PROJET ML

26 mai 2021

Réseau de Neurones DIY

GIANG Cécile (3530406)

LENOIR Romain (3670199)

PROJET ML

RESEAU DE NEURONES DIY

INTRODUCTION

L'objectif de ce projet est de **développer une librairie python** implémentant un réseau de neurones. L'implémentation est inspirée des anciennes versions de *pytorch* (avant l'autograd) et des implémentations analogues qui permettent d'avoir des réseaux génériques très modulaires. Chaque couche du réseau est vu comme un module, et un réseau est constitué ainsi d'un ensemble de modules. En particulier, les fonctions d'activation sont aussi considérées comme des modules.

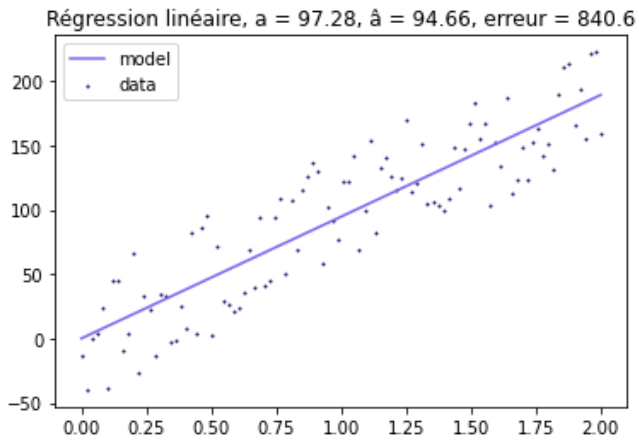
Notre librairie étant centrée autour d'une classe abstraite `Module`, la première étape a donc été de la définir. Le reste de notre projet s'appuiera sur cette classe principale.

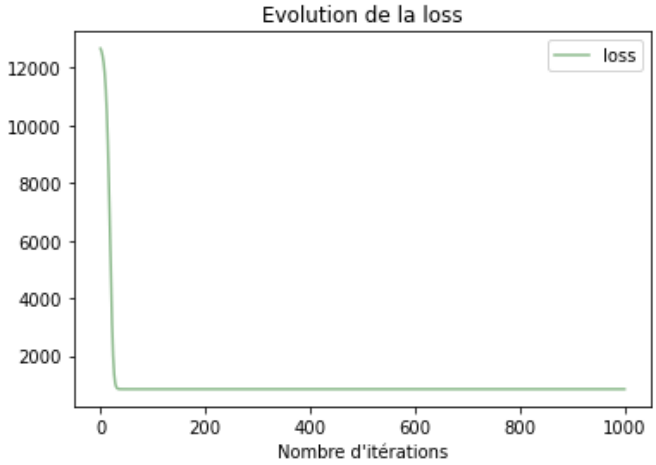
MON PREMIER RESEAU EST … LINEAIRE

Le premier réseau que nous implémentons est une **régression linéaire** : c'est un réseau à une seule couche linéaire cherchant à minimiser un coût aux moindres carrés (classe RegLin). Nous générons un ensemble de points (x,y) où y est défini selon un coefficient directeur a à partir de l'abscisse x. L'objectif de notre réseau est donc de retrouver ce coefficient a en essayant de minimiser la fonction de coût (MSE).

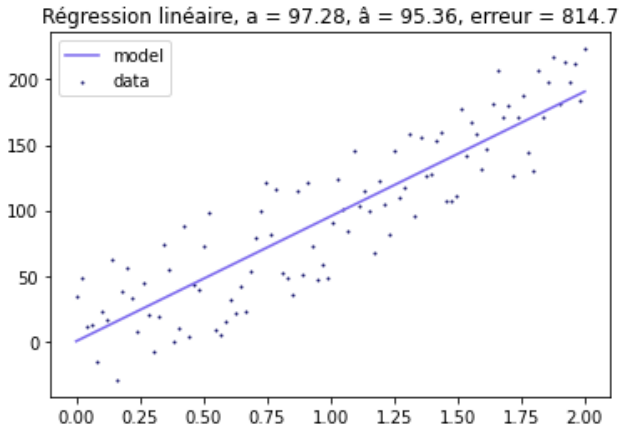
Dans nos tests nous rajoutons également un bruit suivant une loi uniforme dans un intervalle $[\sigma, \sigma]$ que nous fixerons arbitrairement.

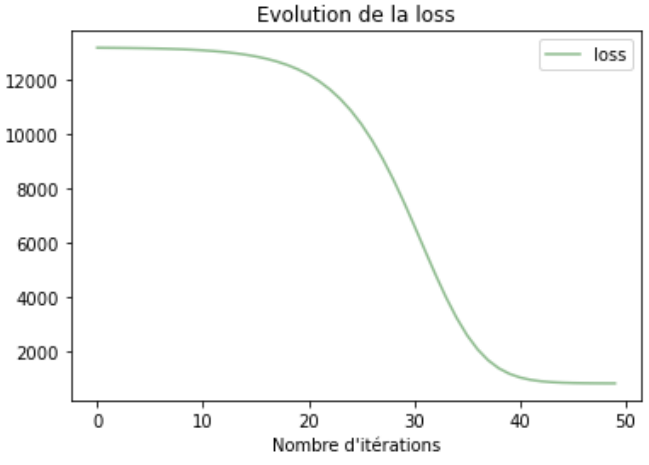
Premier test : Nous fixons arbitrairement le coefficient directeur à $a = 97.28$ ainsi que $\sigma = 50$.





