Master DAC

Projet ML

Réalisé par :

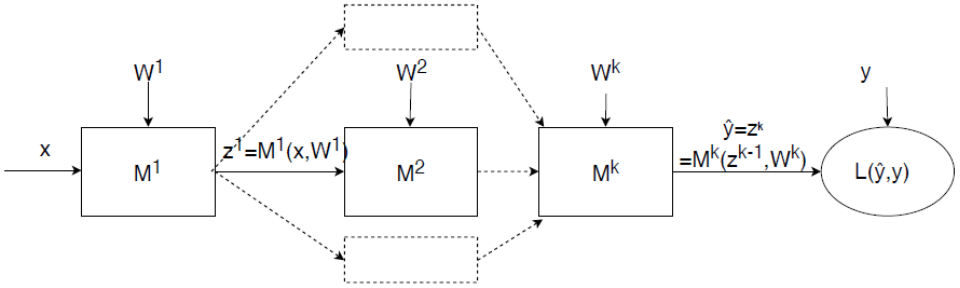
Samy NEHLIL

Allaa BOUTALEB

Paris, Mai 2023

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.



Description de la structure du code:

Le code est disponible dans le répertoire github suivant :
https://github.com/samynhl/ML-Projet-NN.git

Le code source du projet est structuré de la manière suivante:

```
├──rapport
├──src
│   ├──activation
│   ├──loss
└──test
```

Tel que:

src : Un répertoire qui contient tout les classes abstraites (modules) nécessaires à l'implémentation des différentes architectures de réseaux de neurones, notamment **Module, Loss, Linear, Sequentiel, Optim, AutoEncodeur, Conv1D, Conv2D, Flatten,** ainsi que des classes d'activation et de calcul de cout.

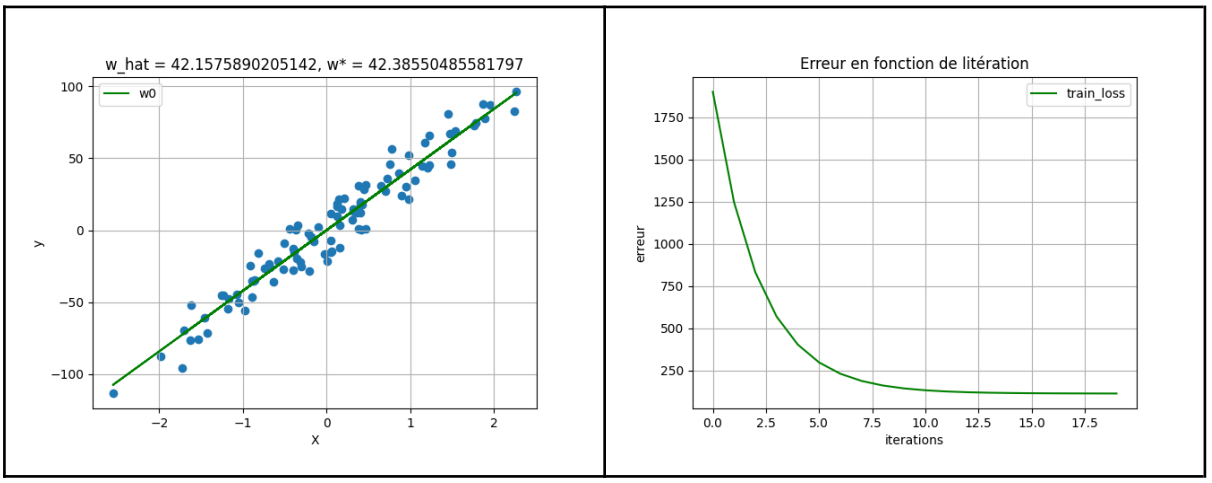
test: Un répertoire qui contient les différents tests et expériences menés afin de tester les modules de notre réseau de neurones. Chaque test porte sur une partie du projet et est réalisé dans un notebook jupyter à part.

Mon premier réseau est … Linéaire

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 Linéaire). 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 ajoutons également un bruit suivant une loi uniforme dans un intervalle [σ, σ] que nous fixons arbitrairement.

