Deep Learning II : Projet

Armand Foucault (armand.foucault@polytechnique.edu)
Lotfi Kobrosly (lotfi.kobrosly@polytechnique.edu)

Professeur :
Yohan Petetin

Département :
Communications, Images and Information Processing
Télécom SudParis

4 avril 2021

Table des matières

1 Introduction

Ce document présente les résultats de nos travaux sur l'analyse de l'influence de la taille des données d'entraînement et des paramètres structurels (profondeur des couches et nombre d'unités par couche) d'un réseau de neurones *fully-connected* sur les performances de celui-ci. Nous avons pour chaque étude observé nos résultats pour un réseau de neurones pré-entraîné à l'aide d'une structure *Deep Belief Network* et un réseau à initialisation de poids aléatoire.

Nous avons pour cela implémenté en *python* les architectures *Restricted Blotzmann Machine* (RBM), *Deep Belief Network* (DBN) et *Deep Neural Network* (DNN) ainsi que leurs procédures d'apprentissage *à partir de rien* (sans l'aide des librairies usuelles de *deep learning*). Puis nous avons mesuré la performance en terme de taux d'erreur de prédiction de différentes architectures en faisant varier à chaque fois un paramètre - taille des données d'entraînement, nombre d'unités par couche ou nombre de couches cachées - pour un réseau pré-entraîné et un réseau "vierge" sans pré-apprentissage. Notre repo github est accessible **ici**.

Le présent document se divise en deux parties ; dans une première nous décrivons brièvement l'implémentation que nous avons proposé, et dans une seconde nous exposons les analyses et comparaisons réalisées sur les performances des différents réseaux.

2 Implémentation

Dans un premier temps, nous présentons quelques éléments d'implémentation à disposition du lecteur, permettant de mieux comprendre le fonctionnement général du code.

Les architectures des réseaux et l'entraînement de ceux-ci sont codés dans les fichiers *principal_RBM_alpha.py*, *principal_DBN_alpha.py* et *principal_DNN_MNIST.py*. Nous avons suivi les recommandations de l'énoncé en écrivant les fonctions entrée-sortie / sortie-entrée, *train* et *pretrain* pour les différentes architectures RBM, DBN et DNN.

Afin de faciliter le stockage et la mise à jour des paramètres des réseaux, nous avons choisi de créer des classes *RBMStruct* et *DNNStruct* respectivement. La première a pour attributs *input_bias*, *hidden_bias* et *weights* qui sont les paramètres standards d'un RBM ; elle contient également une méthode pour mettre à jour ces paramètres après entraînement. La seconde a pour attributs une liste de RBMs utiles à l'entraînement d'un DBN et une liste de paramètres qui sont ceux du DNN et qui sont partagés avec les biais cachés et les poids des RBMs (pour pouvoir passer de l'entraînement séquentiel des RBMs à l'initialisation des couches du DNN). La classe contient une méthode pour mettre à jour ces paramètres couche par couche, et de manière simultanée entre la couche RBM et la couche DNN, afin de toujours conserver l'alignement entre les deux.

Pour la phase d'expérimentation, nous avons écrit les scripts suivants :

— *main.py* : effectue une des trois expériences décrites dans la section 3.2, selon le choix de l'opérateur qui exécute le code. Il contient aussi deux fonctions : *init_dnns* qui initialise de façon alétoire deux DNNs identiques (sans couche finale) et *add_same_layer* qui ajoute la même couche (aléatoirement initialisée) à deux instances de DNNStruct. Il contient aussi une troisième fonction *get_errors* dont le but est une simple factorisation de code.

— *analysis_perfect_model.py* : génère un graphe du taux d'erreur par epochs du modèle *optimal* selon la description dans la section 3.4.

Ces deux scripts importent des fonctions introduites dans les autres scripts décrits plus haut pour un meilleur fonctionnement.

3 Expérimentation

3.1 Expériences à réaliser

Nous présentons maintenant les expériences que nous avons menées pour connaître l'intérêt de l'architecture que nous avons présentée, en la comparant notamment aux réseaux de neurones *classiques*. L'idée est de prendre deux réseaux identiques, dans notre cas instances de *DNNStruct*, de considérer le premier comme un DBN et par conséquent effectuer un pré-entraînement en faisant abstraction de la dernière couche (celle de l'output) et ne rien faire pour le deuxième. Ensuite, nous entraînons, dans les mêmes conditions, ces deux modèles sur les données puis nous comparons les résultats obtenus.