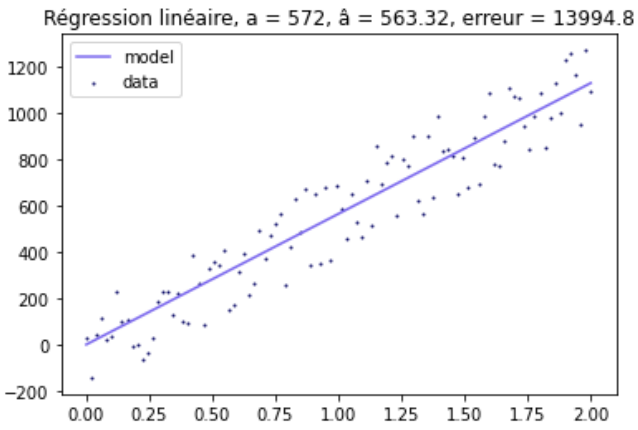
Dans notre première figure, les points sont les données en entrée du réseau de neurones et la droite tracée correspond à $f(x) = \hat{a}x$ où $\hat{a}$ est le paramètre w optimisé par le réseau. Le résultat trouvé correspond bien à ce que nous cherchions : le réseau renvoie un coefficient directeur $\hat{a} = 94.66$ (valeur cherchée : $a = 97.28$) malgré le bruit. Nous notons également que la loss décroît très rapidement : bien que nous ayons fait tourner notre réseau sur 1000 itération, le réseau n'a besoin que de 50 itérations pour converger. Voici ce que nous obtenons pour 50 itérations :

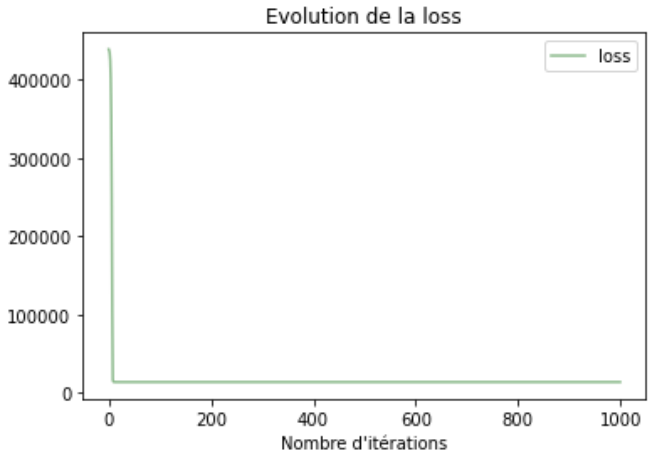




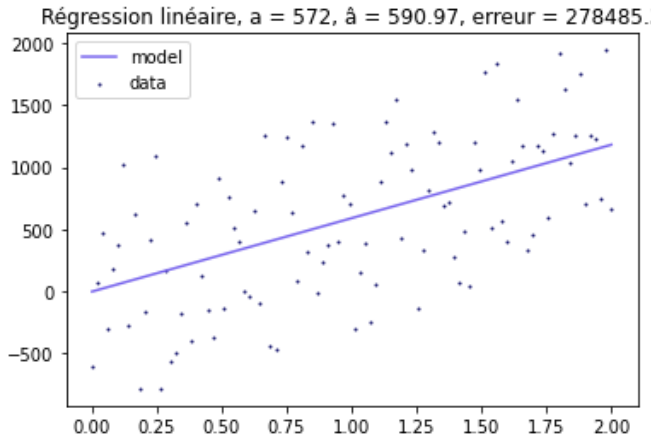
Deuxième test : Nous fixons cette fois a = *572* et faisons maintenant varier le bruit. Nous observons que la loss explose pour un bruit très élevé.

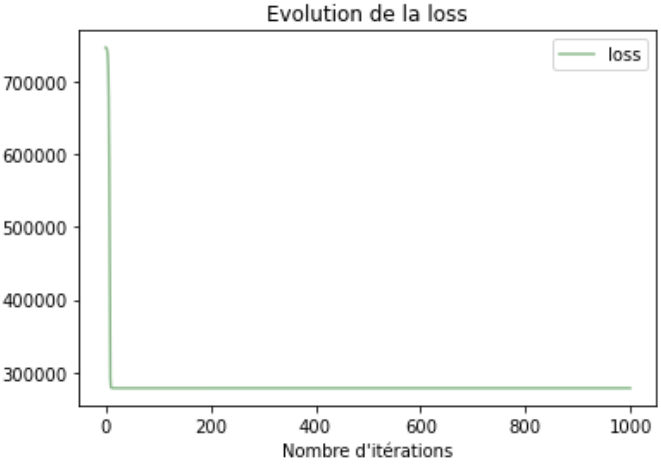
- Pour σ = *1000*.