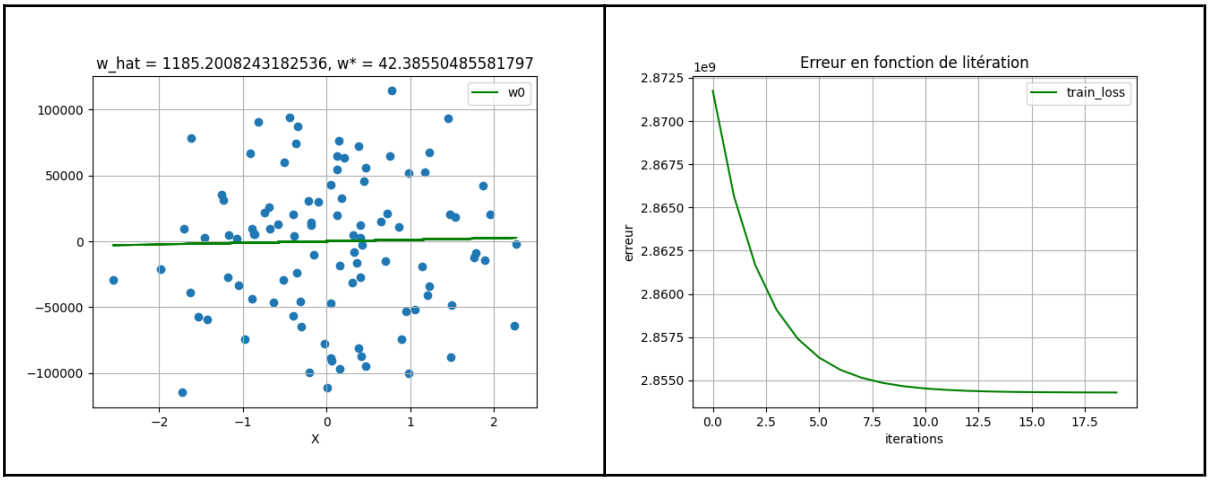
Premier test:

Nous commençons par générer un dataset artificiel où les données sont linéairement séparables, nous utilisons pour cela le module make_regression de sklearn, et nous appliquons notre réseau linéaire à une couche sur le dataset pour évaluer les résultats



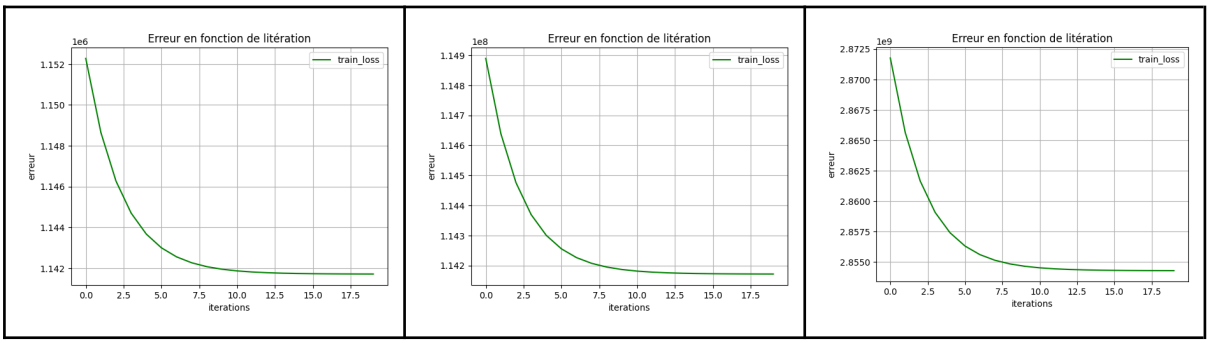
Dans notre première figure, les points sont les données en entrée du réseau de neurones et la droite tracée correspond à f(x) = âx où â est le paramètre w optimisé par le réseau. Le résultat trouvé correspond bien à ce que nous cherchions : le réseau renvoie un coefficient directeur â = 42,61 (valeur cherchée : a = 42.38) malgré le bruit.
Nous notons également que la loss décroît très rapidement : bien que nous ayons fait tourner notre réseau sur 200 itération, le réseau n'a besoin que de 50 itérations
pour converger.

Voici ce que nous obtenons pour 20 itérations :



Deuxième test:

Nous allons maintenant faire varier le bruit. Nous observons que la loss explose pour un bruit très élevé.

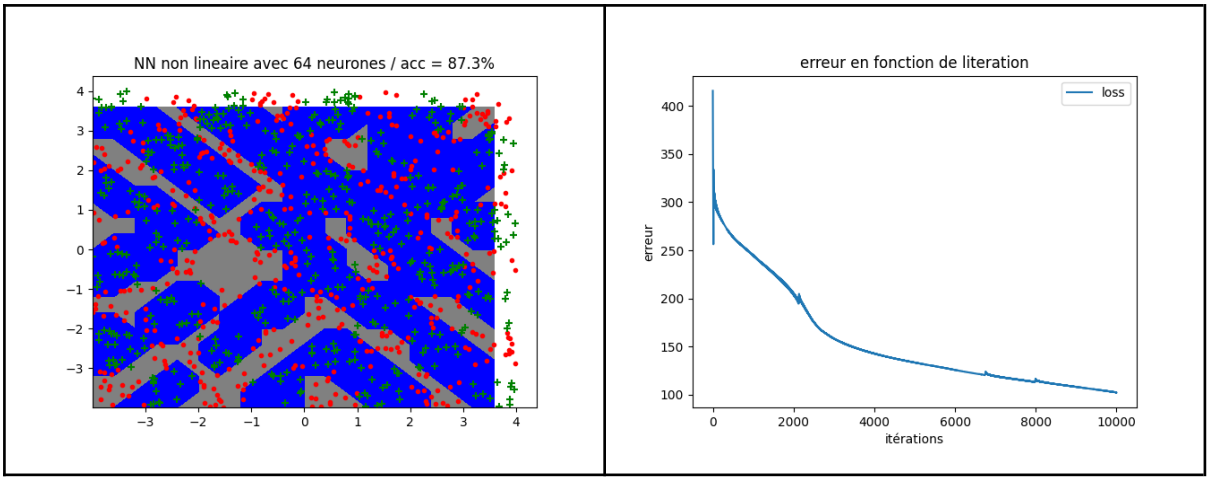


Analyse des résultats:

Nous observons que la fonction de coût explose quand le bruit introduit dans les données d'entraînement est important.

