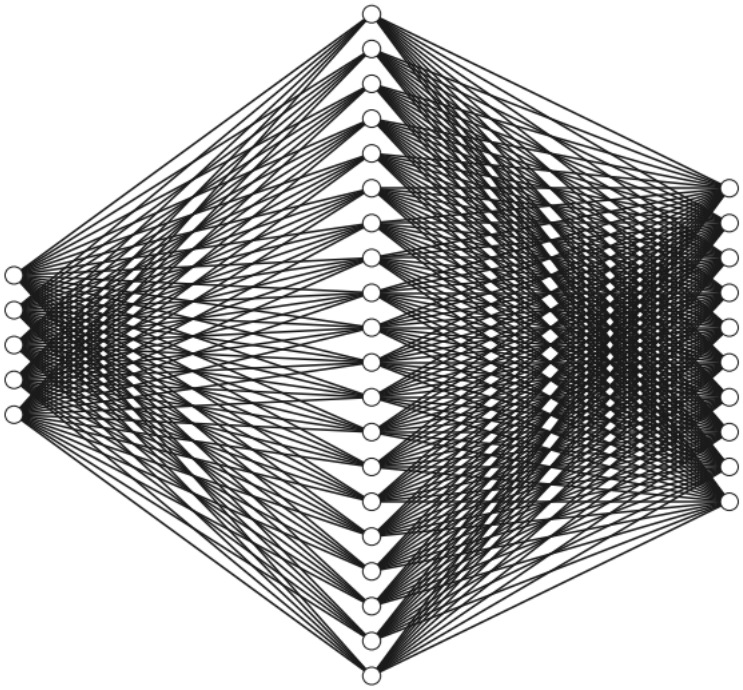
Télécom SudParis

Deep Learning

Compte rendu : Réseaux de Neuronnes



Nicolas Dufour

Janvier 2020

Table des matières

Table des figures

Liste des tableaux

1 Apprentissage non supervisé

Nous allons commencer par approcher le problème de l'apprentissage non supervisé.

1.1 Restricted Boltzmann Machines

1.1.1 Le modèle

Commençons par les machines de Boltzmann restreintes (RBM). Ce modèle va être la fondation pour des modèles plus complexes. On dispose donc de 2 vecteurs. Un vecteur V visible et un vecteur H caché. H est ce que l'on appelle une variable latente. L'utilisation d'une variable latente permet de modéliser une distribution complexe par des opérations simples. On a $V \in \{0,1\}^p$ et $H \in \{0,1\}^q$, où p le nombre de variables de V et q le nombre de variables de H. Notre objectif est de disposer d'un jeu de données $X \in \mathbb{R}^{n \text{x} p}$ et l'on souhaite trouver la distribution de probabilité la plus proche au sens de la Divergence de Kullback Leibler. Pour cela, on se donne le modèle suivant :

$$P_{RBM}(V,H) = \frac{e^{-E(V,H)}}{Z} \tag{1}$$

$$\text{avec } E(V,H) = -\sum_{i=1}^{p} a_i V_i - \sum_{j=1}^{q} b_j H_j - \sum_{i=1}^{p}\sum_{j=1}^{q} W_{ij} V_i H_j \tag{2}$$

avec $a \in \mathbb{R}^p$, $b \in \mathbb{R}^q$ et $W \in \mathbb{R}^{pxq}$. L'objectif de l'apprentissage va être d'estimer ces paramètres avec les données. On se retrouve avec le modèle suivant :

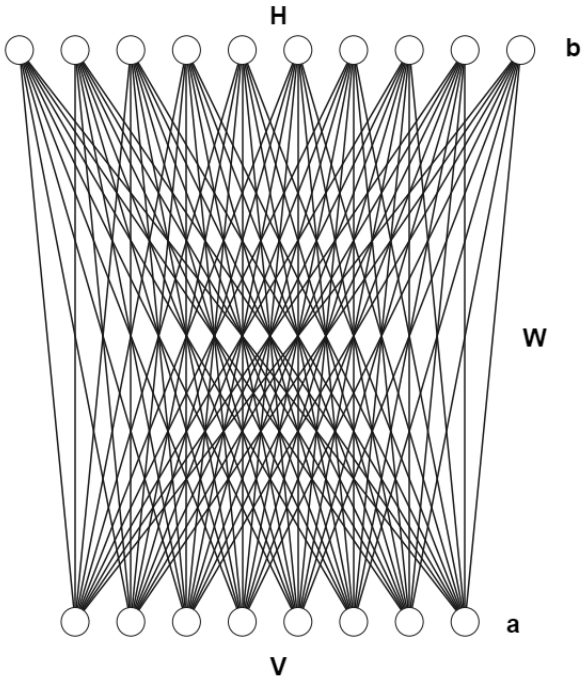


Figure 1 – *Structure d'un RBM*

On note σ la fonction sigmoide :

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3}$$

et

$$P(H|V) = \prod_{j=1}^{q} P(H_j|V) \tag{4}$$

avec

$$P(H_j = 1|V) = \sigma(b_j + \sum_{i=1}^{p} W_{ij} V_i) \tag{5}$$

On a de même

$$P(V|H) = \prod_{i=1}^{p} P(V_i|H) \tag{6}$$

avec

$$P(V_i = 1|H) = \sigma(a_i + \sum_{j=1}^{q} W_{ij} H_j) \tag{7}$$

1.1.2 Estimation des paramètres

Nous allons donc essayer d'apprendre a,b et W. Pour cela, nous allons chercher à maximiser la log-vraisemblance. Ainsi, on a :

$$log(p_{RBM}(V)) = log(\sum_{H} p(V,H)) = log(\sum_{h} e^{-E(V,H)}) - log(\sum_{V,H} e^{-E(V,H)}) \tag{8}$$

Or, si on cherche le maximum selon un paramètre θ, on se retrouve avec :

$$\nabla_\theta(log(p_{RBM}(V))) = -\sum_{H} p(H|V) \frac{\partial E(V,H)}{\partial \theta} + \sum_{H,V} p(H,V) \frac{\partial E(V,H)}{\partial \theta} \tag{9}$$