Nous nous intéressons particulièrement à 3 critères :

— L'influence du nombre de couches cachées

— L'influence de la taille des couches cachées *i.e* le nombre d'unités cachées représentées

— L'influence de la taille des données d'entraînement fournies aux modèles considérés.

Le but ensuite est d'obtenir les paramètres optimaux qui produisent les meilleures performances sur le dataset MNIST. Nous avons donc, pour chaque expérience, fixé les deux autres variables aux valeurs suivantes (selon l'expérience) : REF_N_HIDDEN_LAYERS = 2 ; REF_N_HIDDEN_UNITS = 200 ; REF_DATA_SIZE = 60000, et nous avons fait varier la variable d'intérêt dans les listes suivantes :

— N_HIDDEN_LAYERS = [2, 3, 4, 5, 6, 8, 10]

— N_HIDDEN_UNITS = [100, 200, 300, 500, 700]

— DATA_SIZE = [1000, 2000, 3000, 5000, 7000, 10000, 20000, 30000, 60000]

Pour toutes les architectures de DBN considérées, quel que soit le nombre de couches ou d'unités cachées et quelle que soit la taille des données considérées par la suite pour la phase d'entraînement, le pré-entraînement a été fait sur la totalité du dataset, c'est-à-dire, sur une concaténation des données d'entraînement et des données de test. Nous avons pour cela initialisé de manière aléatoire deux DBN identiques (en utilisant la méthode *update_layer* de la classe *DNNStruct*) sans couche de sortie aux dimensions imposées par le nombre de classes (qui est égal à 10). Ensuite, après ce pré-entraînement, nous rajoutons la même couche (initialisée aléatoirement également) aux deux DBNs (avec la méthode *stack_layer* de *DNNStruct*), pour pouvoir procéder à l'entraînement et à la phase de test.

Nous avons choisi de fixer les paramètres suivants pour l'entraînement et le pré-entraînement :

— Nombre d'epochs : 10

— Taille de batch : 64

— Taux d'apprentissage : 0.01

3.2 Résultats obtenus et interprétations

Nombre de couches cachées

Tout d'abord, nous visualisons les résultats obtenus pour différents réseaux de neurones pour lesquels nous faisons varier le nombre de couches cachées :

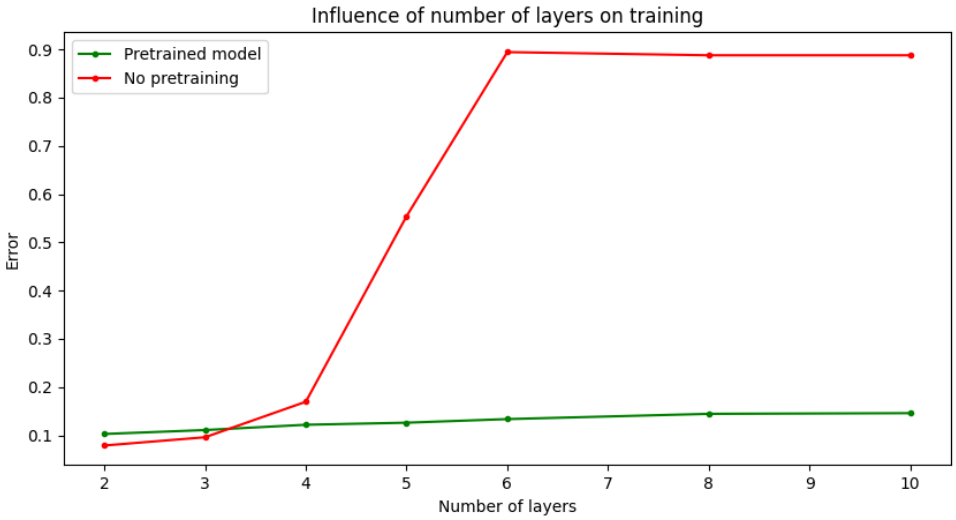

FIGURE 1 – Erreur de classification sur les données d'entraînement : Influence du nombre de couches cachées