Mon deuxième réseau est … non-Linéaire

Nous souhaitons maintenant implémenter un réseau non-linéaire avec deux couches cachées, dont les fonctions d'activation sont respectivement une tangente hyperbolique (tanH) et une sigmoide (classe NonLin). Notre réseau cherche à trouver le paramètre w minimisant un coût aux moindres carrés (mse loss).
Pour tester ce module, nous commençons par générer un dataset artificiel où les données suivent 4 gaussiennes, et nous appliquons notre réseau linéaire avec deux couches cachées sur le dataset pour évaluer les résultats.

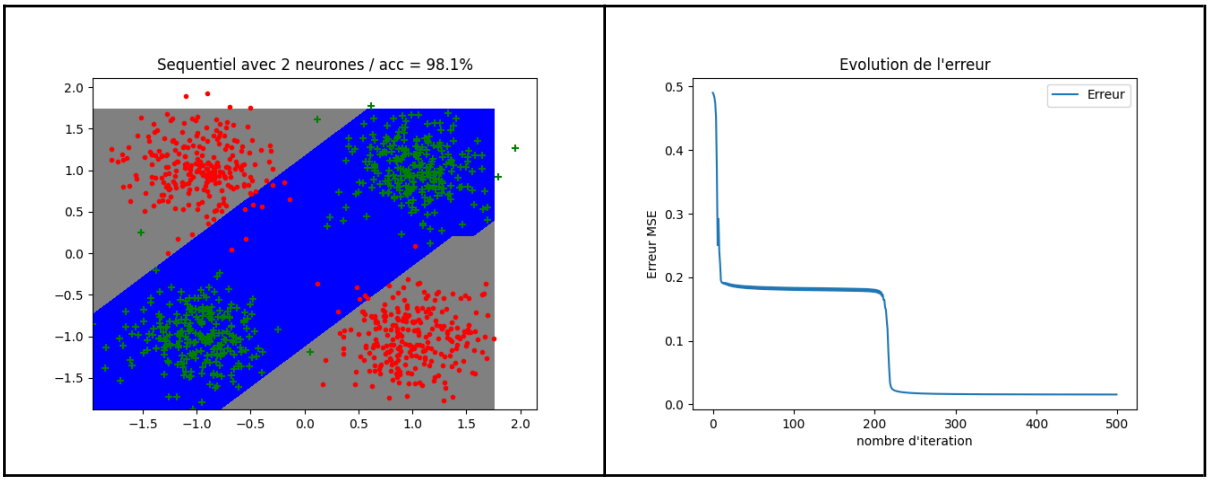


Mon troisième réseau est … encapsulage

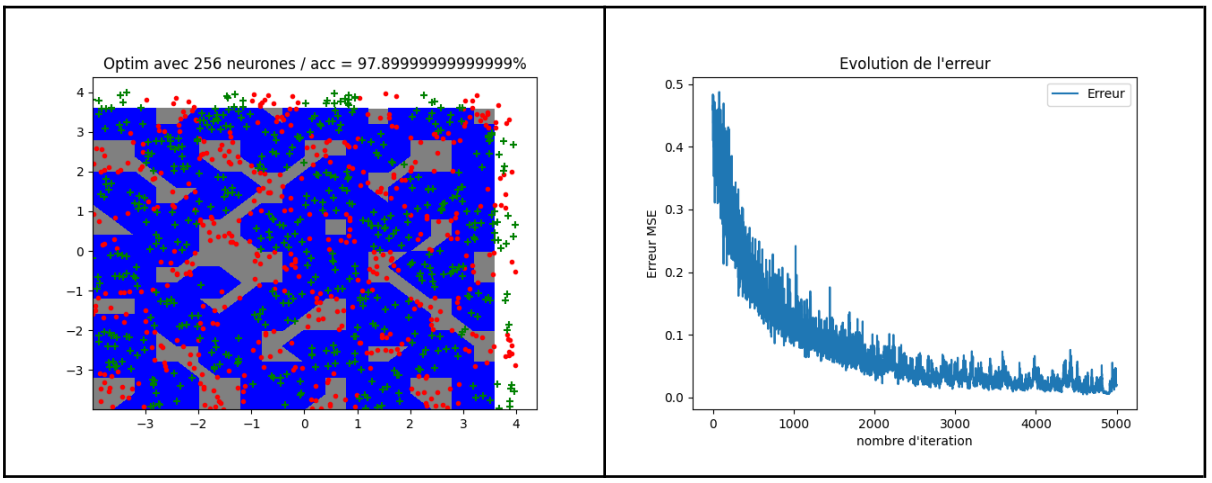
Nous nous apercevons très vite que le processus d'inférence (forward) et de rétro-propagation (backward) peuvent devenir très répétitifs à coder (et par conséquent un code moins efficace), et il serait difficile de s'imaginer créer un réseau plus complexe de cette manière.