Or, ici, on cherche à estimer a, b et W. Faisons le calcul pour W_{ij}. On a :

$$\frac{\partial E(V,H)}{\partial W_{ij}} = -H_j V_i \tag{10}$$

On se retrouve au final avec

$$\sum_{H} p(H|V) \frac{\partial E(V,H)}{\partial W_{ij}} = P(H_j = 1|V) V_i \tag{11}$$

Cependant, l'autre terme reste incalculable. Le nombre de calculs est exponentiel avec la taille de notre RBM. Pour résoudre ce problème, nous allons avoir recours à un échantillonnage de Gibbs. En effet, on remarque que :

$$\sum_{H,V} p(H,V) \frac{\partial E(V,H)}{\partial W_{ij}} = \sum_{H,V} p(H,V) V_i H_j = \mathbb{E}(V_i H_j) \tag{12}$$

Pour le calculer, nous allons appliquer ce que l'on appelle le Contrastive Divergence 1.
On commence par initialiser V^0 avec x un échantillon. Puis, on tire H^0 selon $P(H|V^0)$ puis V^1 selon $P(V|H^0)$ et finalement H^1 selon $P(H|V^1)$ On prendra donc :

$$\nabla_{W_{ij}}(log(p_{RBM}(V))) \approx P(H_j = 1|V) V_i - V_i^1 P(H_j = 1|V^1) \tag{13}$$

Or, on ne peut pas trouver un maximum explicite. Nous allons donc procéder par remontée de gradient. Nous allons devoir vectoriser nos formules pour accélérer les calculs. On se retrouve avec pour X un jeu de données avec n échantillons :

$$dW = \nabla_W(log(p_{RBM}(X))) \approx X^T P(H_j = 1|X) - V^{1T} P(H|V^1) \tag{14}$$

$$da = \nabla_a(log(p_{RBM}(X))) \approx \sum_{n} (X - v^1) \tag{15}$$

$$db = \nabla_b(log(p_{RBM}(X))) \approx \sum_{n} (P(H|X) - P(H|V^1)) \tag{16}$$

On peut ensuite faire les mise à jour des paramètres par remontée du gradient. On se fixe ϵ un pas d'apprentissage. On initialise a_i^0, b_j^0 et W^0 puis on effectue les pas suivants. $\forall l \in \mathbb{N}$, on recalcule les gradients par contrastive divergence comme expliqué ci dessus puis, nous mettons à jour les paramètres comme suit :

$$W^{l+1} = W^l + \epsilon dW \tag{17}$$

$$a^{l+1} = a^l + \epsilon da \tag{18}$$

$$b^{l+1} = b^l + \epsilon db \tag{19}$$

Pour estimer les performances de notre RBM, on peut faire 2 choses : Se donner un jeu de données X, calculer H puis à partir de H recalculer V, puis on pourra calculer l'erreur quadratique entre X et V. Ensuite pour vérifier que l'on n'a pas sur-appris, on peut essayer de générer des images à partir d'une initialisation aléatoire et à l'aide d'un échantillonneur de Gibbs obtenir des données similaires au jeu de données initial. Par contre si les données sont identiques aux données initiales, on a surappris.

1.1.3 Expériences

Dans la pratique, nous allons utiliser la technique du mini-batch(Voir Annexe). On va travailler sur le jeu de données Binary Alpha Digits. On prendra donc $p = 320$. Par une recherche par grid-search en essayant de minimiser l'erreur quadratique, on se rend compte que la meilleure dimension pour la variable latente est $q = 500$. On initialisera notre RBM avec $\forall i \in [|1, p|], \forall j \in [|1, q|]$ $a_i = 0$, $b_j = 0$ et $W_{ij} \hookrightarrow \mathcal{N}(0, 0.1)$. On prendra de plus un pas d'apprentissage $\epsilon = 0.1$ et une taille de mini-batch de 32. On prendra 100 epochs pour notre étude. On entraîne d'abord notre RBM sur un seul caractère. On le fait pour tout les caractères de notre dataset et ensuite, on régénère 10 caractères à partir d'un V aléatoire. On obtient le graphique suivant :