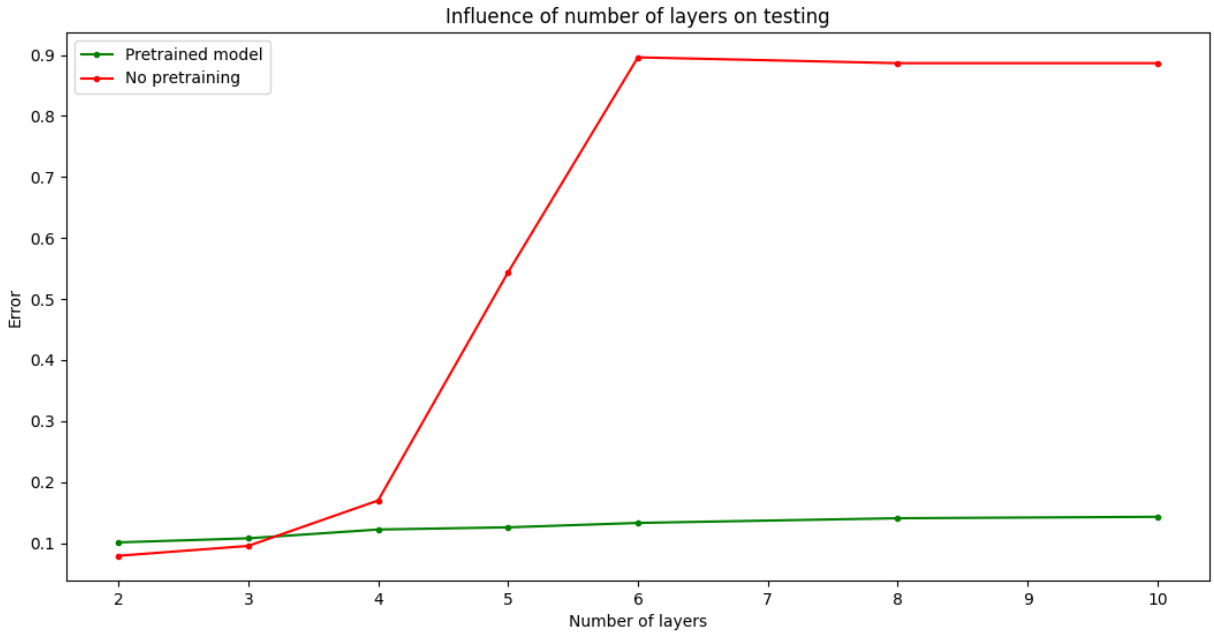

FIGURE 2 – Erreur de classification sur les données de test : Influence du nombre de couches cachées

Dans les deux graphiques ci-dessus, nous remarquons immédiatement deux choses :

— La performance du réseau sans pré-entraînement décroît significativement entre 4 et 6 couches cachées ;

— La performance du réseau pré-entraîné est presque constante par rapport au nombre de couches cachées (même si on note une très légère croissance du taux d'erreur)

Pour le premier point, nous nous sommes demandé ce qui pouvait causer une telle chute de la performance du réseau pré-entraîné, y compris sur le jeu de données d'entraînement. En effet, une première idée est que cela est dû à du sur-aprentissage (on augmente la complexité du modèle en augmentant le nombre de couches cachées). Mais précisément, dans ce cas-là, l'erreur d'entraînement devrait tendre vers 0, alors que l'on observe le phénomène opposé. Le problème venait donc d'ailleurs.

En cherchant l'origine du problème, nous avons analysé l'évolution de la fonction de perte pendant l'apprentissage pour différents nombres de couches. Une trace des ressources utilisées est donné dans la figure suivante :

