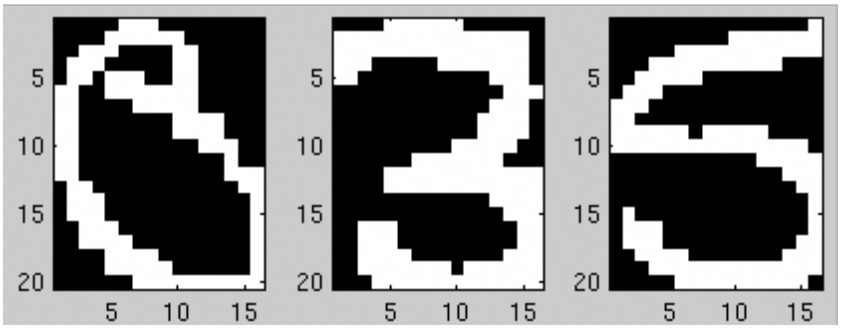
# Rapport Tp Deep neural networks

## I-Données

On commence par télécharger les données depuis les sites disponibles sur le sujet du tp.
Quelques exemples d'images dans la base de données sont donnés ci dessous :




Après avoir chargé les données, on code la fonction lire_alpha_digits qui prend en argument un caractère ou une chaine de caractères et les labels. Cette fonction retourne une matrice contenant sur chaque ligne une image du caractère qu'on veut sélectionner, chaque colonne représentant un pixel. Par exemple **lire_alpha_digit( 'C', dat.dat,dat.classlabels)** est censé chercher toutes les images correspondant au label 'C' dans la base de données et les retourner dans une matrice contenant une image par ligne.

## II- Fonctions élémentaires et entrainement

Pour mettre en œuvre un DBN on a besoin de quelques fonctions qui permettent d'effectuer les opérations élémentaires dont on a besoin, comme calculer les sorties sachant les entrées, les entrées sachant les sorties, les sorties de tout le réseau, les initialisations, etc.

***Partie RBM :***
**entree_sortie_RBM :** Nous permet de calculer la probabilité des sorties sachant les entrées pour un RBM.

$$p(h|v) = \prod_{j=1}^{q} p(h_j|v) \qquad p(h_j = 1|v) = sigmoid(b_j + \sum_i v_i w_{ij})$$

**sortie_entree_RBM** : Nous permet de calculer la probabilité des entrées sachant les sorties pour un RBM. Dans le cadre d'un simple RBM cette fonction nous permet d'avoir la loi des entrées. En tirant selon la loi jointe qu'on approxime on arrive donc à reproduire ce que le RBM a appris.

$$p(v|h) = \prod_i p(v_i|h) \qquad p(v_i = 1|h) = sigmoide(a_i + \sum_j h_j w_{ij})$$

**init_RBM** : Prend en argument les tailles d'entré et les tailles de sortie du RBM. Grâce a ces tailles on initialise selon une loi normal, à l'aide de la fonction randn( ) les paramètres w, a et b. w étant de taille [tailleEntree, tailleSortie] a de taille tailleEntree et b de taille tailleSortie.

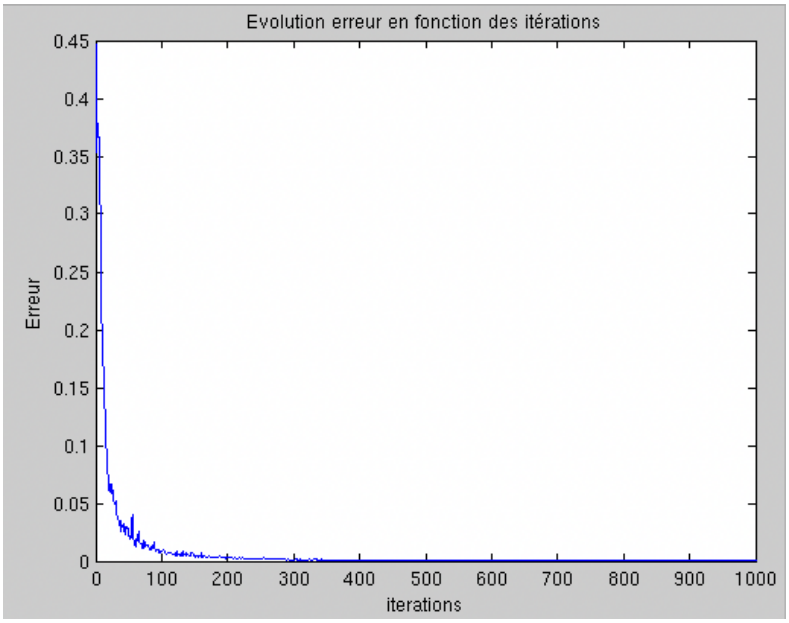
**train_RBM** :L'entrainement du RBM consiste a mettre à jour les paramètres du RBM, c'est à dire w, a et b. Cette mise à jour doit être effectué de façons à avoir les paramètres qui maximisent la vraisemblance. Pour cela on utilise la monté de gradient avec minibatch, le principe est simple, on commence par diviser notre base de données en plusieurs sous bases de taille batchSize ensuite on met à jour les paramètres batch par batch de la façon suivante :