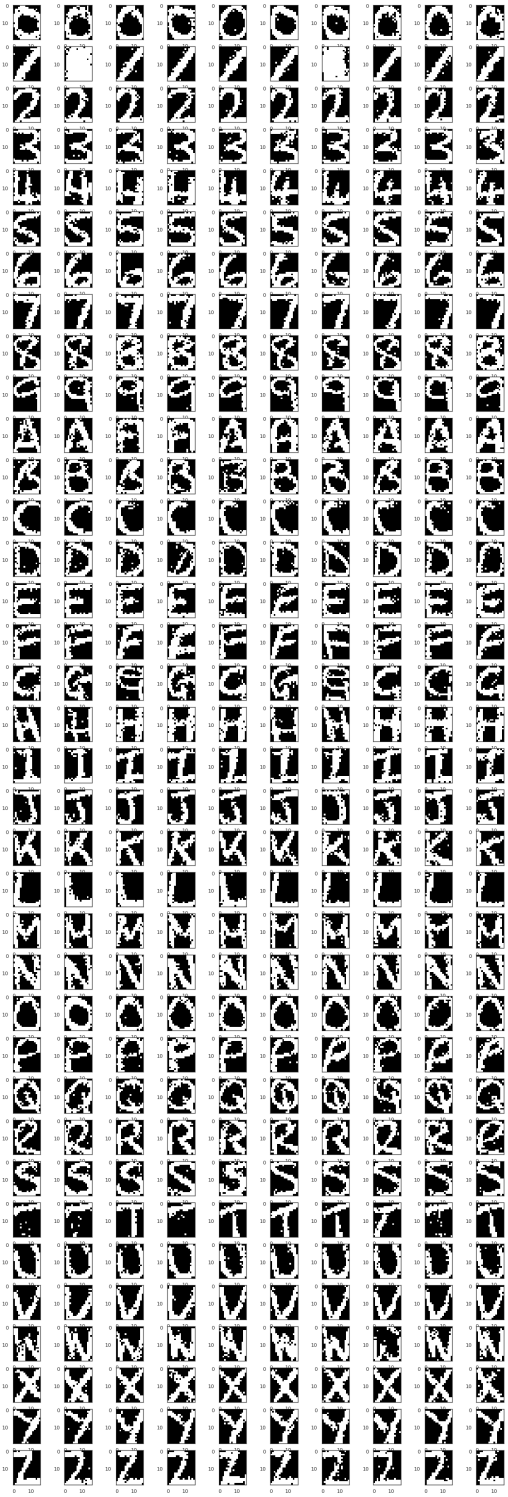


Figure 2 – *Caractères générés par le RBM suite à l'apprentissage du caractère.*

On se rend compte que notre RBM a bien appris les différents lettres et chiffres. On peut cependant observer un peu d'overfitting pour certains caractères comme le 1. Cependant, cela est dû à un manque de diversité dans les données originales. Pour le reste des caractères, on remarque que la plupart sont bien reconstruit et varient suffisamment du dataset original pour que l'on n'ait pas d'overfitting.
Cependant, notre objectif à terme est de reconnaître les différents caractères. Donc, nous devons essayer de faire apprendre à notre modèle les différents caractères. Nous allons donc choisir aléatoirement n caractères puis n+1 et ainsi de suite puis essayer de recréer des données et nous analyserons les résultats.

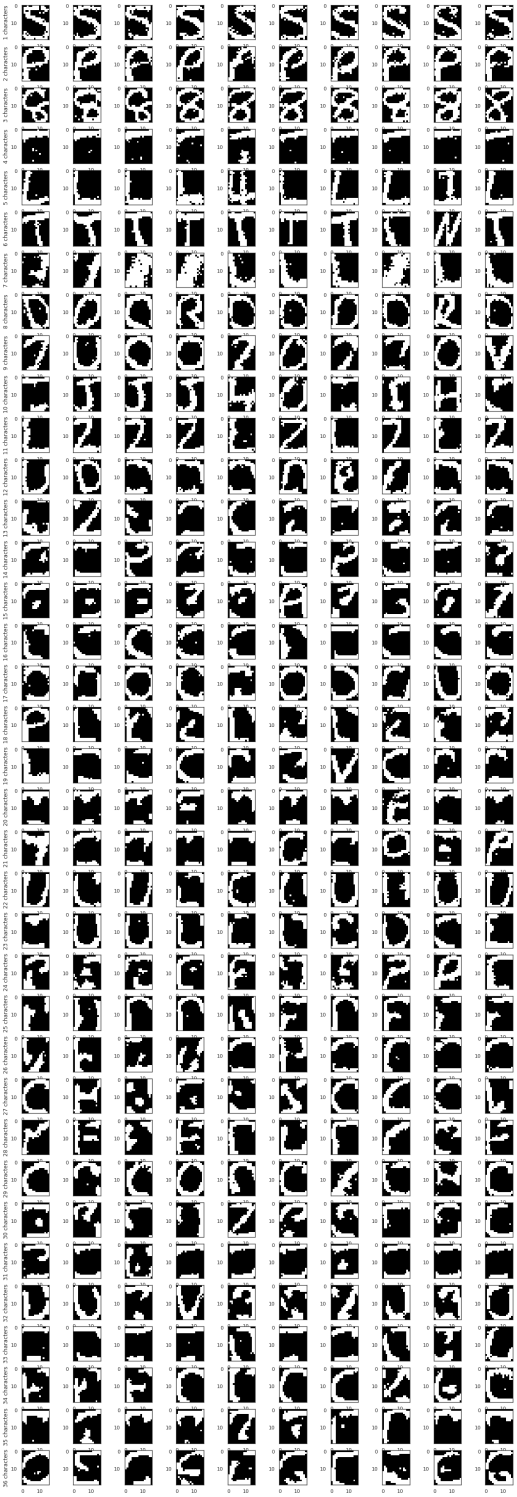


Figure 3 – *Caractères générés par le RBM suite à l'apprentissage d'un certain nombre de caractères.*

On voit qu' à partir d'un certain nombre de classes données au RBM, il ne sait plus régénérer les différents

caractères et régénère des features commun sans que l'on reconnaisse aucune lettre ou chiffre.
Le RBM est donc limité dans notre construction d'un classifier des différents caractères. Nous allons donc proposer un modèle plus complexe, le DBN

1.2 Deep Belief Network

1.2.1 Le modèle

Nous allons maintenant considérer un modèle plus complexe. C'est aussi un modèle non supervisé. Nous allons prendre plusieurs couches $H^k \in \{0;1\}^{q_k}$ avec q_k la taille de la couche H^k avec $k \in [0,d]$ où d le nombre de couches latentes. Entre chaque couche (H^{k-1}, H^k) nous associons une matrice de poids W^k et un biais b^k. On notera que H^0 correspond à notre couche visible. On associe de plus la probabilité de passage suivante :

$$P(H^k|H^{k-1}) = \prod_{j=1}^{q} P(H_j^k|H^{k-1}) \tag{20}$$

Avec, $\forall j$

$$P(H_j^k = 1|H^{k-1}) = \sigma(b_j^k + \sum_{j=1}^{q} W_{ij}^k H_i^{k-1}) \tag{21}$$