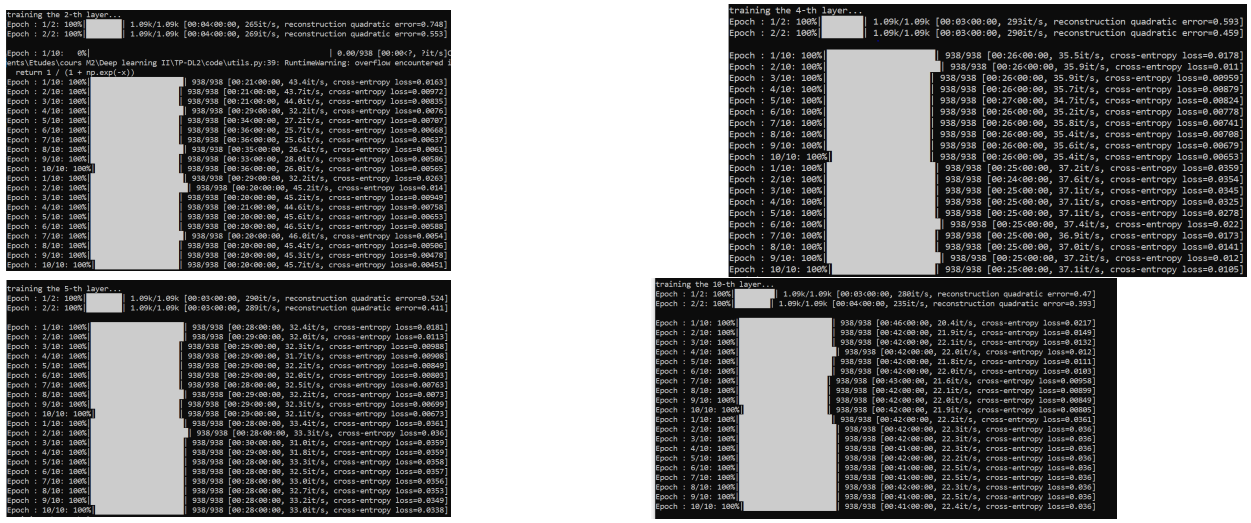

FIGURE 3 – De gauche à droite et de haut en bas : évolution de la fonction de perte pendant l'apprentissage pour un réseau pré-entraîné (10 premières lignes) et non pré-entraîné (10 dernières lignes) pour un nombre de couches cachées égal à 2, 4, 5 et 10 respectivement

Nous observons que la fonction de perte ne bouge presque pas pour un nombre de couches cachées égal à 5, et plus du tout pour un nombre de couches égal à 10. En revanche, pour le réseau pré-entraîné celle-ci continue de diminuer régulièrement, comme attendu.

L'explication la plus vraisemblable à ce phénomène est celle du *vanishing gradient*. Nous savons en effet que la dérivée de la fonction sigmoïde est toujours strictement inférieur à 1 ; pour cette raison, la dérivée partielle de la fonction de perte par rapport aux paramètres des couches les plus profondes, résultant du produit de plusieurs dérivées (étant donné la formule de dérivation de fonctions composées), tend rapidement vers 0 dans un réseau à activations sigmoïde. En conséquence, plus il y a de couches dans le réseau, plus le gradient de la perte par rapport aux couches profondes est proche de 0 (par coordonnée) - voire égal à 0. Et ces couches ne sont pas, ou à peine, mises à jour à chaque itération de descente de gradient stochastique.

Dans le cas de l'initialisation aléatoire (pas de pré-entraînement), le réseau échoue donc à apprendre des paramètres permettant de prédire correctement même les données d'entraînement, ce qui coïncide avec l'observation des figures 1 et 2, où l'erreur de prédiction est approximativement celle d'un prédicteur aléatoire uniforme (90%).

Inversement, pour un réseau pré-entraîné, les paramètres (pré)-appris ont été entraînés à redécrire les données dans un espace de variable latentes. Chaque couche du réseau est donc, avant même l'entraînement, à-même de passer l'information de la donnée d'entrée à la couche suivante, et la dernière couche avant la sortie du réseau reçoit en entrée un vecteur de variables latentes contenant approximativement l'information de la donnée d'entrée. Ce qui explique le fait qu'un réseau pré-entraîné soit relativement insensible au *vanishing gradient* et par extension au nombre de couches cachées dans le réseau.

Nombre d'unités cachées

Ensuite, nous obtenons les résultats concernant l'influence du nombre de neurones cachées, illustrés sur les figures 4 et 5, respectivement sur le jeu de donnée d'entraînement et celui de test.