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 (Type architecture Keras).

Test du module Sequentiel:

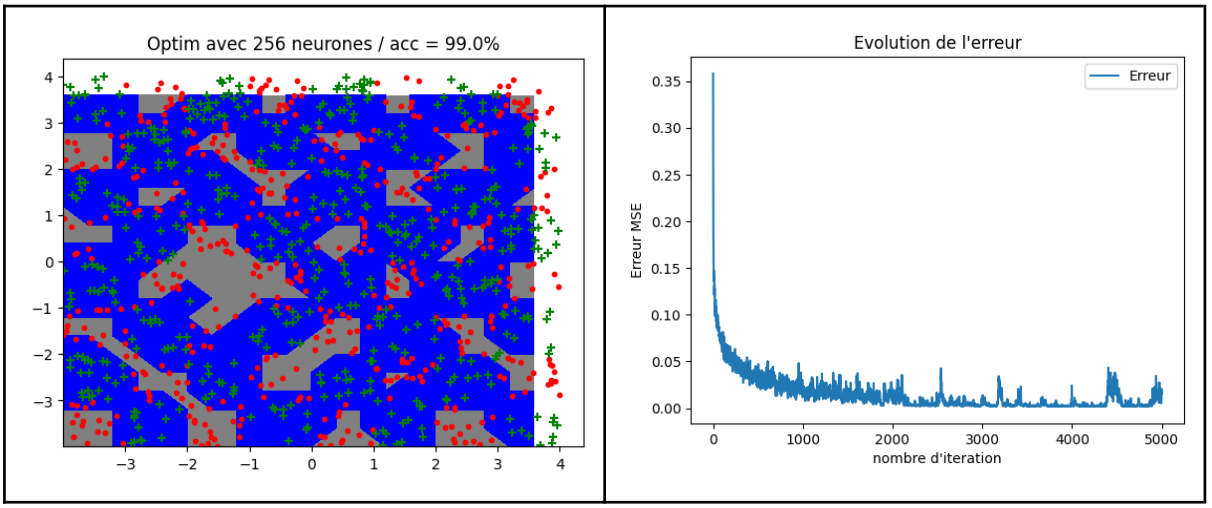


Test du module Optim:



Test du module Optim avec une descente de gradient stochastique SGD:

Nous implémentons donc une nouvelle version de notre réseau non-linéaire (classe SGD(Optim)) s'appuyant sur ces deux nouvelles classes. La particularité de SGD par rapport à Optim est qu'elle effectue un fit qui est basé sur une descente de gradient stochastique 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.

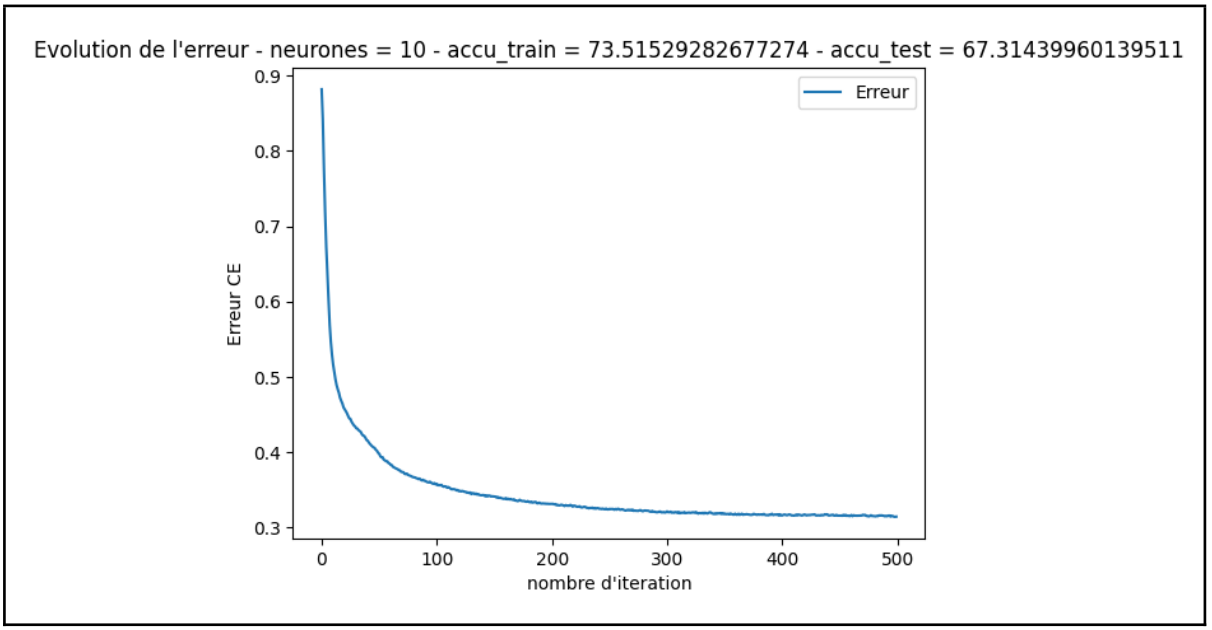


Mon quatrième réseau est … MultiClasse

Jusqu'à maintenant nous n'avons traité que les problèmes binaires. 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 prédisons 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 respective 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.