RBM.w=RBM.w+((learningRate*dw)/batchSize)
RBM.a=RBM.a+((learningRate*da)/batchSize)
RBM.b=RBM.b+((learningRate*db)/batchSize)

ici dw étant la dérivé partiel de la log-vraisemblance par rapport à w. Pareil pour da et db.
Encore faut il calculer ou approximer dw, da et db. On obtient les résultats intermédiaires suivants :

$$\frac{\partial log p_v(x)}{\partial w_{ij}} = p(h_j = 1|x)x_i - \sum_{v_i h_j} p(v_i, h_j)v_i h_j$$
$$\frac{\partial log p_v(x)}{\partial a_i} = x_i - \sum_{v_i h_j} p(v_i, h_j)v_i$$
$$\frac{\partial log p_v(x)}{\partial b_j} = p(h_j = 1|x) - \sum_{v_i h_j} p(v_i, h_j)h_j$$

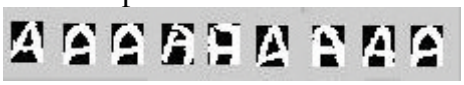
Malheureusement dans les trois expressions ci-dessus on sait calculer les termes positifs avec les fonctions précédemment décrites, mais pas les termes négatifs. Il faut donc approximer les termes négatifs. Ceux ci sont approximable si on fait un tirage selon la loi du couple (entrée, sortie), mais on ne sait pas faire cela non plus. Par contre on peut calculer la loi des entrées sachant les sorties et vice versa. Grâce à cela on peut utiliser l'échantillonneur de gibbs pour approximer la loi du couple.
Concrètement on commence par calculer la loi des sorties sachant les entrées (hid0) à l'aide de entree_sortie_RBM, puis on fait un tirage selon cette loi(sample_hid0). On utilise ensuite ce tirage pour approximer la loi des entrées sachant les sorties(vis1). On refait un tirage selon cette nouvelle loi(sample_vis1), qui va nous permettre d'approximer la loi des sorties sachant les entrées approximés(hid1). À l'aide de vis0, hid0, vis1, sample_vis1,hid1 on arrive à donner une valeur approximative de dw, da et db, ce qui nous permet de faire notre monté de gradient. Plus de détails sont disponibles sur le code fournit en pièce jointe.
La monté de gradient étant un algorithme itératif, où l'on s'approche de plus en plus du maximum, on choisit nb_iter suffisamment grand pour être sûr de converger. Pour être sûr que notre RBM apprend correctement, on calcul l'erreur sur chaque itération. Celle ci est sensée diminuer avec les itérations.




En effet ici l'erreur décroit exponentiellement en fonction des itérations. L'entrainement est donc bien effectué.

**generer_image_RBM:** Prend en argument le RBM entrainé, le nombre d'images qu'on veut générer, le nombre d'itérations de gibbs et la taille des images d'entrées.
Grâce au RBM déjà entrainé, de entree_sortie_RBM et sortie_Entree_RBM on fait des tirages de gibbs nbiterGibbs fois. On choisit un grand nombre d'itérations de Gibbs pour être sûr de converger.

Avec les paramètres suivants et 'A' comme caractère en entré on obtient les images ci dessous :



taille_out=5000
nbIter=1000
learningRate=0.1
batchSize=100
nbImages=20
nbIterGibbs=3000

Par contre le RBM étant une architecture assez simple, on ne peut pas modéliser des loi très complexe. Preuve à l'appuie avec un seul caractère on arrive à reproduire, mais si on donne plusieurs caractères on voit bien que celui-ci n'arrive à les reproduire (voir résultats ci dessous pour la chaine 'A158').