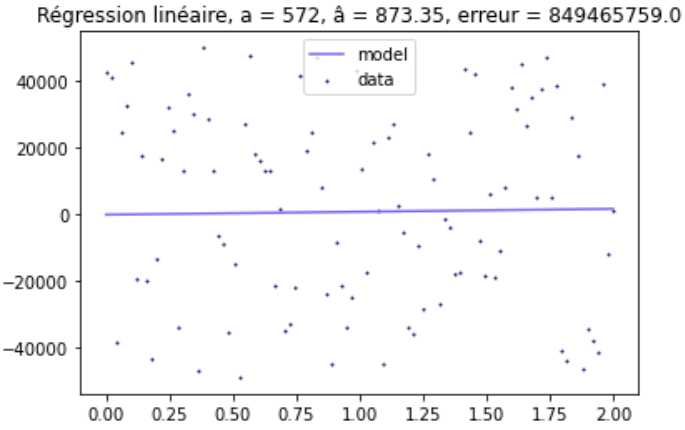


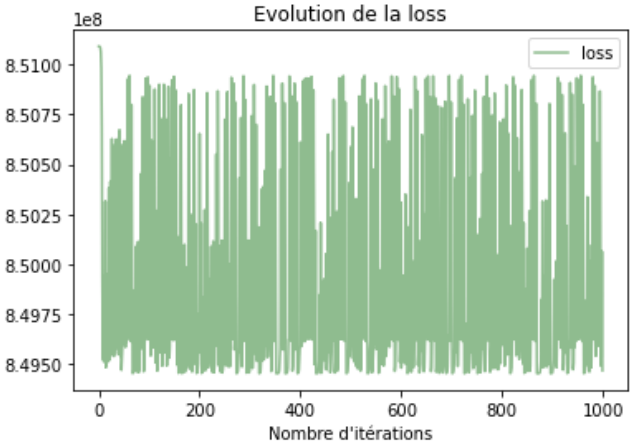
- Pour σ = *10000*.





Pour σ = *50000*.





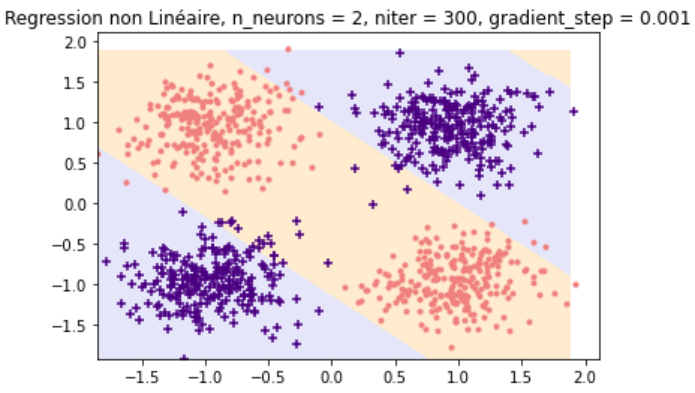
MON SECOND RESEAU EST … NON-LINEAIRE

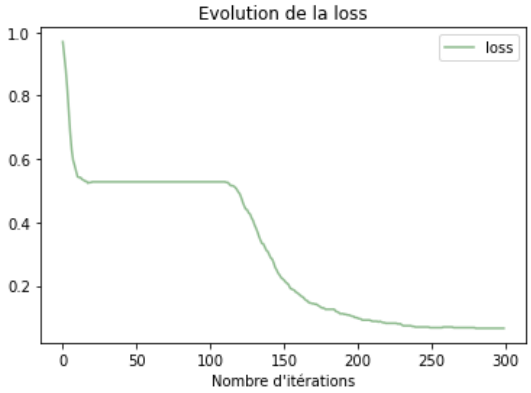
Nous souhaitons maintenant implémenter un réseau non-linéaire avec deux couches cachées, dont les fonctions d'activation sont respectivement une tangente hyperbolique et une sigmoide (classe NonLin). Notre réseau cherche à trouver le paramètre *w* minimisant un coût aux moindres carrés (*mse loss*).

Nous nous rendons vite compte que le processus d'inférence (*forward*) et de rétro-propagation (*backward*) peuvent devenir très répétitifs à coder, et il serait difficile de s'imaginer créer un réseau plus complexe de cette manière. Nous créons donc deux nouvelles classes Sequentiel et Optim, dont les buts sont respectivement de rajouter en série des modules à un réseau puis appliquer la phase forward, et d'exécuter la passe backward.

Nous implémentons donc une nouvelle version de notre réseau non-linéaire (classe NonLin2) s'appuyant sur ces deux nouvelles classes. La particularité de NonLin2 par rapport à NonLin est qu'elle contient une fonction *SGD* qui correspond à la fonction *fit* de NonLin, mais qui offre en plus la possibilité de spécifier la taille des batchs en apprentissage : ainsi, un batch de taille 1 correspond à une descente de gradient stochastique, un batch de taille *len(data)* à un batch, et un batch de taille compris entre 1 et *len(data)* à un mini-batch. Nos tests se font sur des données artificielles suivant 4 gaussiennes et 2 classes.

