre RBM, nous souhaitons générer des images ayant les mêmes caractéristiques que celle que notre DBN a apprises.
Pour ce faire nous définissons une fonction prenant en argument notre DBN pré-entraîné, un nombre d'itérations pour notre échantillonneur de Gibbs et un nombre d'images à générer. Cette fonction initialise aléatoirement une matrice de taille $[n, p*q]$ avec n = Nombre d'image à générer et $p*q$ la dimension de l'image à créer. Nous rappelons que nous considérons les images comme des vecteurs lignes de taille $[1, p*q]$ dans nos travaux.
L'échantillonneur de Gibbs va utiliser notre "image" initialisée aléatoirement comme image initiale. A chacune de ses itérations, il utilisera les probabilité apprises que le pixel étudié (défini par une position dans notre image) soit de valeur 1 ou 0 en fonction de son voisinage.
Après avoir affecté une valeur à ce pixel, il passe au pixel suivant et ainsi de suite jusqu'à affecter une valeur à chaque pixel.
Ensuite, il itère un nombre de fois égal au nombre d'itération et nous savons que ce processus converge vers une image ayant les même caractéristique que celles qui ont été apprises au préalable.
Le nombre d'itération est un choix arbitraire, nous aurions pu tout aussi justement choisir de nous arrêter quand l'image ne varie plus beaucoup, encore faut-il savoir comment déterminer la variation de manière correcte et cela pourrait impliquer un grand nombre d'itération, d'où notre choix d'utiliser un nombre d'itération prédéfini.

4 Travail préliminaire sur Binary AlphaDigits

4.1 Initialisation et entraînement d'un RBM à détecter la lettre E

Nous avons terminé la définition des fonctions nous permettant l'apprentissage non supervisé d'un RBM et d'un DBN.
Nous passons désormais à une phase de test de ces fonctions, pour vérifier que nos algorithmes effectuent correctement les tâches qu'ils sont supposés réaliser et évaluer les performances en terme de racine de l'erreur quadratique moyenne (RMSE).
Nous commençons par l'initialisation d'un RBM avec une couche d'input de 320 neurones (correspondant au nombre de pixels dans les images de notre base de données) et une couche de sortie avec 200 neurones. Nous chargeons nos données, nous choisissons d'apprendre la lettre E puis nous entraînons notre RBM. Ci-dessous, nous trouvons le résultat suivant.

```
Entrée [42]: RBM0 = init_RBM(320,200)
             a=loadmat("C:/Users/jaber/Downloads/binaryalphadigs.mat")['dat']
             b=lire_alpha_digit(a,"e")
             RBM_T=train_RBM(RBM0,30,0.1,8,b)
```

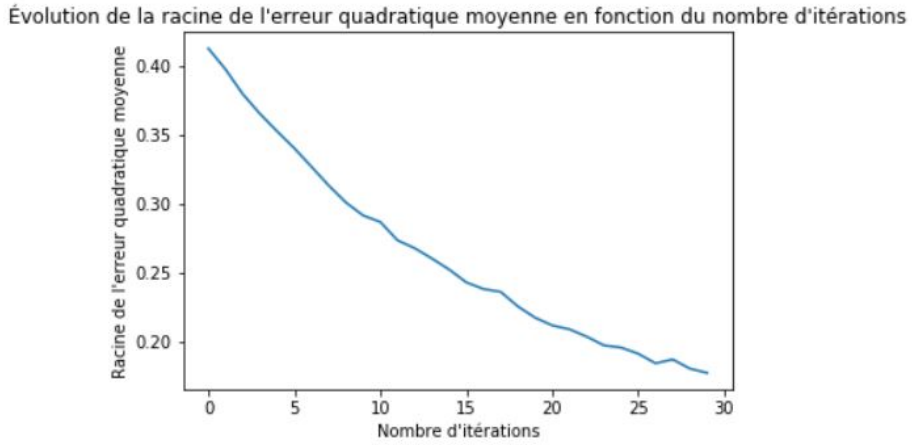

FIGURE 1 – Apprentissage non supervisé du RBM et évolution de la RMSE.