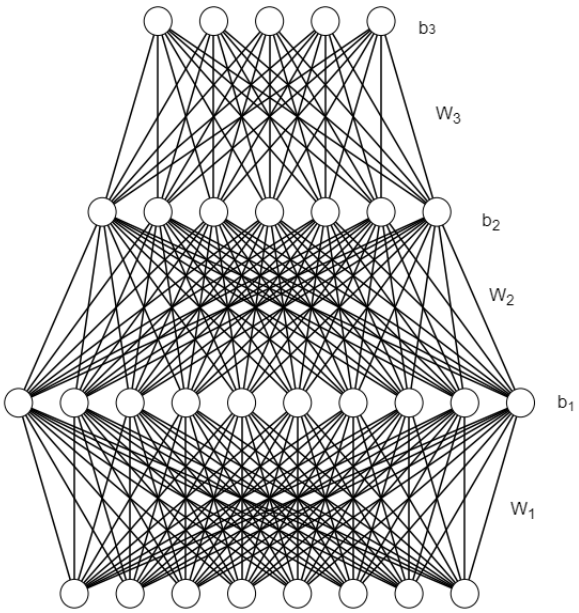


Figure 4 – *Structure d'un DBN*

1.2.2 Entraînement

Pour entraîner le modèle, nous allons considérer que chaque paire (H^{k-1}, H^k) est un RBM de paramètres (W^k, b^{k-1}, b^k). Nous entraînerons chaque RBM par Contrastive divergence 1 comme vu dans la première partie. Nous garderons ensuite (W^k, b^{k-1}, b^k). La procédure commence par entraîner H^d puis H^{d-1} et ainsi de suite.

1.2.3 Expériences

Nous allons refaire les mêmes expériences que avec le RBM. Nous allons considérer le modèle composé de 3 couches latentes de taille 400. Chaque paramètre sera initialisé de la façon suivante : avec $\forall k \in [|1,d|]$, $\forall j \in [|1,q_k|]$ $b_j^k = 0$ et $W_{ij}^k \hookrightarrow \mathcal{N}(0,0.1)$. On prendra de plus un pas d'apprentissage $\epsilon = 0.1$ et une taille de mini-batch de 32. On prendra 100 epochs pour notre étude. On entraîne d'abord notre DBN sur un seul

caractère. On le fait pour tout les caractères de notre dataset et ensuite, on régénère 10 caractères à partir d'un H^{d-1} aléatoire. Puis, on calcule les probabilités descendantes de chaque RBM jusqu'à arriver à H^0. On génère ainsi les caractères. On obtient le graphique suivant :

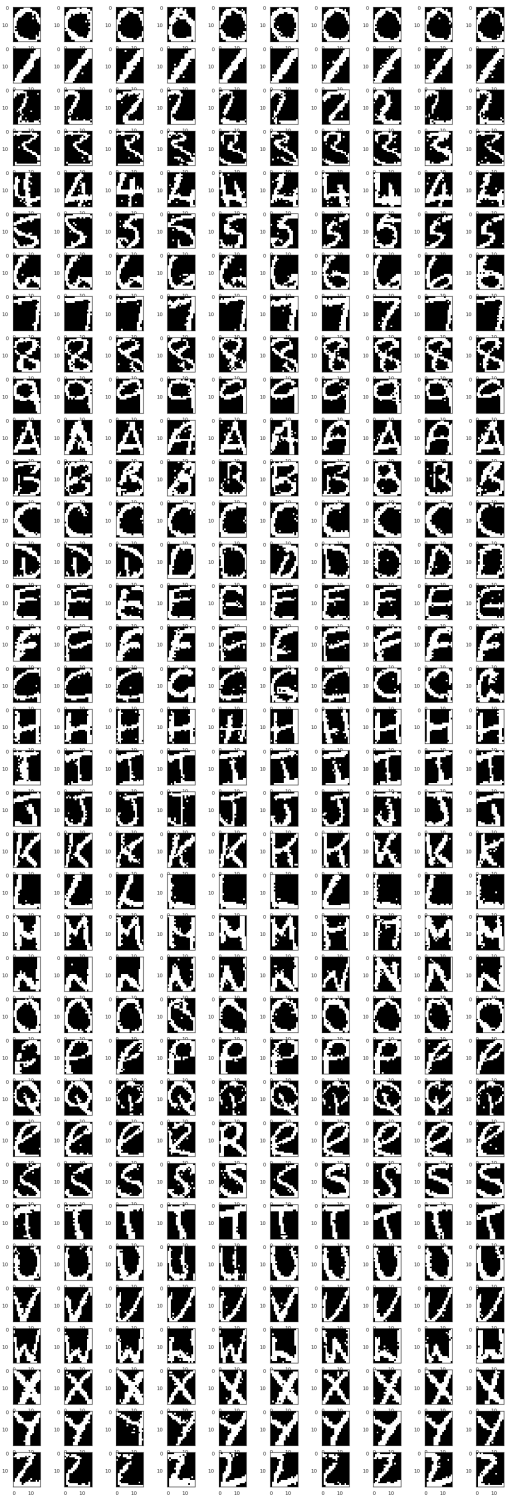


Figure 5 – *Caractères générés par le DBN suite à l'apprentissage du caractère.*

On se rend compte que comme pour le RBM, notre modèle n'a aucune difficulté à régénérer les données. Notre modèle a donc bien appris le modèle de données pour 1 caractère individuel.
Comme on veut entraîner notre DBN pour plusieurs caractères, on va progressivement apprendre un paramètre de plus à chaque itération. On obtient le résultat suivant :