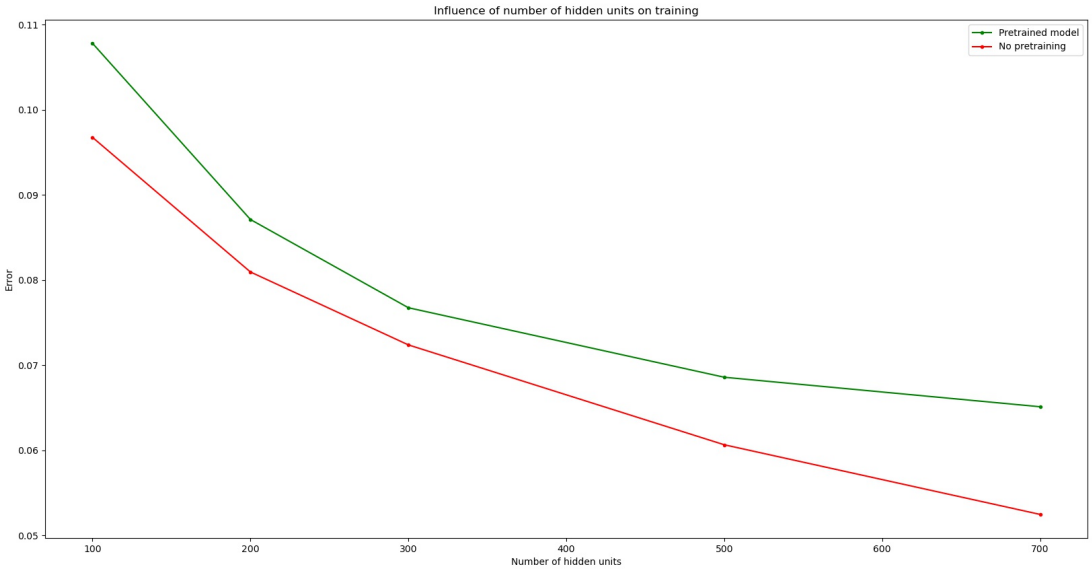

FIGURE 4 – Erreur de classification sur les données d'entraînement : Influence du nombre d'unités cachées

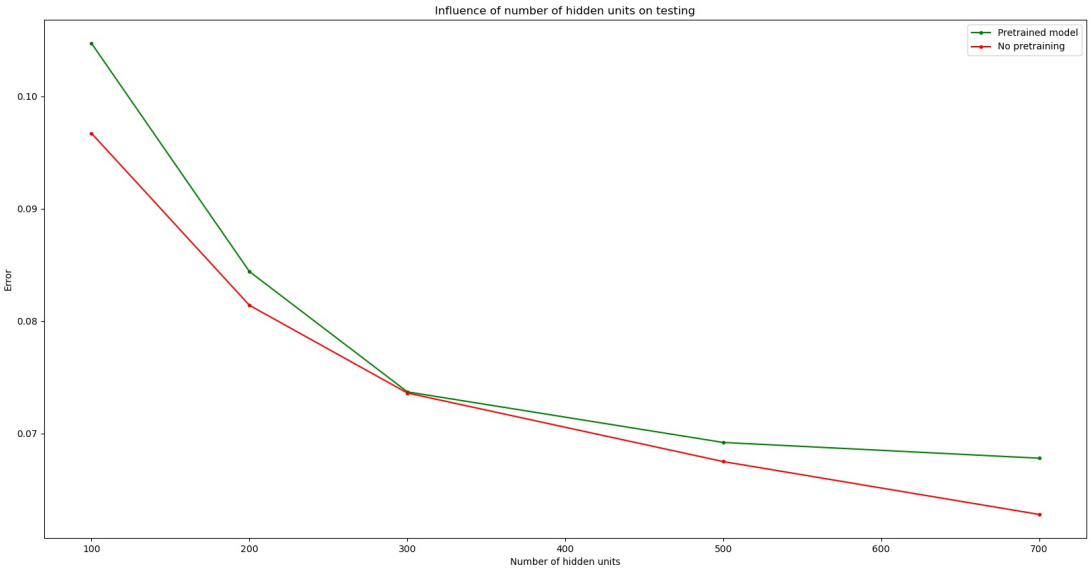

FIGURE 5 – Erreur de classification sur les données de test : Influence du nombre d'unités cachées

Nous remarquons que l'erreur de classification diminue quand le nombre de variables cachées par couche augmente, sur les deux jeux de données de manière très similaire. Ceci est cohérent avec ce qu'on pouvait prévoir théoriquement. En effet, les variables cachées permettent de créer une nouvelle représentation des paramètres/features des données d'entrée. Ceci veut dire que plus on a de variables cachées sur la même couche, plus nous sommes capables de séparer les caractéristiques descriptives des données d'entrée, ce qui permet de mieux définir les différences entre les classes de sortie.

Nous observons aussi des meilleures performances par le réseau non pré-entraîné, de l'ordre de 1%. Même si cela semble insignifiant, il reste intéressant surtout parce que c'est le cas pour toutes les valeurs de la taille des couches cachée, ce qui est en concordance avec les résultats de la première expérience.

Taille des données d'entraînement

Pour la troisième expérience réalisée, nous obtenons les résultats qui apparaissent sur les figure 6 et 7.