Le résultat est cohérent avec ce que l'on attend. Au fur et à mesure que notre RBM étudie les données de la base de données, sa prédiction du caractère est également de plus en plus juste.

4.2 Génération d'images ayant les caractéristiques de la lettre E avec un RBM

Nous souhaitons désormais évaluer de manière visuelle si les caractéristiques de la lettre E sont bien apprises, c'est à dire 3 barres horizontales reliées par une barre verticale à gauche de ces dernières.
Nous générons donc une image à l'aide de notre RBM entraîné et nous obtenons le résultat suivant.

```
Entrée [34]: simulation_images_RBM = generer_image_RBM(RBM_T,80,4)
```

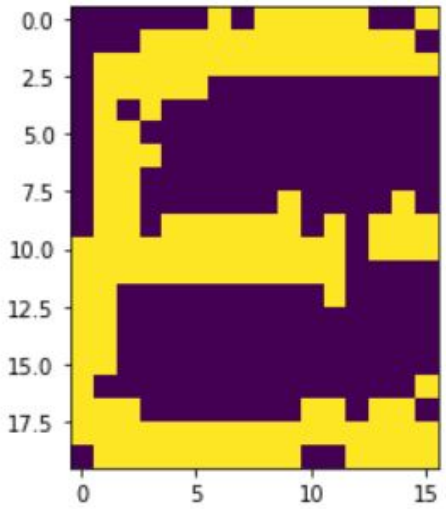

FIGURE 2 – Génération d'une image ayant les caractéristiques de la lettre E via notre RBM entraîné.

Nous avons généré 4 images et nous choisissons celle qui semble être le plus proche de la réalité, les images sont toutes conformes aux caractéristiques de la lettre E mais la première était la plus ressemblante (cf. première partie du code jointe pour voir les autres images)
Nous observons bien les caractéristiques de la lettre E citées plus haut. Nous pouvons donc considérer que notre RBM a correctement appris à distinguer un E et que nos fonctions retournent le bon résultat pour les "bonnes" raisons.

4.3 Initialisation et entraînement d'un DBN à détecter la lettre E

Nous passons maintenant à la suite de notre étude, nous nous intéressons dorénavant à la construction d'un DBN et à l'apprentissage de la même lettre que précédemment.

Nous obtenons le résultat suivant :