C'est pour cela qu'on complexifie le modèle dans la suite en utilisant des DBN.

**calcul_softmax** : Prend en argument un RBM et les données d'entrée et calcule la sortie du RBM à l'aide de la fonction softmax et non de la fonction sigmoid comme pour entree_sortie_RBM.

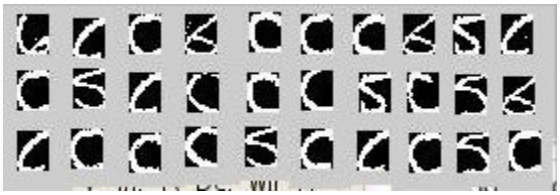
**Entree_sortie-reseau** : Cette fonction calcule les sorties des différentes couches du DBN. En supposant qu'on a N couches, alors on calcule les sorties des n-1 premières couches à l'aide de

entree_sortie_RBM ( vu qu'on peut pas approximer p(x) on opte pour l'approche variationnel qui consiste à considérer le DBN comme un empilement de RBM)  et la n-iéme couche avec calcul_softmax vu que c'est la sortie de notre réseau.

**init_DBN** : Vu qu'on voit notre DBN comme un empilement de RBM, alors on initialise notre DBN en utilisant init_RBM. Initialiser le DBN ici revient à initialiser les RBM qui le constitue.

**train_DBN** : Ici pareil, pour avant entrainer le DBN revient à entrainer les RBM le constituant, en prenant bien évidemment à chaque fois comme entrée la sortie du RBM d'avant.

**generer_image_DBN** :  Prend en argument un DBN entrainé, la taille du réseau et des images d'entrées, nbIterGibbs et le nombre d'images en sortie. Pour l'avant dernière couche on fait nbIterGibbs tirage de gibbs comme décrit plus haut, puis pour les couches inférieures on utilise sortie_entree_RBM , et on tire selon la loi obtenue. On fait cette opération jusqu'à arriver à la couche d'entrée, puis on tire selon la loi estimée des entrées pour obtenir des images produites de manière non supervisées ressemblant aux images de la base d'entrée.  Avec la configuration DBN on arrive à approcher la loi des entrées de données beaucoup plus complexes et diversifiées.
Avec les paramètres suivant et avec la chaine 'CD56' on obtient les résultats ci dessous :



nbIter=1000;
learningRate=0.1;
batchSize=100;
nbImages=50;
nbIterGibbs=3000;
tailleReseau=[320,500,500,500,500,4];

**NB : pour générer les images ci dessus, faire tourner mainDBN et mainDBN**

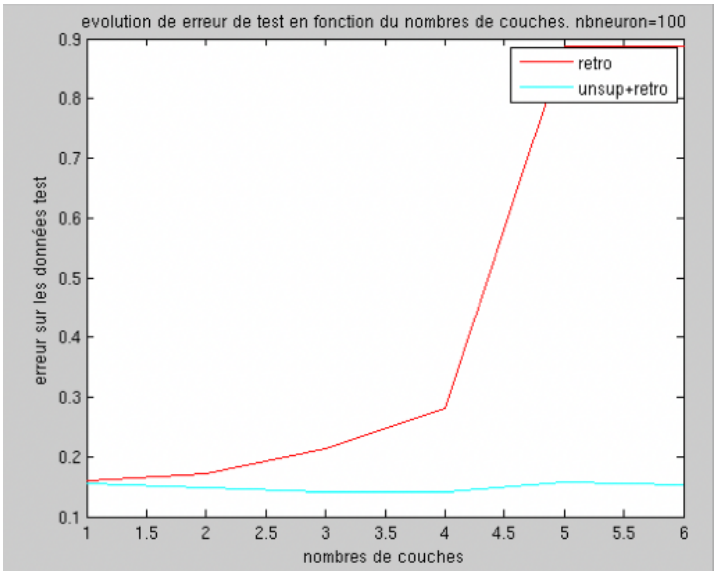
Le fait qu'on ai beaucoup plus de C que de 5 ou de 6 et de D est surement dû à l'initialisation. Aussi la distribution du 6 et du C sont à peu près similaire ce qui explique qu'on ai beaucoup plus de ces caractères là.