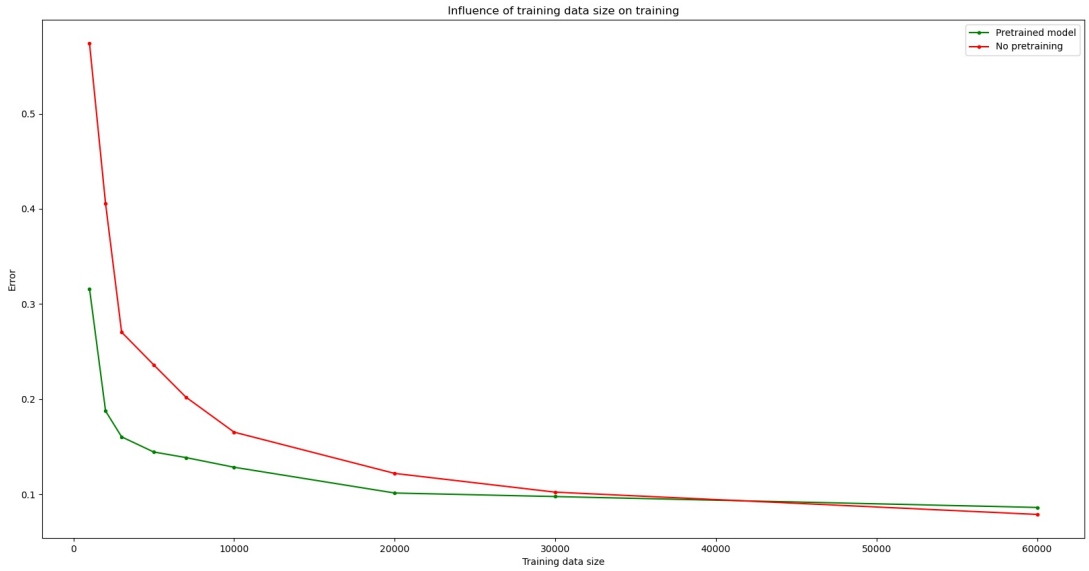

FIGURE 6 – Erreur de classification sur les données d'entraînement : Influence de la taille des données d'entraînement

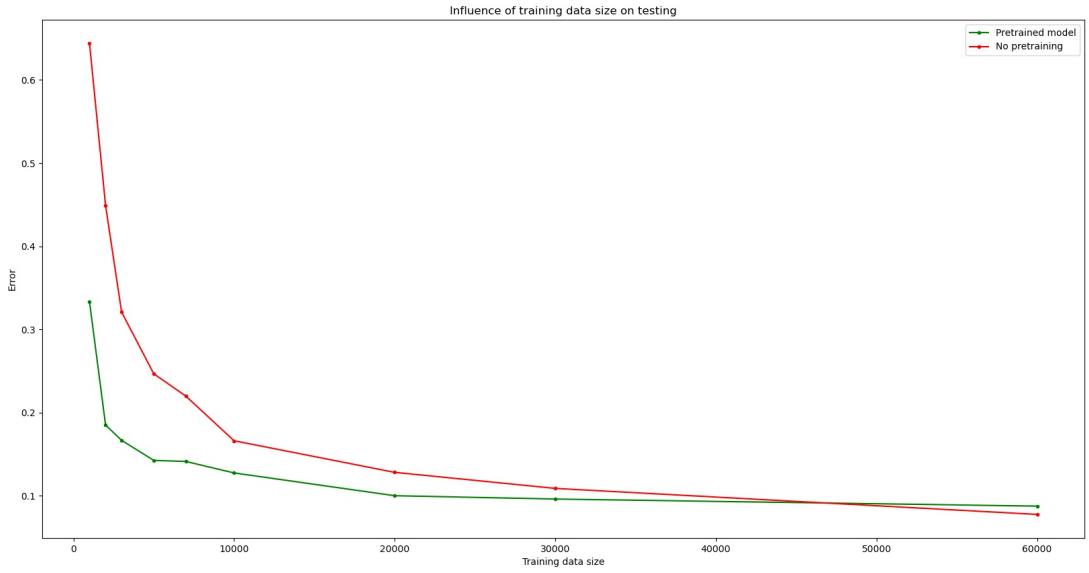

FIGURE 7 – Erreur de classification sur les données de test : Influence de la taille des données d'entraînement

Nous remarquons que l'erreur de classification décroît avec la taille des données d'entraînement, ce qui correspond aux attentes théoriques. En effet, un jeu de données de taille plus petit engendre un apprentissage moins exhaustif par le

modèle sur la distribution et les caractéristiques des données. La différence du taux d'erreur est très significative entre les premières valeurs des tailles données (1000, 2000 et 3000) pour ensuite décrire une pente de variations moins importante. Cela veut dire qu'à partir d'un certain moment, l'ajout de nouvelles données a un impact moins important sur la capacité des modèles à apprendre, relativement aux apports du même ordre de grandeur (augmentation de 100% en passant de 1000 à 2000 échantillons par exemple).