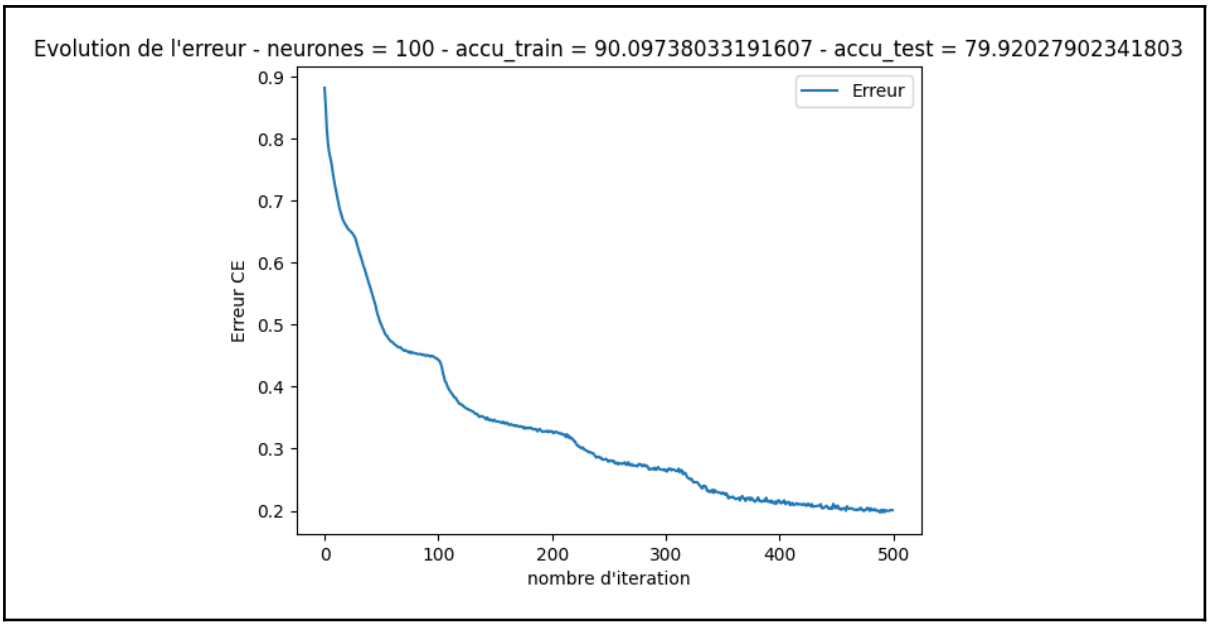
Premier test:

Réseau multiclasse à 10 neurones : 10 neurones, niter = 500, gradient_step = 1e-2



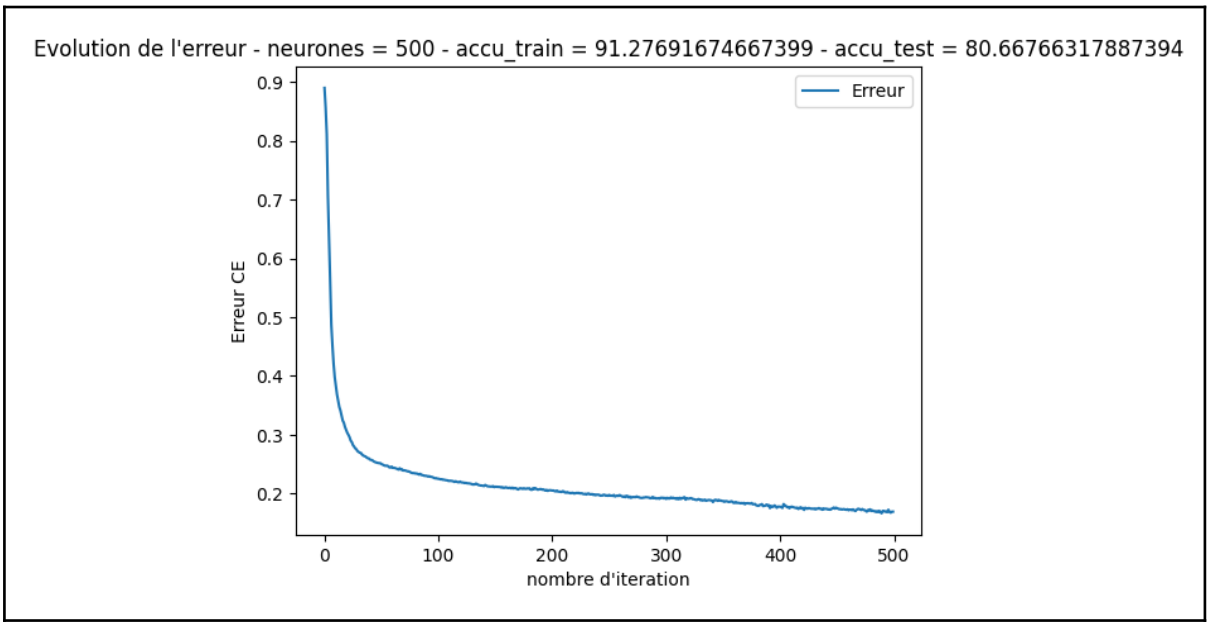
Deuxième test:

Réseau multiclasse avec : 100 neurones, niter = 500, gradient_step = 1e-2



Troisième test:

Réseau multiclasse avec: 500 neurones, niter = 500, gradient_step = 1e-2



Analyse des résultats:

Nombre de neurones	Taux de bonne classification train %	Taux de bonne classification test %
10	73.52	67.31
100	90	80
500	91	81

Nous remarquons que plus le nombre de neurones utilisé est élevé, meilleure est la performance et plus la convergence est rapide.

Mon cinquième réseau est un … auto encodeur

Nous souhaitons désormais proposer un réseau capable de compresser l'information contenue dans nos données tout en minimisant la perte de cette dernière. 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 sigmoïde.

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.

Concernant les tests de ce module, nous reprenons l'architecture Encodeur-Décodeur proposée dans le cahier des charges du projet:

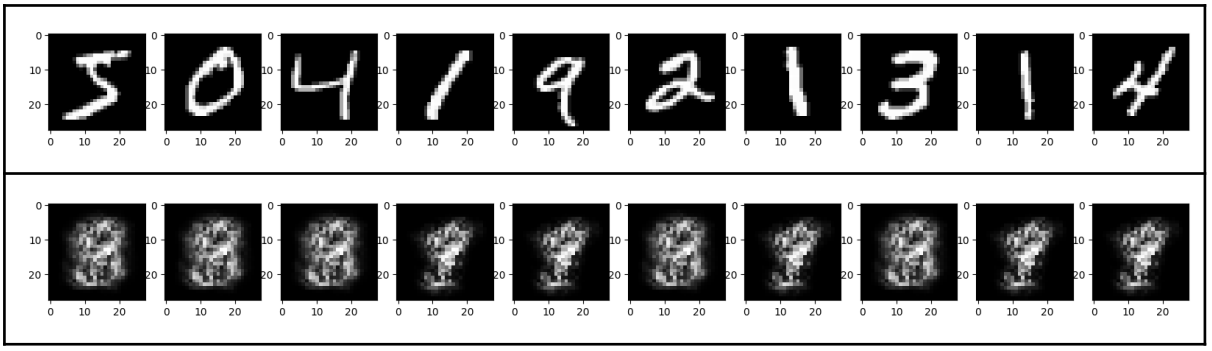
- Encodage : Linear(256,100) → TanH() → Linear(100,10) → TanH()
- Décodage : Linear(10,100) → TanH() → Linear(100,256) → Sigmoïde()

Dans la série de trois tests suivants, nous évaluons l'impact du nombre d'itérations sur les résultats de notre auto Encodeur, nous varions pour cela le nombre d'itérations afin d'en évaluer visuellement les résultats.
Les valeurs prises pour le nombre d'itérations sont : 20, 50, 500.

Première architecture:

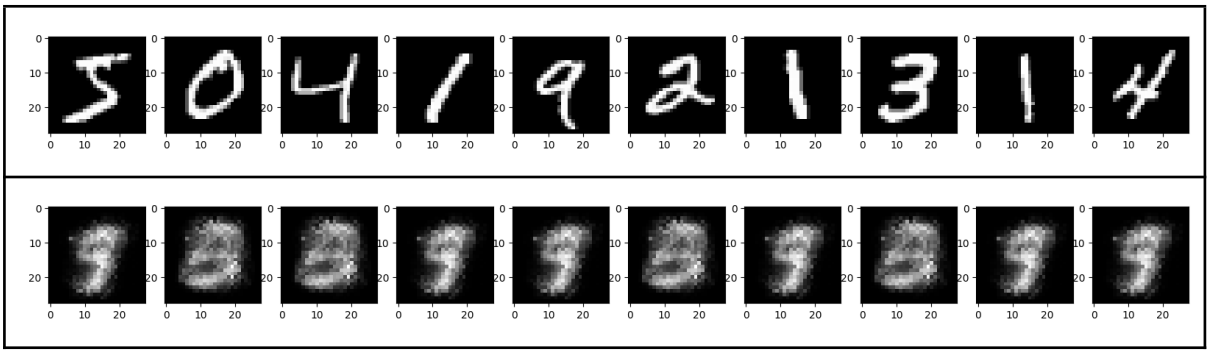
Premier test:

Pour 100 neurones (20 itérations): en haut les images originales, en bas les images reconstruites.