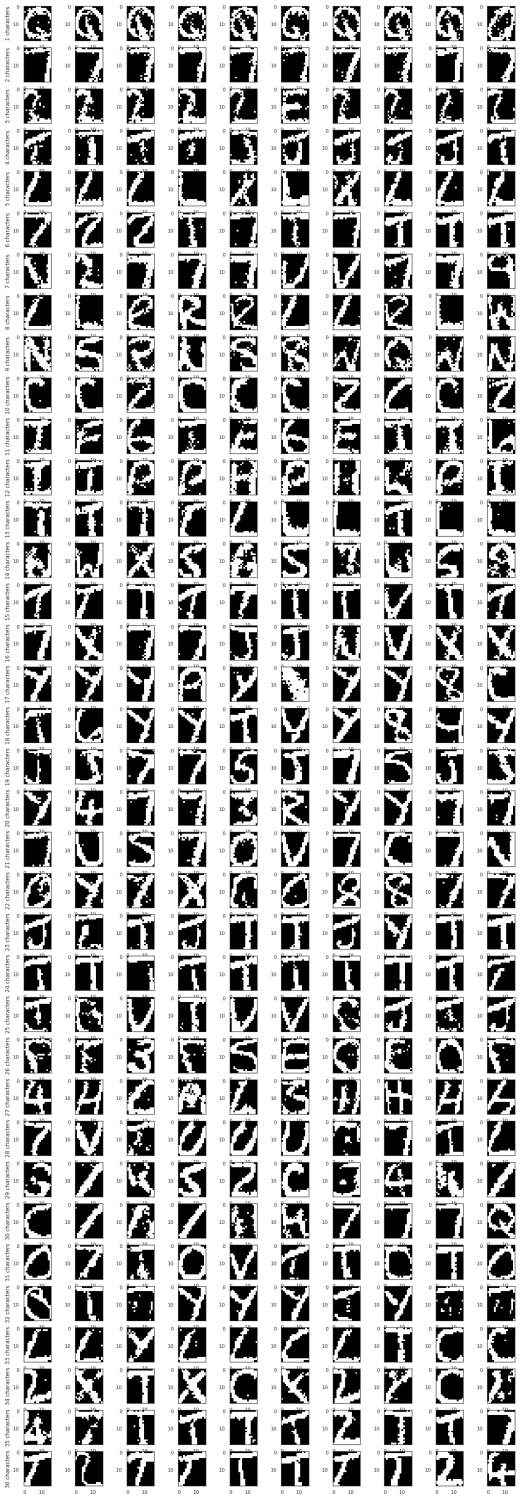


Figure 6 – *Caractères générés par le DBN suite à l'apprentissage d'un certain nombre de caractères. Variance = 0.01*

On se rend compte que contrairement au RBM, le DBN arrive à régénérer des caractères même quand l'on apprend de toutes les données. Cela nous montre que le DBN a appris un modèle plus riche. On se rend par contre compte que certains caractères sont plus souvent régénérés comme le T. Cela est peut etre dû au fait que le T est le caractère qui partage la plupart des features avec les autres caractères. Ce serait un peu le centroide de notre apprentissage de la représentation des données. Pour essayer d'avoir plus de diversité dans notre génération nous allons augmenter la variance sur l'initialisation aléatoire de la couche visible du RBM. Ainsi, à la place d'une variance de 0.01, on prend 1. On obtient le résultat suivant :

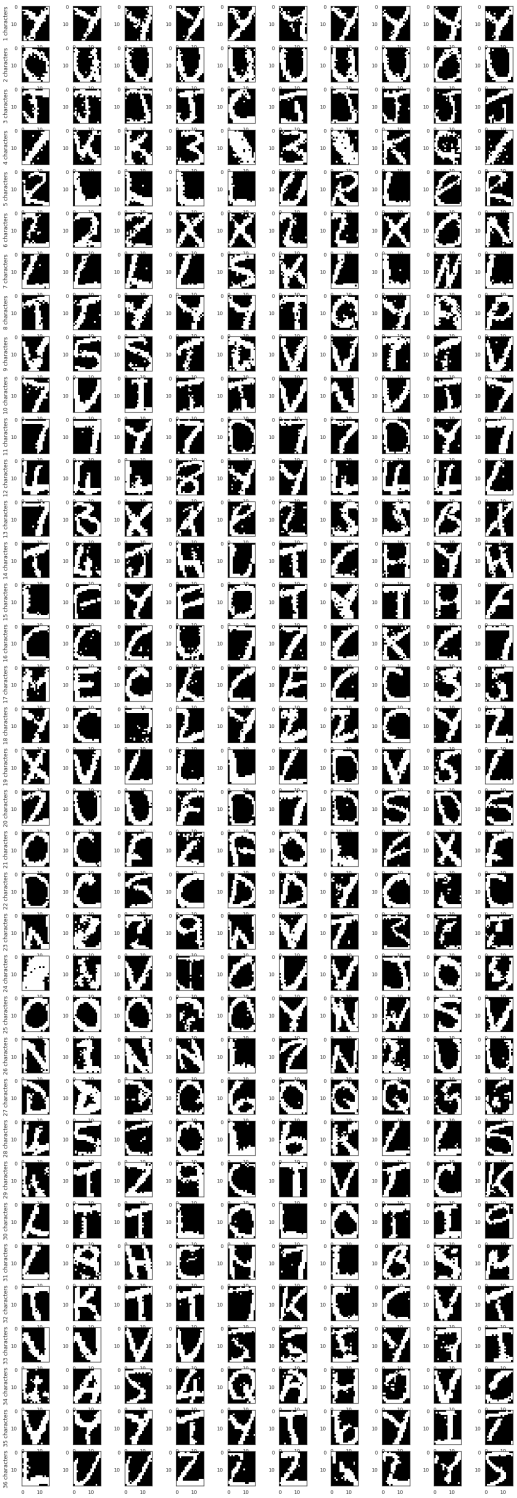


Figure 7 – *Caractères générés par le DBN suite à l'apprentissage d'un certain nombre de caractères. Variance = 1*

On voit bien que l'on a plus de diversité de caractères dans notre génération. On en conclu que notre modèle n'a pas seulement appris le T mais a aussi bien appris toute la diversité des données.

2 Apprentissage supervisé : Le réseau de neurones

Dans le cas de l'apprentissage supervisé, la différence est que l'on dispose d'une labelisation de nos données.