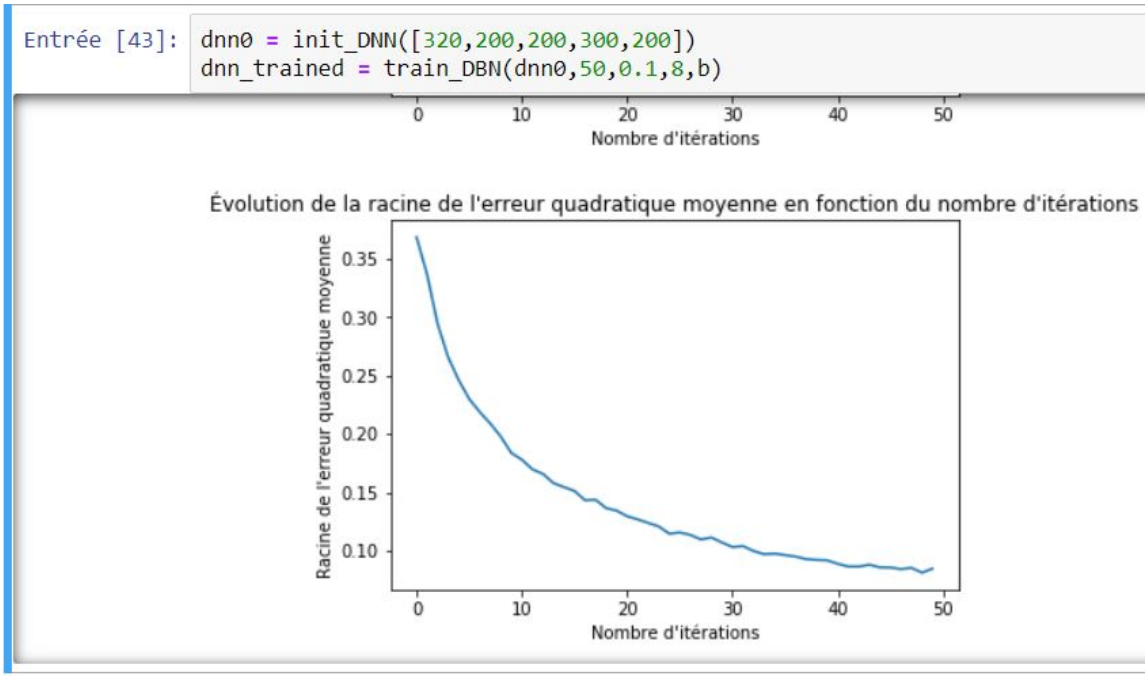

FIGURE 3 – Apprentissage non supervisé du DBN et évolution de la RMSE.

Étant donné que nous utilisons en boucle la fonction d'entraînement du RBM, nous obtenons plusieurs courbes d'évolution de la RMSE en fonction du nombre d'itérations.
La courbe qui nous intéresse correspond à celle de la dernière couche que nous pouvons observer ci-dessus.
Encore une fois le résultat est cohérent avec nos attentes, nous gagnons en RMSE puisque nous la diminuons de moitié.
Le DBN a bien logiquement un meilleur pouvoir de reconnaissance de la lettre E qu'un RBM simple, ce qui justifie l'empilement de ces dernier pour créer un réseau plus grand.

4.4 Génération d'images ayant les caractéristiques de la lettre E avec un DBN

Nous poursuivons notre étude avec la génération d'image correspondant à la lettre E par notre DBN.

Nous utilisons notre fonction de création d'image et nous obtenons le résultat ci-dessous.