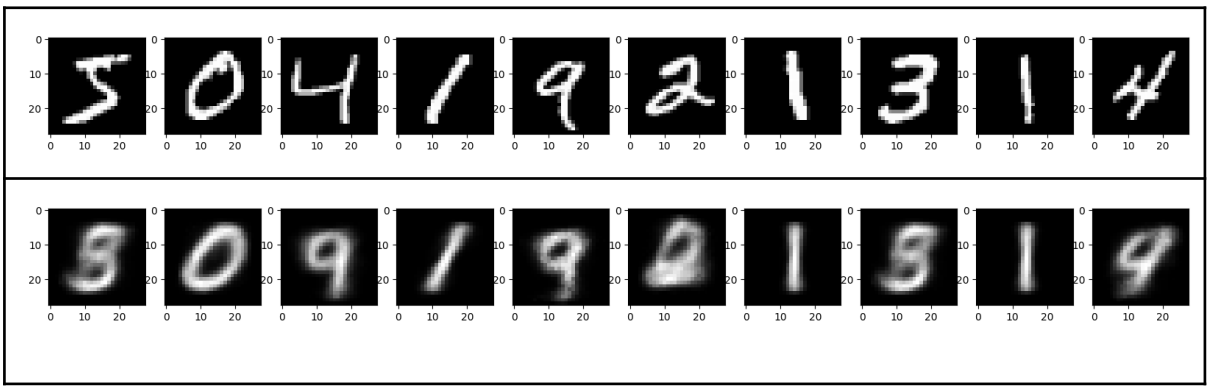
Deuxième test:

Pour 100 neurones (50 itérations): en haut les images originales, en bas les images reconstruites.



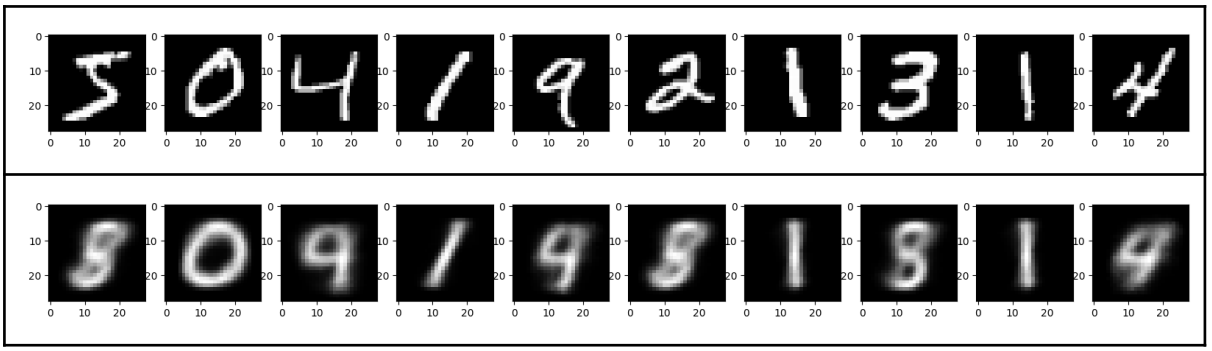
Troisième test:

Pour 100 neurones (100 itérations): en haut les images originales, en bas les images reconstruites.



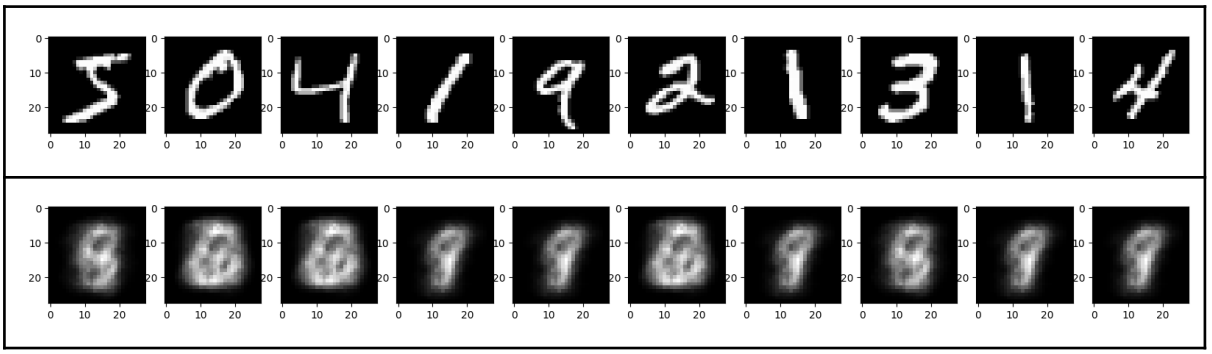
Deuxième architecture:

- Encodage : Linear(256,50) → TanH() → Linear(50,5) → TanH()
- Décodage : Linear(5,50) → TanH() → Linear(50,256) → Sigmoïde()