En ce qui concerne les comparaisons entre réseau pré-entraîné et non pré-entraîné, un réseau pré-entraîné a de meilleures performances (ce qui souligne son aptitude lors du pré-entraînement à s'approprier les caractéristiques des différentes classes sans même les connaître, ce qui serait visible aussi lors d'une tâche de *clustering* par exemple). Cela dit, à partir de 60000, la tendance s'inverse (pour l'entraînement et pour le test), en donnant un avantage au réseau non pré-entraîné. Cela est dû entre autres au fait que le pré-entraînement ait été effectué sur la totalité du jeux de données, ce qui donne un avantage au réseau pré-entraîné du côté du nombre d'échantillons avec lequel il est entré en contact, Mais cet avantage quand le deuxième réseau effectue son apprentissage sur la totalité du jeu de données également (même si cet avantage est relativement petit). Aussi, du moment où on part d'un jeu de données d'entraînement assez grand, un entraînement qui part de 0 peut exhiber une meilleure flexibilité quant à la mise à jour des paramètre du modèle, ce qui est probablement plus contraignant pour un modèle pré-entraîné.

3.3 Génération de données MNIST par une structure DNN

Nous avons également entraîné un réseau de type DNN, et nous avons visualisé les probabilités obtenues en sortie du modèle de quelques échantillons d'images en entrée. Nous illustrons cela dans les figures 8, 9, 10 et 11.

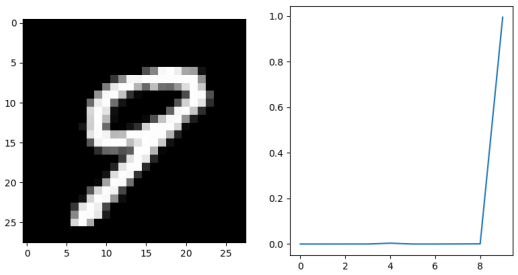

FIGURE 8 – Sample 1 probability distribution

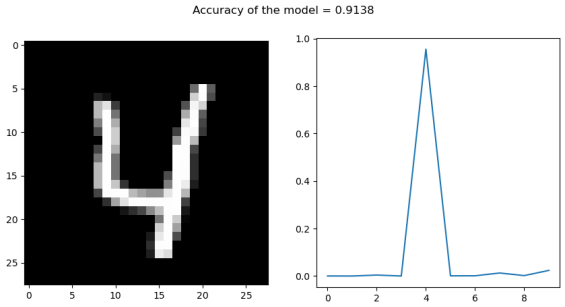

FIGURE 9 – Sample 2 probability distribution

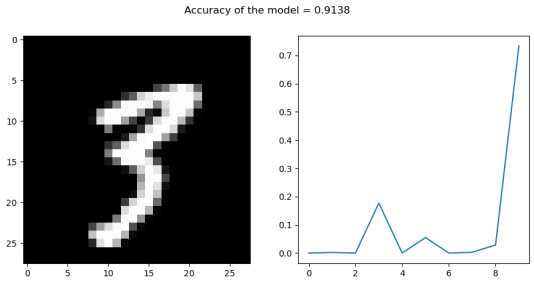

FIGURE 10 – Sample 3 probability distribution

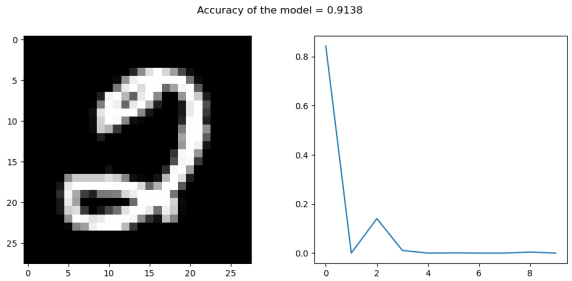

FIGURE 11 – Sample 4 probability distribution

Nous pouvons observer que dans les deux premiers cas, le modèle est très catégorique : il parvient à attribuer les probabilités de façon très distincte sur les classes. Pour les deux figures qui suivent (figures 10 et 11), même avec une

accuracy >91%, le modèle met un poids important sur des valeurs erronées, sans pour autant être catégorique (il y a bien un poids relativement important sur les classes réelles). Si on regarde les images données en entrée, on peut discerner les raisons qui ont mené le modèle à donner une fausse classification.