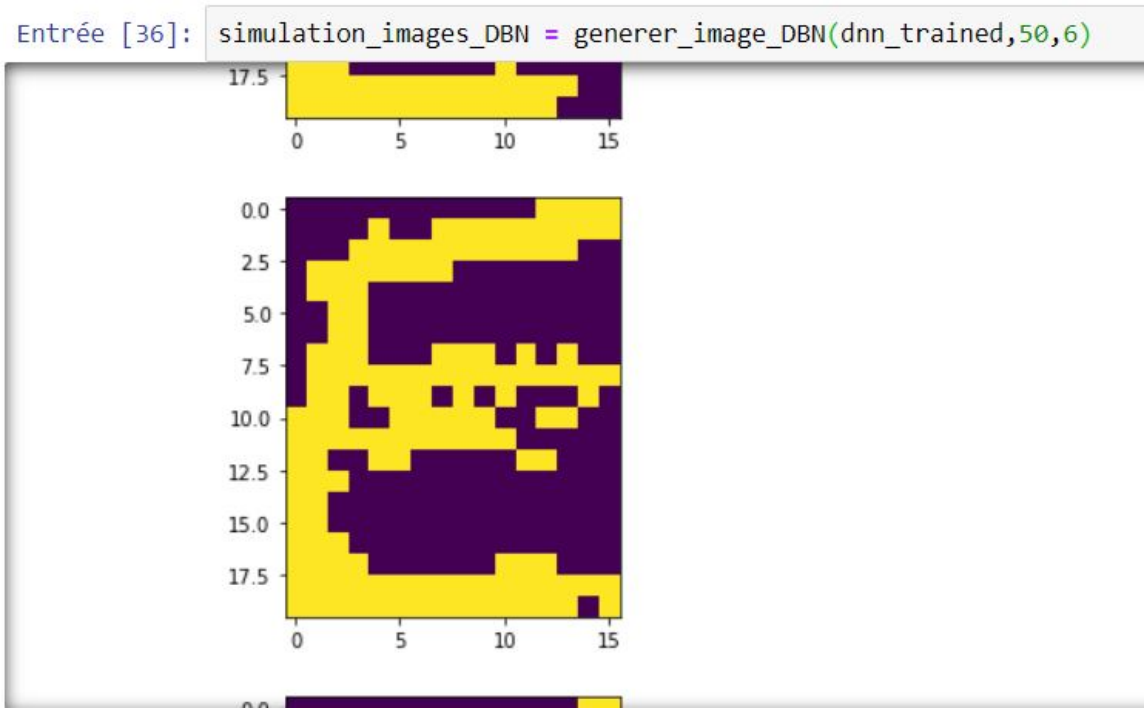

FIGURE 4 – Génération d'une image ayant les caractéristiques de la lettre E via notre DBN entraîné.

Nous choisissons cette fois-ci de générer 6 images et d'utiliser celle ressemblant le plus à un E. Les résultats sont ici un peu moins ressemblant à ce qu'un être humain considère être un E, cependant, la RMSE est bien plus faible que pour notre RBM.
Nous pouvons en déduire que cette représentation permet peut-être à notre DBN de mieux généraliser l'aspect d'un E et de ne pas être trop sensible aux variations entre les différentes écritures.

5 Construction d'un DNN et test sur MNIST

Nous revenons une dernière fois à la définition de fonctions pour nous atteler désormais à l'étude de la base de données MNIST.
Toutes nos fonctions précédentes ne sont pas obsolètes, au contraire, nous les réutilisons toutes, nous souhaitons juste désormais travailler avec plus d'outils à notre disposition et surtout sur une base de données connues et utilisée par de nombreux chercheurs en Machine Learning pour mieux pouvoir situer notre réseau de neurone créé de 0 par nos soins.

5.1 Préparation du calcul de la sortie

Pour débuter, nous définissons une fonction nous permettant de calculer les sorties de notre réseau de neurones à l'aide de la fonction softmax. Elle prend en argument un DNN et les données d'entrées puis elle retourne la valeur de la fonction softmax pour le vecteur des données d'entrée.

Nous rappellons la définition de la fonction softmax :

$$Softmax(x_j) = \frac{e^{-x_j}}{\sum_{i=1}^{n} e^{-x_i}}$$

Cette fonction peut être vue comme la généralisation de la fonction sigmoïde pour un vecteur de taille n, c'est pourquoi nous l'utilisons dans notre cas puisque nous utilisions la fonction sigmoïde pour le calcul des sorties sur une couche.

5.2 Calcul de la sortie du réseau

Maintenant que nous savons comment calculer la sortie du réseau, nous définissons une fonction pour l'effectuer.
Cette fonction prend en argument un DNN et nos données d'entrée pour retourner une matrice contenant l'ensemble des sorties pour chaque couche cachée ainsi que les probabilité pour les unités de sorties.
Cela revient tout simplement à appliquer le calcul de la sortie d'un RBM pour chaque couche et les garder en mémoire dans une liste.

5.3 Calcul de la rétropropagation

Nous souhaitons désormais appliquer la méthode de rétropropagation à notre réseau de neurones afin d'améliorer ses performances en terme de prédiction du label de l'image analysée.
Nous définissons donc une fonction prenant en argument un DNN, un nombre d'itération pour notre méthode de descente de gradient, un taux d'apprentissage, une taille de batch, nos données d'entrée et le label de celles-ci.
La rétropropagation diffère de l'entraînement de nos structures dans le sens où elle agit de manière rétroactive sur les poids et les biais, alors que l'entraînement agissait "vers l'avant", il modifiait les poids et biais suivants pour avoir une meilleure prédiction.
Ces deux fonctions ont un objectif commun : celui d'améliorer la qualité de notre prédiction et elles ne diffèrent que dans le sens de leur application.
Notre objectif sera de comparer les performances d'un DNN entraîné puis utilisant la rétropropagation et un DNN n'utilisant que la rétropropagation comme méthode d'amélioration de sa prédiction.