2.1 Théorie

On se retrouve avec une architecture assez similaire au DBN. On a d couches latentes (ou cachées) $H_k \forall k \in [1, d]$, une couche d'entrée V (où H_0) et une couche de sortie S (où H_{d+1}. De plus $\forall k \in [1, d+1]$, on associe à la couche k une matrice de poids W^k et un vecteur de biais b^k. De chaque couche, on calcule la couche suivante à partir de la couche antérieure. $\forall k \in [1, d]$, on calcule la transition de la façon suivante :

$$H^k = \sigma(< H^{k-1}, W^k > + b^k). \tag{22}$$

Pour la dernière couche, c'est un peu particulier. On introduit la fonction softmax :

$$\begin{aligned} &\forall x \in \mathbb{R}^p, \forall i \in [1, p], \\ &Sofmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}} \end{aligned} \tag{23}$$

Ainsi pour notre dernière couche, on a :

$$H^{d+1} = softmax(< H^d, W^{d+1} > + b^{d+1})) \tag{24}$$

2.2 Entraînement

L'entraînement est effectué par une descente de gradient. Pour calculer les gradients, on effectue ce que l'on appelle une backpropagation. Il est primordial de bien vectoriser notre problème pour accélérer le calcul des gradients. On notera Y le vecteur des labels et $\hat{Y}$ les labels prédits. On utilisera une perte de type cross-entropie. Puis, on calculera les gradients de la façon suivante :

$$\begin{aligned} C &= \hat{Y} - Y \\ dW^{d+1} &=< H^{dT}, C > \\ db^{d+1} &= \sum_{x \in X} C_x (\text{X l'ensemble des données}) \\ &\text{Pour k entre d et 1} \\ C &= (H^h \otimes (1 - H^k)) \otimes < C, W^{kT} > \\ dW^k &=< H^{k-1T}, C > \\ db^k &= \sum_{x \in X} C_x (\text{X l'ensemble des données}) \end{aligned} \tag{25}$$

Une fois les gradients calculés, on peut faire notre descente de gradient et l'on itère sur un nombre donné d'itérations.

2.3 Initialisation

2.3.1 Initialisation aléatoire

Une approche classique est d'initialiser aléatoirement nos poids. On prend souvent une variance de 0.01 et une espérance de 0. Pour les biais, on prends des valeurs nulles

2.3.2 Initialisation DBN

On peut aussi essayer d'entraîner un DBN sur notre dataset et prendre les poids et biais des couches latentes. Ceci devrait accélérer notre entraînement car on initialise avec une distribution qui est supposée représenter nos données.

2.4 Expériences

Nous allons effectuer une série d'expériences sur le dataset MNIST qui regroupe des chiffres manuscrits. Nous allons dans un premier temps ramener notre dataset sur des pixels valant 1 ou 0 pour faciliter l'entraînement. On utilisera une taille de MiniBatch de 128 et un pas d'apprentissage de 0.1. Pour vérifier que l'on a bien appris on peut observer la courbe de la perte sur le jeu d'apprentissage et sur le jeu de test comme la courbe suivante :

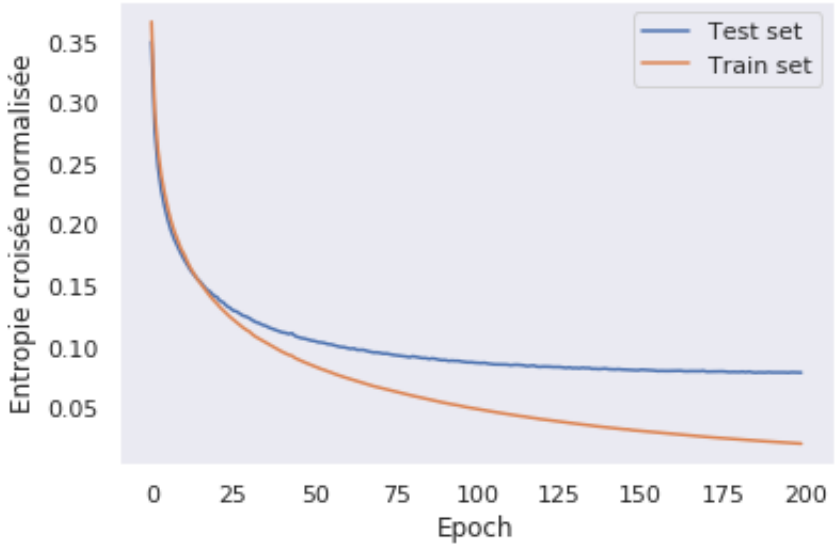


Figure 8 – *Exemple de perte pendant l'entrainement*

Cette courbe nous permet de vérifier que l'apprentissage se passe bien. Il n'y a pas d'overfitting car les courbes de train et de test sont assez proches. Maintenant, nous allons tester certaines configurations à la fois sur une initialisation aléatoire et sur une initialisation avec des poids pré-entrainés avec un DBN. Nous allons commencer par vérifier quel est l'effet du nombre de couches :