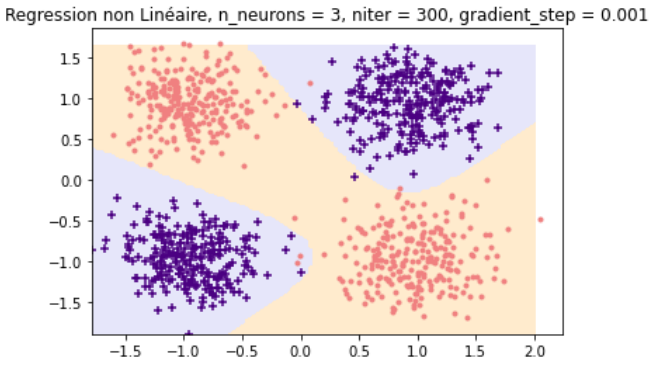
- **Réseau à 2 neurones :** 2 neurones, *niter = 300*, *gradient_step = 1e-3*, *batch_size = len(datax)*

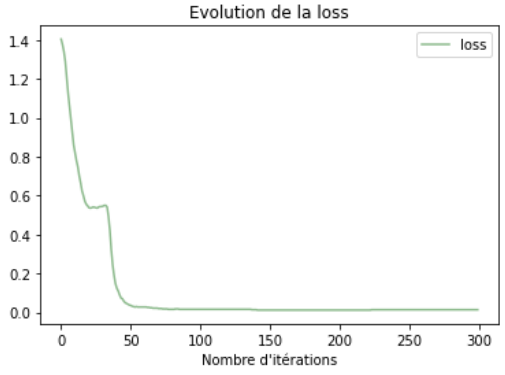




Le score de bonne classification est de 0.967 pour une convergence en environ 250 itérations. Nous augmentons le nombre de neurones pour voir l'impact du nombre de neurones sur la performance du réseau.

- **Réseau à 3 neurones :** 3 neurones, *niter = 300*, *gradient_step = 1e-3*, *batch_size = len(datax)*

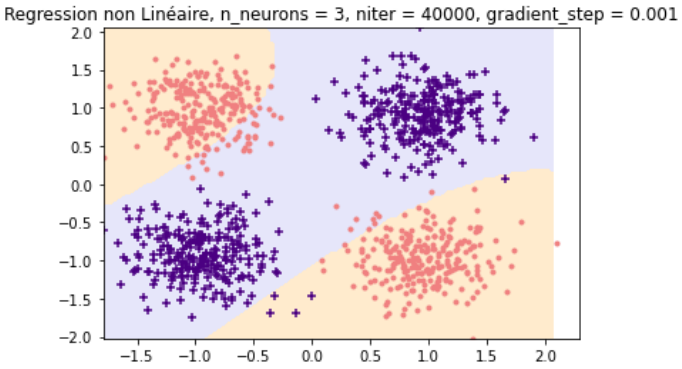


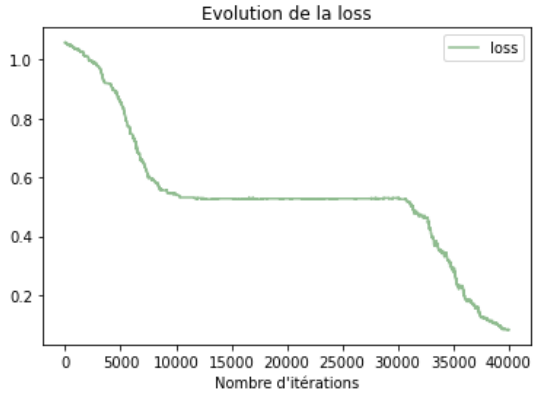


Nous voyons qu'avec 3 neurones, le réseau converge beaucoup plus rapidement (50 itérations) avec une loss bien plus faible (presque nulle). La performance est bien supérieure au réseau à 2 neurones : le taux de bonne classification est monté à 0.99.

Les tests précédents ont été faits pour un batch de taille *len(datax)*, soit pour une descente de gradient en batch. Nous voulons maintenant tester les performances du réseau en stochastique et mini-batch.