Nous pouvons voir que les arguments de la fonction sont semblables à celles du train. Nous utilisons encore une fois les même méthodes pour minimiser notre fonction de coût que nous choisissons comme étant l'entropie croisée. On cherche à minimiser cette fonction par descente du gradient sur un batch de données pour réduire le temps de calcul et nous affectons d'un taux d'apprentissage les nouvelles informations apprises pour leur donner un poids plus ou moins élevé.

5.4 Test des performances de notre DNN

La dernière fonction que nous définissons nous permet de mesurer les performances de notre DNN non pas sur les données d'entraînement mais sur des images de test que le DNN n'a jamais vu auparavant. L'objectif est de déterminer sa capacité de généralisation et pour ce faire nous utilisons la précision de notre réseau de neurone comme métrique de performance.
Cela correspond au nombre de prédiction justes sur le nombre total de prédiction réalisées. Il existe d'autre métriques plus sophistiquée mais nous nous contenterons dans notre cas de cette métrique qui est également celle utilisée par les autres chercheurs ayant déjà travaillé sur cette tâche.

6 Travail et analyse sur MNIST

Dans cette partie, nous cherchons à comparer les performances de deux types de réseau de neurones. Les deux cas de figures qui se posent sont : le cas d'un réseau de neurones pré-entraîné et le cas d'un réseau de neurones non pré-entraîné. Pour ce faire, nous allons étudier l'évolution des performances des réseaux de neurones selon le nombre de couches, le nombre de neurones dans un nombre de couches fixé, et la taille de la base de données utilisée pour l'apprentissage.

6.1 Taux d'erreur en fonction du nombre de couches cachées dans le DNN

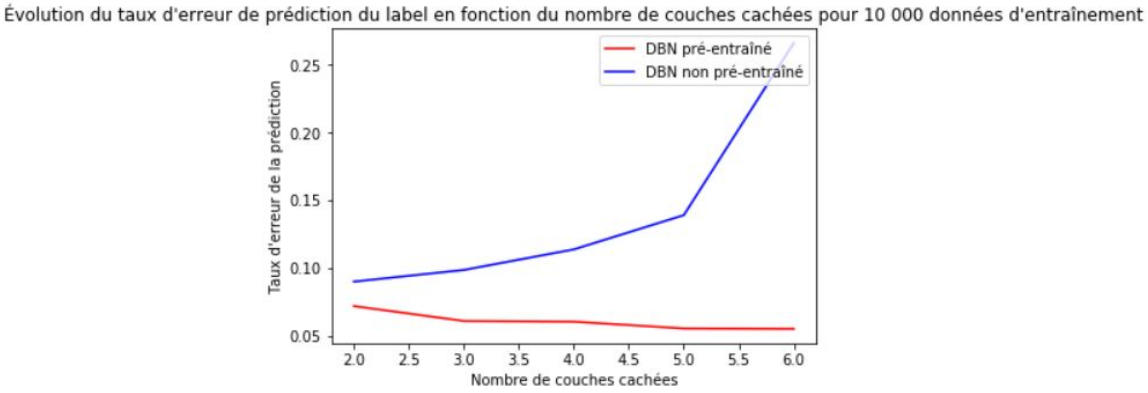

FIGURE 5 – Évolution du taux d'erreur en fonction du nombre de couches de 200 neurones pour 10 000 données d'entraînement.