On compare les performances d'un réseau de neurones dont l'apprentissage supervisé a été initialisé avec un pré-entrainement par RBM avec un autre réseau directement appris par l'algorithme de rétro-propagation, initialisé aléatoirement. Cette comparaison est faite dans le script test_DNN

## III. Analyse :

### A- Variation du nombre de couches

On fait varier le nombre de couches avec un nombre de neurones fixe. On prend 100 neurones et on fait six variations de couches. 1-2-3-4-5 puis 6 couches. Pour chaque variation on prend l'erreur finale sur le test-set et on dessine le graph représentant l'erreur de test en fonction du nombre de couches ci dessous :




On remarque que au début les deux courbes commencent avec une erreur de 15 % à peu prés.

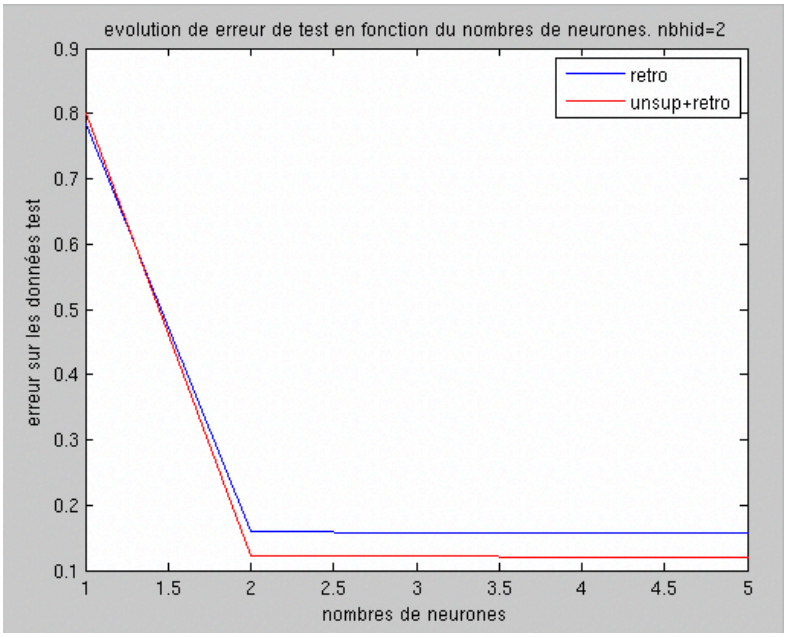
Pour la courbe rouge avec un apprentissage complétement supervisé, on voit que l'erreur augmente avec le nombre de couches. Par contre l'erreur sur le training-set elle est nul, ce qui veut dire que au bout d'un nombre de couche, le réseau commence à faire du sur-apprentissage. Ce sur-apprentissage intervient au passage de 4 couches à 5 couches et plus, et est marqué par un gap d'erreur. En effet l'erreur passe d'approximativement 25 % à une erreur de 90 % sur la base de test.

Pour la courbe bleu représentant l'apprentissage avec comme initialisation les paramètres du DBN entrainé. On voit que celle ci connaît deux phases. La première phase ou elle décroit avec l'augmentation du nombre de couches, celle si s'arrête à quatre couches, puis se met à croitre très doucement à partir de quatre couches.
Dans la deuxième courbe on a prévenu le sur-apprentissage en évitant les initialisations aléatoires. En effet en initialisant nos paramètres de façon réfléchie, c'est à dire ici avec les paramètres trouvés à la fin de l'entrainement de notre DBN.

On conclue donc que l'apprentissage semi supervisé avec DBN est plus performant que l'apprentissage complétement supervisé, et que augmenter le nombre de couche peut être bénéfique, mais il faut savoir s'arrêter au bon moment pour éviter le sur-apprentissage.

### B- Variation du nombre de neurones

Dans cette partie, on fixe le nombre de couches cachées à 2 et on varie le nombre de neurones.
On commence par 1 neurones puis 500, 1000, 1500, 2000 (dans le graph 1:1, 2:500, 3:1000 ainsi de suite) Les résultats obtenus sur le test-set sont représentés ci dessous :




On remarque d'abord que pour un seul neurone, on ne parvient pas à obtenir un modèle assez complexe pour nos données, on se retrouve donc avec une grosse erreur au début. Ensuite à partir de 500 neurones l'erreur se stabilise autour de 16 % pour la courbe bleu et 11 % pour la courbe rouge. Dans ce cas aussi on a de meilleurs performances pour l'initialisation non-supervisé à l'aide du DBN.