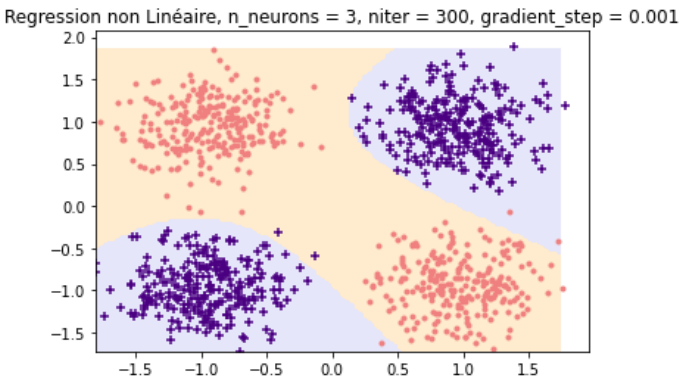
- **Réseau à 3 neurones avec *batch_size = 1*** (stochastique)

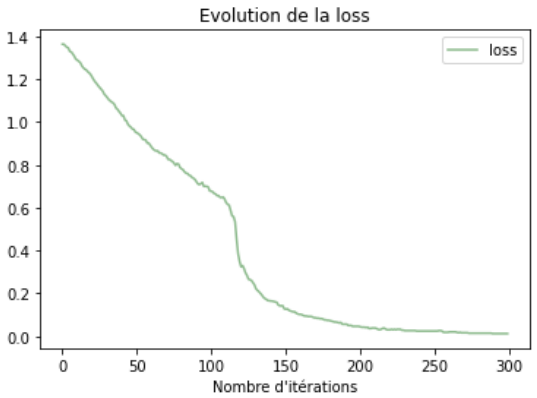




Bien que nous ayons gardé le même pas de gradient, le réseau met bien plus d'itérations à converger. Cette convergence arrive aux alentours de 40 000 itérations, mais ce résultat n'est pas stable : il arrive très souvent que le réseau diverge et que la loss explose. Cela nous paraît compréhensible puisque les poids du réseau sont modifiés à chaque exemple tiré.

- **Réseau à 3 neurones avec batch_size = 100** (mini-batch)





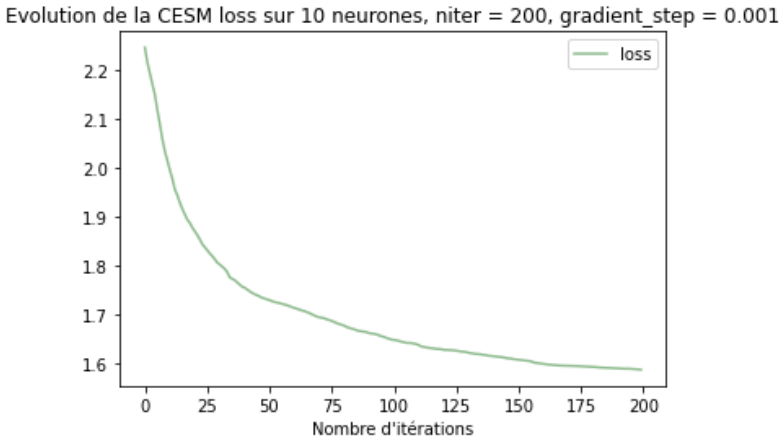
Nous voulons cette fois-ci apprendre sur un dixième des données à chaque itération et fixons donc le paramètre *batch_size* à 100. La performance du réseau est stable (pas de divergence) et nous donne un score de bonne classification de 0.994. Nous notons cependant que le réseau a besoin de plus d'itérations pour converger que dans le cas batch (200 itérations contre 50).

MON TROISIEME RESEAU EST … MULTICLASSE

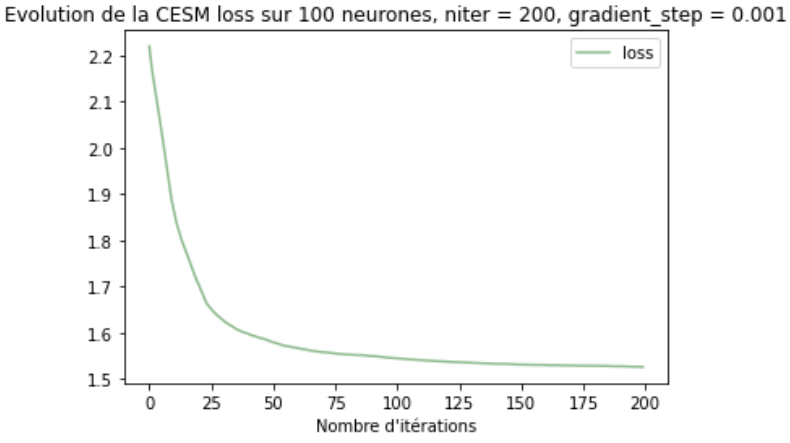
Jusqu'à maintenant nous n'avons traité que les problèmes binaires. Nous nous nous intéressons maintenant au cas multiclasse. Nous introduisons pour cela un SoftMax en dernière couche afin de transformer les données en entrée de couche en distribution de probabilités sur chaque classe. Ainsi pour chaque exemple, nous prédirons la classe qui maximise est probabilité. Le coût utilisé est le coût cross-entropique. Notre réseau est maintenant composé de 1 couches cachées, avec pour fonctions d'activation respectives une tangente hyperbolique et un softmax.

Nous effectuons cette fois-ci nos tests sur les données de chiffres manuscrits USPS. Nous travaillons donc sur 10 classes. Nous souhaitons connaître l'impact du nombre de neurones sur la performance du réseau.