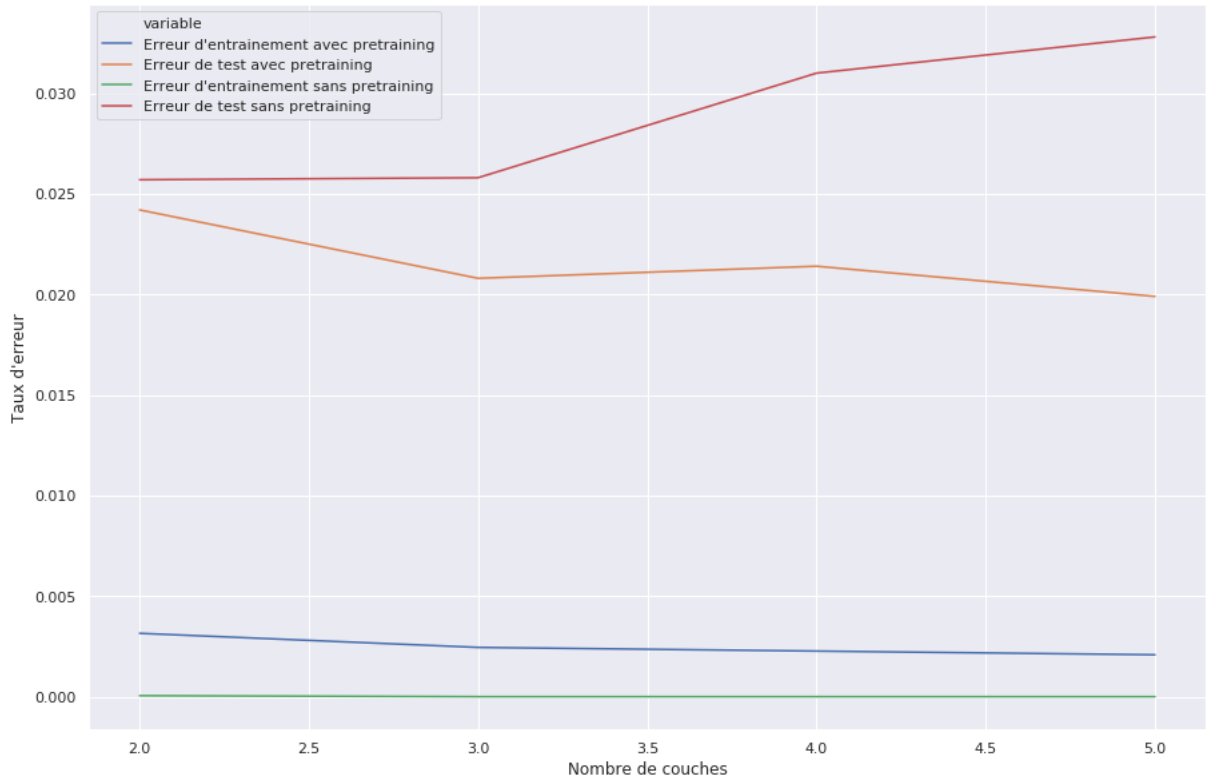


Figure 9 – *Taux d'erreur en fonction du nombre de couches pour un réseau avec des couches latentes de taille 200*

On voit que le pré-apprentissage permet d'avoir de meilleurs résultats en ajoutant plus de couches. Sans pre-apprentissage, on se rend compte que le taux d'erreur augmente avec le nombre de couches, signe d'over-fitting. Intéressant aussi, on peut constater que le train set a de meilleures performances sans pre-training. Or, le plus intéressant sont les performances du test-set car permettent de mesurer la capacité de généralisation de notre modèle.
On peut aussi s'intéresser au nombre de neurones par couche. On prendra un modèle à 2 couches latentes.

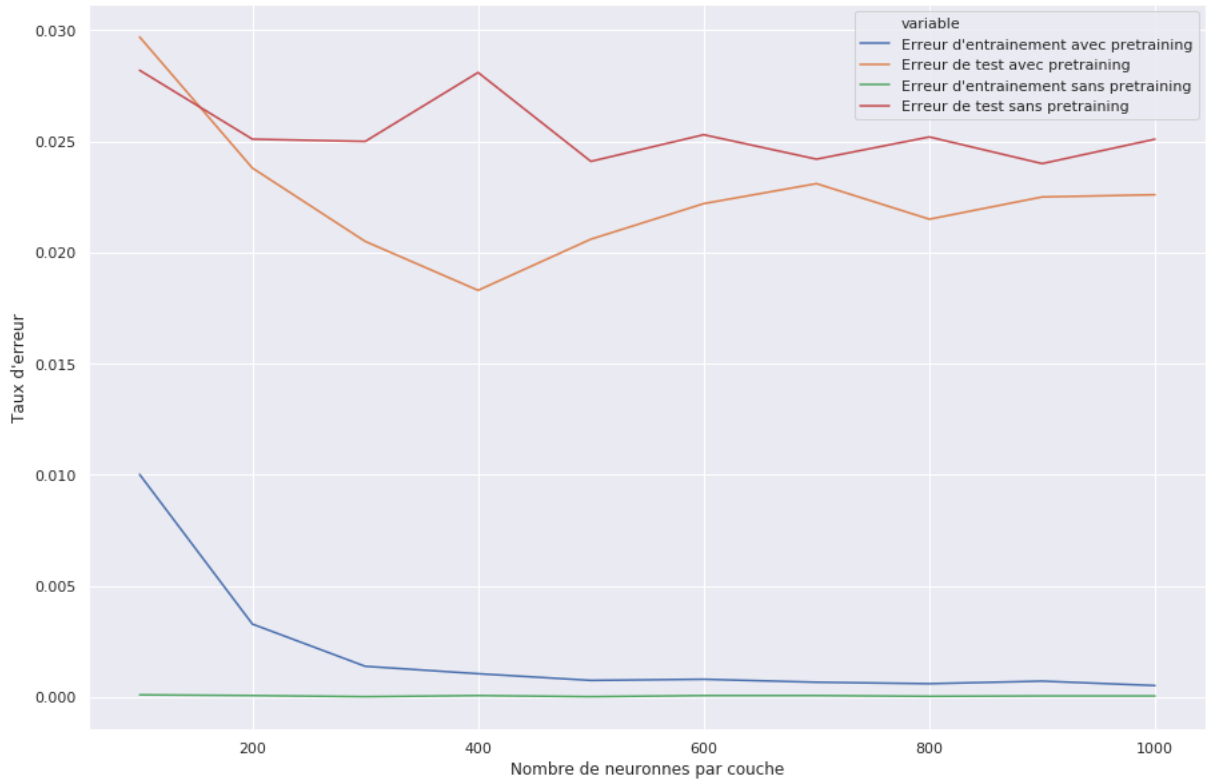


Figure ***10*** – *Taux d'erreur en fonction du nombre de neurones pour un réseau avec 2 couches latente*

On se rend compte qu'ici, on obtient les meilleurs performances pour le test set avec un pre-apprentissage avec 400 neurones. Pour cette configuration, c'est donc le nombre de neurones idéal. Au delà, on observe une augmentation du taux d'erreur sur le test set (et donc plus d'overfitting).
Pour finir, étudions l'impact du nombre de données. On prendra un réseau de neurones avec 2 couches de 200 neurones chacun.