En observant les courbes ci-dessus,nous observons que le taux d'erreur diminue en augmentant le nombre de couches de notre réseau de neurones pour le réseau pré-entraîné.
Le taux d'erreur évolue plus précisément de 0,0719 pour une seule couche cachée à 0,0551 pour 5 couches cachées.
Nous remarquons également que le taux d'erreur est beaucoup plus élevé pour le DBN non pré-entraîné, en plus d'augmenter avec le nombre de couche cachées, que pour le DBN pré-entraîné.
Cela semble être contre-intuitif de penser que le taux d'erreur augmente en fonction du nombre de couche cachées pour un réseau non pré-entraîné, étant donné que nous savons que nous sommes sensés améliorer notre estimation de la fonction avec notre réseau de neurones de manière linéaire selon le nombre de neurones dans l'ensemble des couches, mais il existe parfois des différences entre la théorie et la pratique puisque la théorie nous donne une valeur seuil et non pas une valeur exacte pour notre situation.

6.2 Taux d'erreur en fonction du nombre de neurones pour un DNN de 2 couches

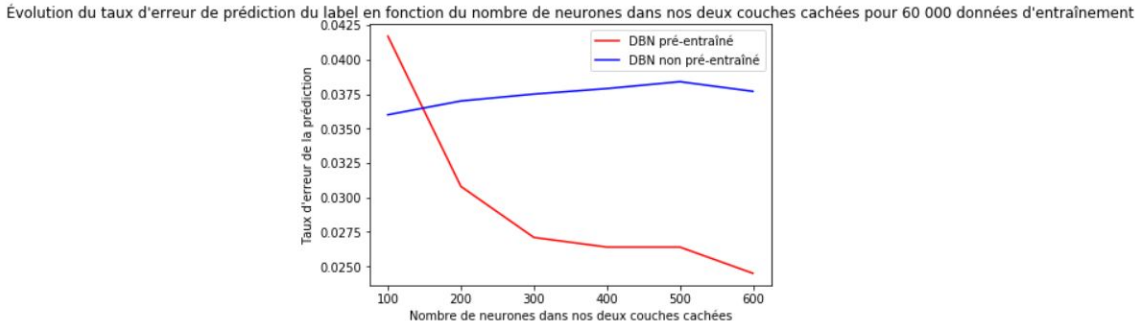

FIGURE 6 – Évolution du taux d'erreur en fonction du nombre de neurones pour deux couches cachées.

Nous nous intéressons maintenant à l'évolution du taux d'erreur en fonction du nombre de neurones par couche pour un réseau de neurone fixé au préalable composé d'une couche d'input, une couche d'output et deux couches cachées.
Nous pouvons observer que pour notre réseau pré-entraîné, nous avons une diminution du taux d'erreur en fonction du nombre de neurones.
Nous passons plus précisément de 0.0417 pour deux couches de 100 neurones à 0.0245 pour deux couches de 600 neurones. Cela semble cohérent avec nos attentes.
Encore une fois, le DBN non pré-entraîné n'a pas un taux d'erreur qui diminue forcément en fonction du nombre de neurones dans nos couches cachées, il varie peu mais augmente dans un premier temps avant de diminuer entre 500 et 600.
On observe également que son taux d'erreur pour 100 neurones est plus faible que pour celui du réseau pré-entraîné. Cela peut montrer certaines limite d'un apprentissage non supervisé préalable pour un nombre de neurones trop faible.