3.4 Modèle DNN optimal avec pré-entraînement

En prenant les valeurs qui ont donné les meilleures performances au cours des trois expériences de la sous-section 3.2, plus précisément :

— N_HIDDEN_LAYERS = 2

— N_HIDDEN_UNITS = 700

— DATA_SIZE = 60000

nous avons construit un modèle DNN "optimal". Nous avons effectué son pré-entraînement comme décrit plus haut, puis les deux phases d'entraînement et de tests sur 10 epochs. Nous avons donc généré le graphe du taux d'erreur de classification selon le nombre d'epochs dans la figure 12.

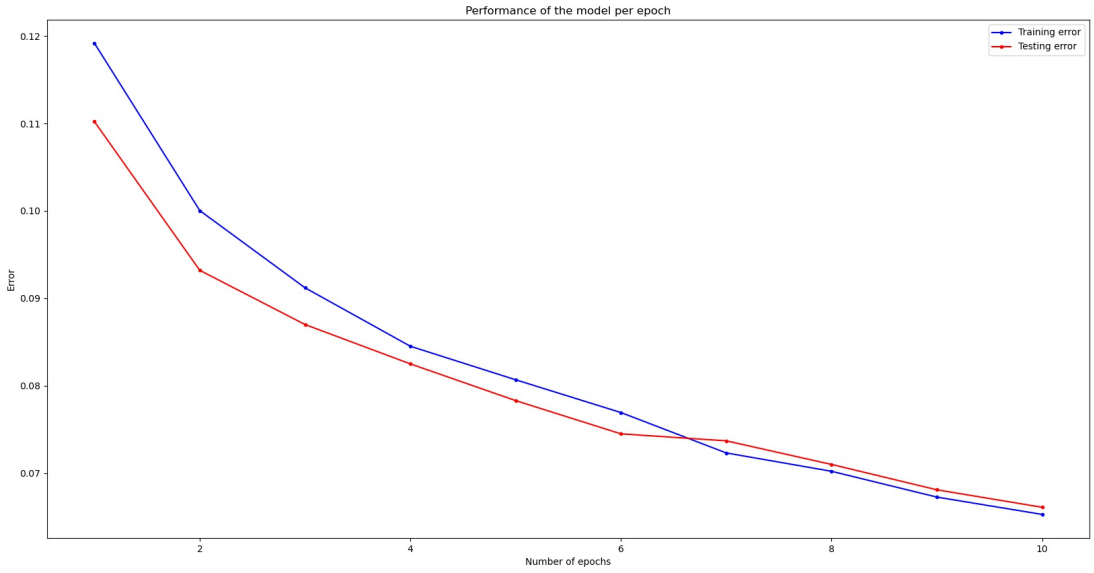

FIGURE 12 – Performances du modèle optimal

On observe ici clairement un comportement classique des réseaux de neurones qui montrent une aptitude plus grande à comprendre les caractéristiques des données s'ils passent par plusieurs phases d'entraînement. On observe aussi l'inversion de la position relative des courbes de taux d'erreur sur le jeu de données d'entraînement et de validation, ce qui peut être signe d'un début de sur-apprentissage si on fait plus d'itération d'entraînement.

4 Conclusion

Nous avons pu au travers de ce TP expérimenter sur les structures appelées *Deep Belief Network* qui sont construites à partir de *Restricted Boltzmann Machines*. Ces structures servent à modéliser une distribution de probabilité - celle des données - et permettent de faire du pré-entraînement sur les données du problème à traiter.

Nous avons réalisé des expériences faisant varier les paramètres de ces réseaux (nombre de couches cachées, taille des couches cachées) ou la taille des données d'entraînement en comparant les performances de ces DBNs (assimilés à des DNNs) à celles de réseaux de neurones aux caractéristiques identiques. Nous avons essentiellement décelé la capacité des structures introduites à compresser l'information des données au travers des couches cachées, ce qui permet notamment de contourner le problème du *vanishing gradient*, contrairement à un réseau classique. Cela dit, dans des conditions optimales, le pré-entraînement n'améliore visiblement pas les performances, voire provoque une légère détérioration, probablement dûe à une forme de rigidité des paramètres qui découlent de ce pré-entraînement.

Toutefois, cette représentation probabiliste peut avoir des avantages, notamment quand on n'a pas accès aux valeurs que nous souhaitons prédire (ni même de quel type d'objectif nous souhaitons atteindre). Elle peut être utile en proposant une sorte de *clustering* des données selon leurs features, ce qui permet d'avoir une meilleure analyse de la structure du jeu de données.