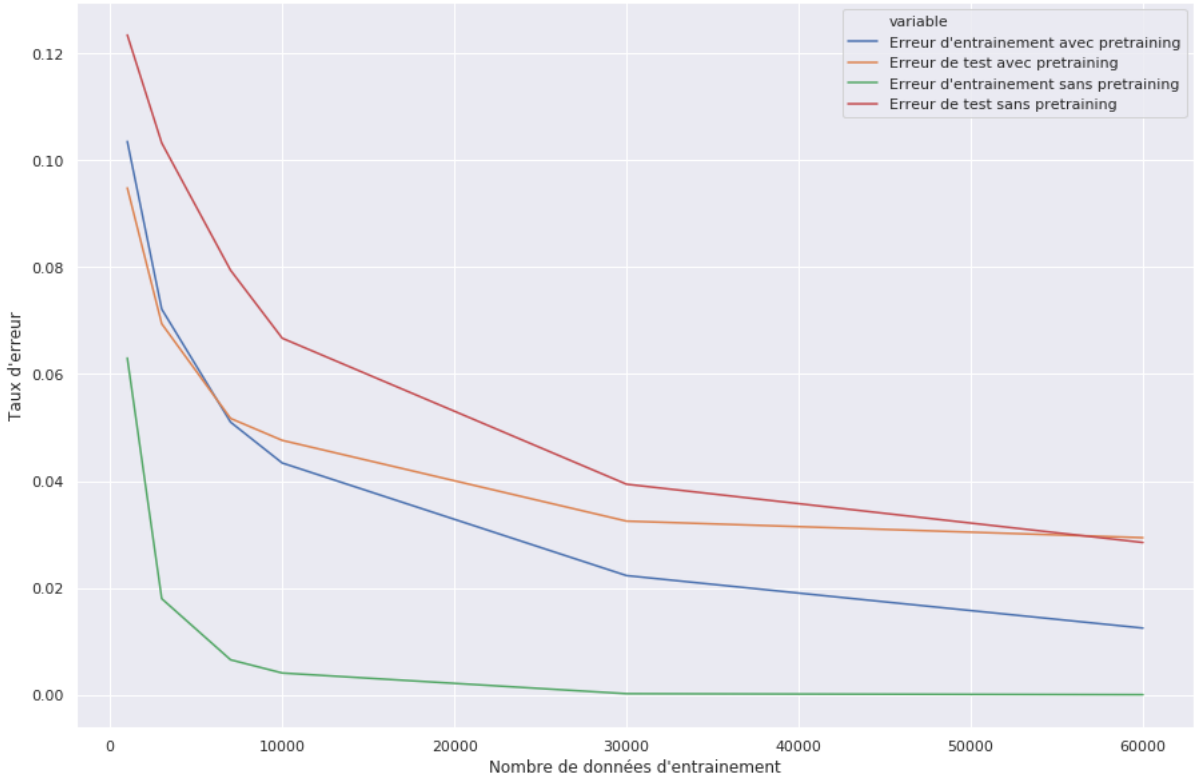


Figure 11 – *Taux d'erreur en fonction de la quantité de données*

On peut voir ici que plus on a de données, meilleures sont nos performances sur le test set. Ceci montre que le pouvoir du deep learning réside dans les données : Plus on a de données, meilleures seront les performances.

2.5 Recherche de la meilleure configuration

Nous allons essayer de chercher la configuration avec les meilleures performances sur le test-set. Nous testerons chaque configuration en initialisation aléatoire et avec du pré-training. Dans la littérature, on peut voire que les architectures à meilleures performances sont de la forme bottle-neck (Une décroissance dans le nombre de neurones à travers les couches), nous allons donc tester ce genre de configuration. Ensuite, nous chercherons d'autres configurations selon notre étude précédente.

Configuration Latente	Perte avec pre-training	Perte sans pre-training
1000-500	0.026	0.0232
1500-100-500	0.021	0.022
2000-1500-1000-500	0.020	0.022
2500-2000-1500-1000-500	0.018	0.022
400-400-400-400-400	0.019	0.030
400-400-400-400	0.0168	0.0280
800-800	0.023	0.0246

Table 1 – *Performances sur le test set de certaines configurations*

On constate que la technique "bottle-neck" de la littérature n'a pas les meilleures performance. Cela est du au fait qu'ils utilisent des techniques d'augmentation de données et des stratégies de pas d'apprentissage plus optimales que les nôtres. Notre meilleure configuration est 4 couches latente de 400 neurones avec pretraining. On a une erreur de 1.6%.

3 Annexe

3.1 Descente de gradient

La descente de gradient consiste à rechercher un minimum en allant dans le sens inverse du gradient de la fonction à minimiser. On effectue plusieures itérations jusqu'à converger vers un minimum local en se fixant un pas d'apprentissage de ϵ. Pour un paramètre θ, on a à l'itération k :

$$\theta^{k+1} = \theta^k - \epsilon d\theta \tag{26}$$

Le choix du pas d'apprentissage détermine la rapidité de la convergence ainsi que la précision de notre minimum. Certaines stratégies peuvent être aussi mises en place pour éviter les minimas locaux.

3.2 Mini-Batch

La méthode du mini-batch consiste à se donner une taille de batch puis de tirer des sous-ensemble de notre jeu d'apprentissage avec cette taille puis apprendre sur ces sous-ensembles. Pour la descente du gradient, l'algorithme est le suivant :

Algorithm 1 Descente de gradient par mini-batch

X = Jeu d'apprentissage
Y = Labels si supervisé
t = taille du mini batch
n = taille du data-set
ϵ = Pas d'apprentissage
θ = paramètre à apprendre
epochs = nombre d'itération du gradient
for epoch in [1,epochs] **do**
  permutations = Permutations de taille n
  for i in range(n//t) **do**
    $batch_X = X[permutations[i * t : min((i + 1) * t, n - 1)]]$
    $batch_Y = Y[permutations[i * t : min((i + 1) * t, n - 1)]]$
    dθ = calcule du gradient selon $batch_X$ et $batch_Y$
    $\theta = \theta - \epsilon d\theta$
  end for
end for

On utilise le mini-batch car parfois, les jeux de données peuvent être trop gros pour entrer en mémoire. En plus, le mini-batch peut permettre de régulariser l'apprentissage car il apporte un coté stochastique dans le choix des données.