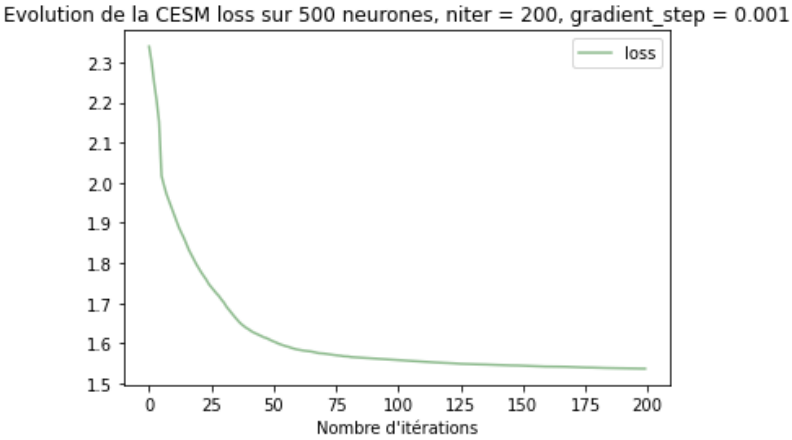
- **Réseau multiclasse à 10 neurones :** 10 neurones, *niter = 200*, *gradient_step = 1e-3*



- **Réseau multiclasse à 100 neurones :** 100 neurones, *niter = 200*, *gradient_step = 1e-3*



- **Réseau multiclasse à 500 neurones :** 500 neurones, *niter = 200*, *gradient_step = 1e-3*



Nombre de neurones	**Score de bonne classification**
10	0.791
100	0.812
500	0.831

Nous remarquons que plus le nombre de neurones utilisé est élevé, meilleure est la performance et plus la convergence est rapide. Le gain de performance est cependant très faible comparé au nombre d'itérations ajouté.

MON QUATRIEME RESEAU EST … UN AUTO-ENCODEUR

Nous souhaitons désormais coder un réseau capable de compresser l'information contenue dans nos données tout en minimisant la perte d'information. Nous implémentons donc un auto-encodeur (classe), qui est un réseau de neurones dont l'objectif est d'apprendre un encodage des données dans le but de réduire les dimensions. Il s'agit d'apprentissage non supervisé ; on se sert de l'exemple encodé comme nouvelle description de l'exemple pour ensuite faire de l'apprentissage supervisé classique.

Notre réseau est composé de deux sous-réseaux :

- Un **encodeur** composé de 2 couches cachées dont les fonctions d'activation sont deux tanh
- Un **décodeur** composé de 2 couches cachées dont les fonctions d'activation sont un tanh et une sigmoide

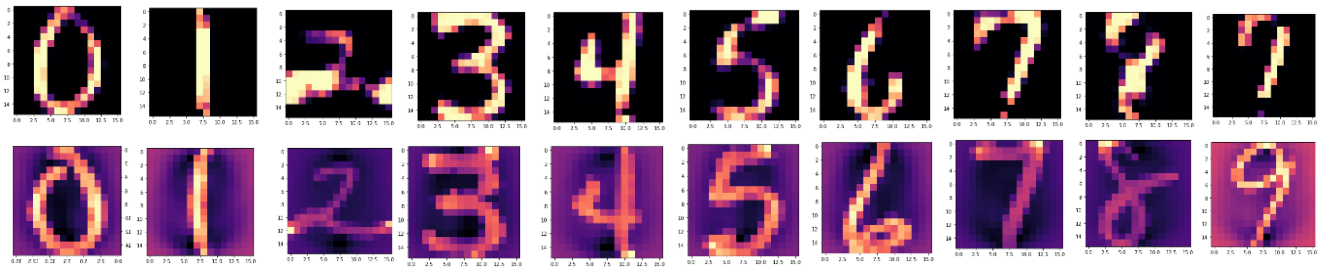
L'objectif du réseau est de minimiser le coût de reconstruction sur une donnée encodée : cela quantifie la perte d'information.

Pour nos tests, nous restons encore une fois sur les données chiffrées USPS : une réduction des dimensions sur une image correspond à une compression, c'est ce que nous voulons confirmer visuellement.

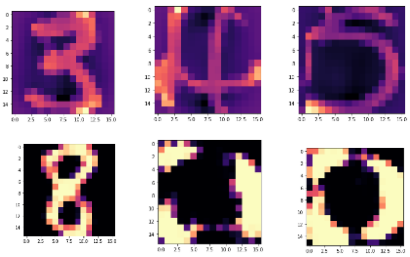
VISUALISATION DE QUELQUES COMPRESSIONS D'IMAGES

Le nombre de neurones correspond à la dimension de l'espace en sortie de l'encodeur. Ainsi plus le nombre de neurones est élevé, moins l'on perd d'information sur les données. Nous regardons quelques images recomposées après une réduction de dimensions pour différents