6.3 Taux d'erreur en fonction du nombre de données d'entraînement pour un DNN de 2 couches

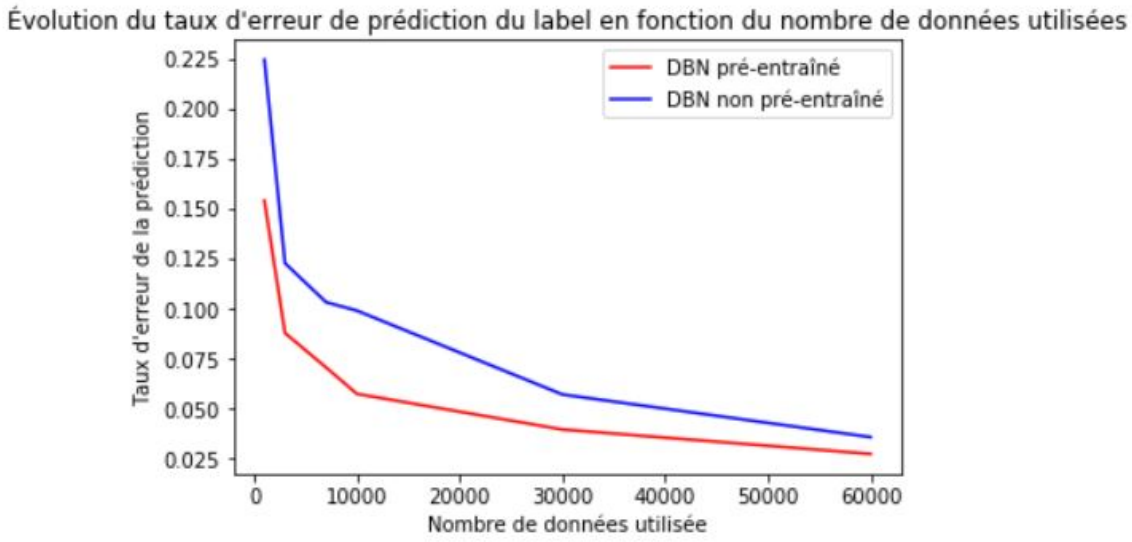

FIGURE 7 – Évolution du taux d'erreur en fonction du nombre de données d'entraînement utilisées

Pour terminer notre étude, nous nous intéressons finalement à l'évolution du taux d'erreur à l'évolution du taux d'erreur en fonction du nombre de données d'entraînement en entrée.

Nous nous réconcilions désormais un peu plus avec la théorie. Nos deux réseaux de neurones, pré-entraîné ou non, ont un taux d'erreur qui diminue fortement en fonction du nombre de données que nous donnons à notre réseau de neurones. Nous observons que la différence entre les deux taux d'erreur diminue également avec le nombre de données utilisées, pour 1000 données d'entraînement, nous avons une différence de 0.0705 et une différence de 0.0084 pour 60000 données d'entraînement.

7 Conclusions

A la lumière de notre étude nous pouvons tirer quelques conclusions.

1. Importance prédominante du nombre de données d'entraînement : Il semble bien que la caractéristique majeure intervenant dans l'évolution du taux d'erreur correspond au nombre de données d'entraînement.
   C'est pourquoi nous savons qu'aujourd'hui la première contrainte en terme d'apprentissage supervisé est dans les bases de données labelisées à notre disposition. Il existe de nombreuses recherches dont l'objectif est de faciliter ce processus long et fastidieux qui est aujourd'hui effectué par des humains et qui prend un temps très long.
   L'idéal serait d'avoir une intelligence artificielle (IA) capable de labeliser des images d'elle-même mais pour le moment il serait déjà intéressant d'avoir une IA permettant d'accélérer le processus

2. Structure la plus performante : En analysant nos résultats, il semble que la structure permettant d'obtenir le taux d'erreur le plus faible serait un réseau de neurone pré-entraîné avec 6 couches cachées de 600 neurones chacune en utilisant l'ensemble de nos 60000 données d'entraînement.
   Nous n'avons malheureusement pas eu le temps de réaliser ce test car cela prendrait plusieurs jours de calcul, même en accélérant notre fonction à l'aide de CuPy. Il est intéressant de noter que le réseau de neurone (non convolutionnel) le plus performant actuellement existant est un réseau de 5 couches cachées. La structure est la suivante 784-2500-2000-1500-1000-500-10 et le taux d'erreur final est de 0.0035, ce qui est bien plus faible que notre meilleur taux à peu près égal à 0.025.