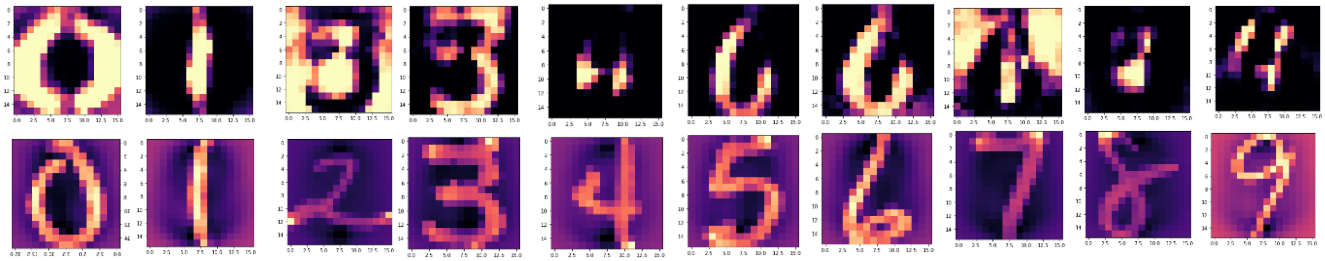
- **Pour 100 neurones** *(500 itérations)*: en haut les images reconstruites, en bas les images d'origine



Les compressions avec 100 neurones gardent en général assez d'information sur les images d'origine pour que leurs reconstructions dans l'espace d'origine soient correctes, comme nous pouvons l'observer ci-dessus. Il y a cependant des exceptions comme nous pouvons le voir ici :



- **Pour 10 neurones :** *(500 itérations)* en haut les images reconstruites, en bas les images d'origine

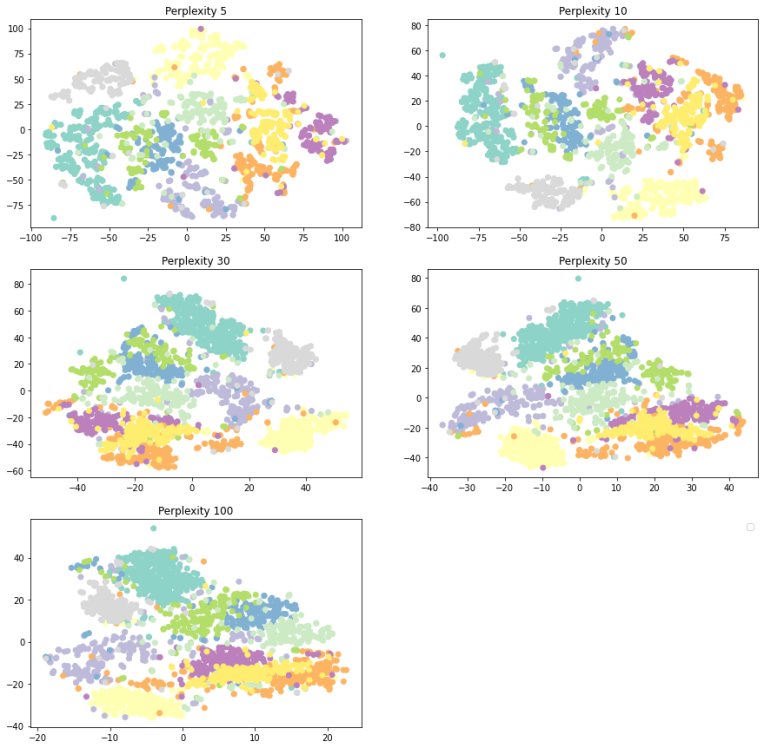


Pour le même nombre d'itérations, la reconstruction des images est beaucoup moins bonne. Nous voyons assez facilement que le 5 reconstruit ressemble plus à un 6, et ressemble même très fortement à la reconstruction du 6. Le 2 reconstruit ressemble à un 3, et est exactement l'inverse du 3 reconstruit. Il semblerait qu'en compressant beaucoup, nous gardons beaucoup moins bien les informations discriminant chaque classe d'image et la reconstruction se généralise beaucoup.

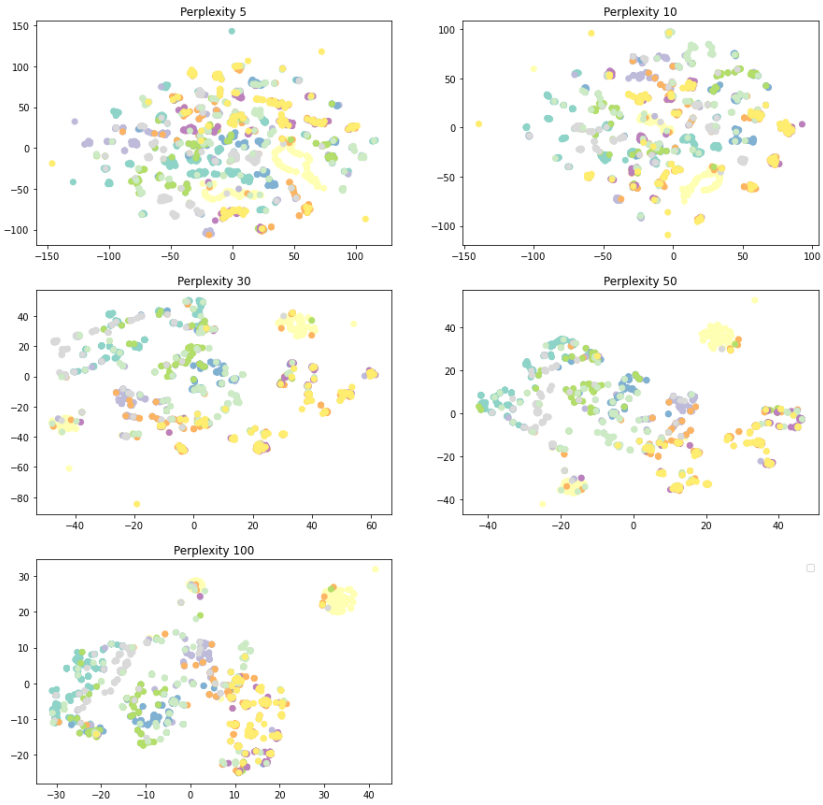
T-SNE : PROJECTION DANS UN ESPACE LATENT

Nous nous attachons maintenant à visualiser de manière plus générale la perte d'information liée à la compression. Nous décidons pour cela d'utiliser l'algorithme t-sne afin de projeter nos données dans un espace latent de dimension 2. Si la compression garde assez bien les informations sur les dimensions discriminantes propres à chaque classe, alors celles-ci devraient se regrouper dans la projection. Si ces caractéristiques discriminantes sont perdues, les chiffres seront entremêlés entre eux.

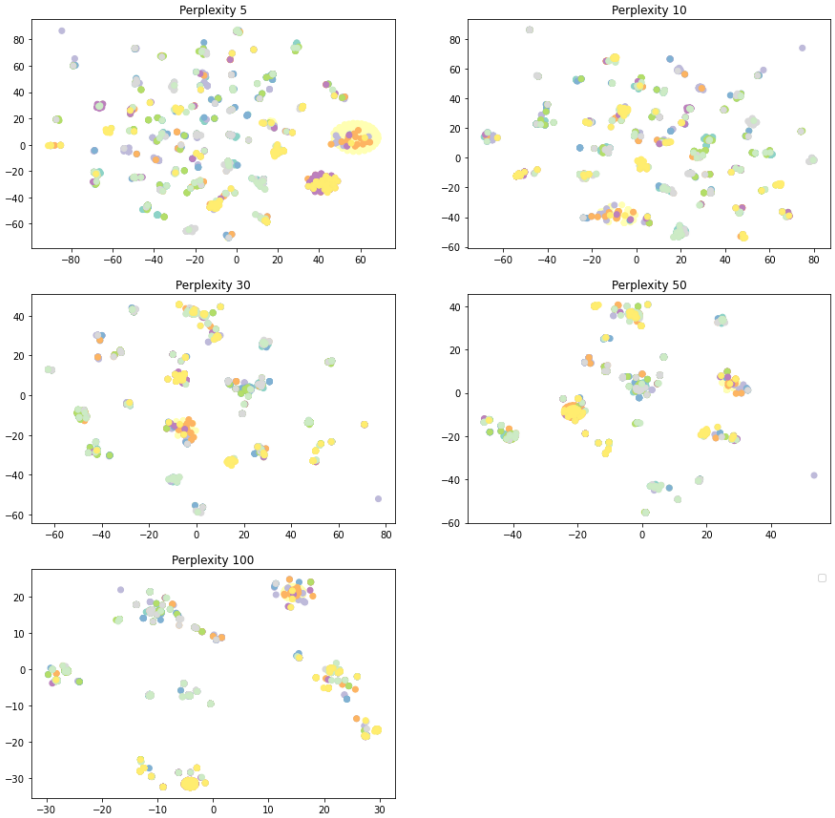
Nous visualisons d'abord notre projection en dimension 2 sur les données originales, pour différentes valeurs de perplexité. La notion de perplexité est reliée au nombre de voisins proches pris en compte dans l'apprentissage non-supervisés.



- **TSNE pour la reconstruction d'une compression à 100 neurones**



- **TSNE pour la reconstruction d'une compression à 10 neurones**



La visualisation t-sne nous permet de voir très clairement ce que nous constations déjà en affichant les images reconstruites : alors que la projection en 2 dimensions sur les données de départ nous permet de distinguer correctement les différentes classes de chiffres, en particulier à perplexité faible (< 10), la projection en 2D sur les images reconstruites après compression en espace de dimension 100 nous indique que les classes ont perdu de l'information discriminante. Nous pouvons toujours remarquer que les données reconstruites de certaines classes gardent tout de même une certaine similarité : par exemple, nous voyons que pour une perplexité 5, les données de la classe jaune sont regroupés vers le haut. En baissant maintenant le nombre de neurones à 10, la visualisation nous permet encore moins de distinguer les classes, les classes ont perdu trop d'information discriminante et l'on est de moins en moins capable de reconstruire une image appartenant à la bonne classe.