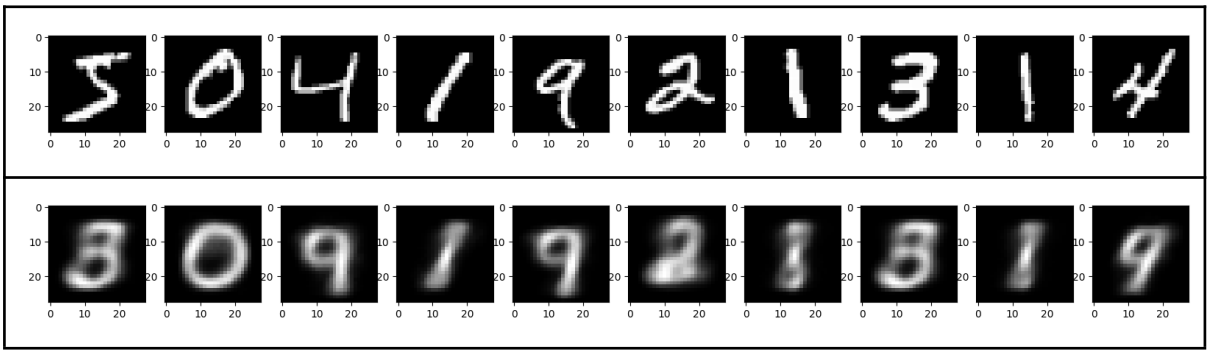
Troisième architecture:

- Encodage : Linear(256,200) → TanH() → Linear(200,20) → TanH()
- Décodage : Linear(20,200) → TanH() → Linear(200,256) → Sigmoïde()



Quatrième architecture:

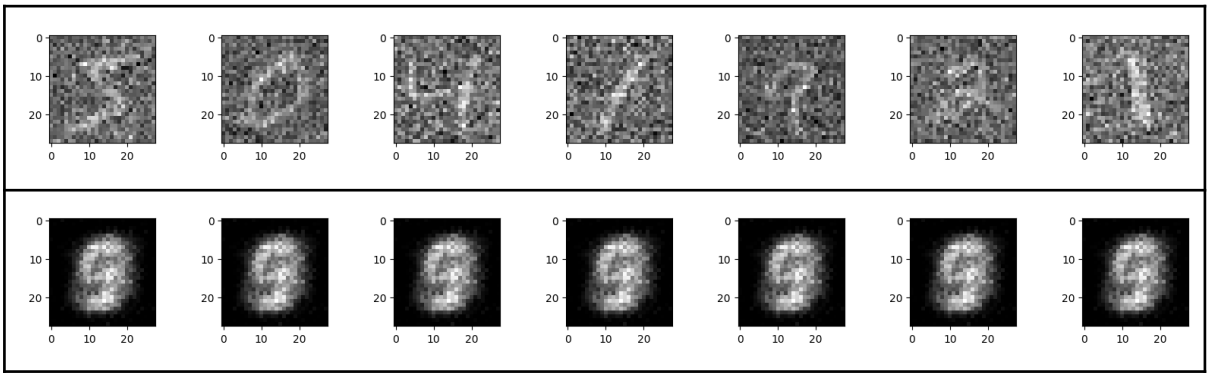
- Encodage : Linear(256,20) → TanH() → Linear(20,2) → TanH()
- Décodage : Linear(2,20) → TanH() → Linear(20,256) → Sigmoïde()



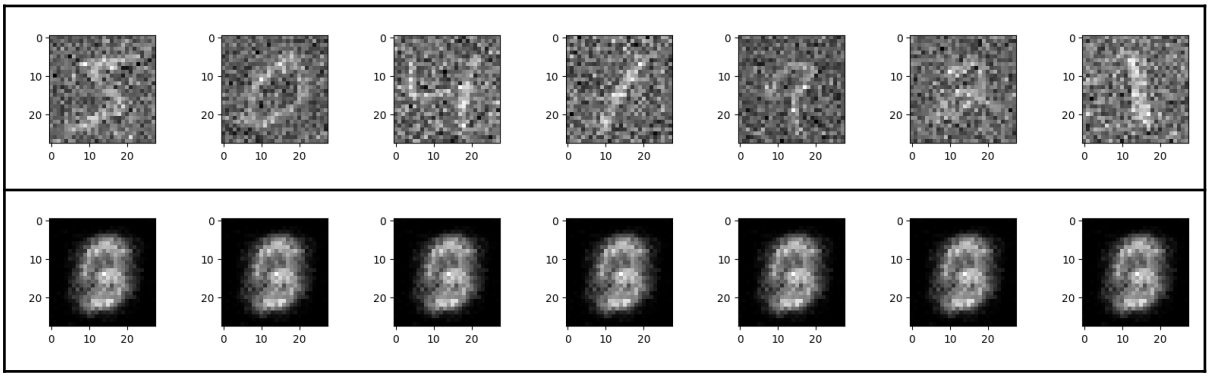
Débruitage des données (bruitées)

Nous allons dans cette partie introduire du bruit à nos données mnist (Handwritten digits), puis appliquer notre auto-encodeur afin de réduire ce bruit (tâche de débruitage des données)

100 itérations, facteur de bruit = 0.5



500 itérations, facteur de bruit = 0.5



Analyse des résultats:

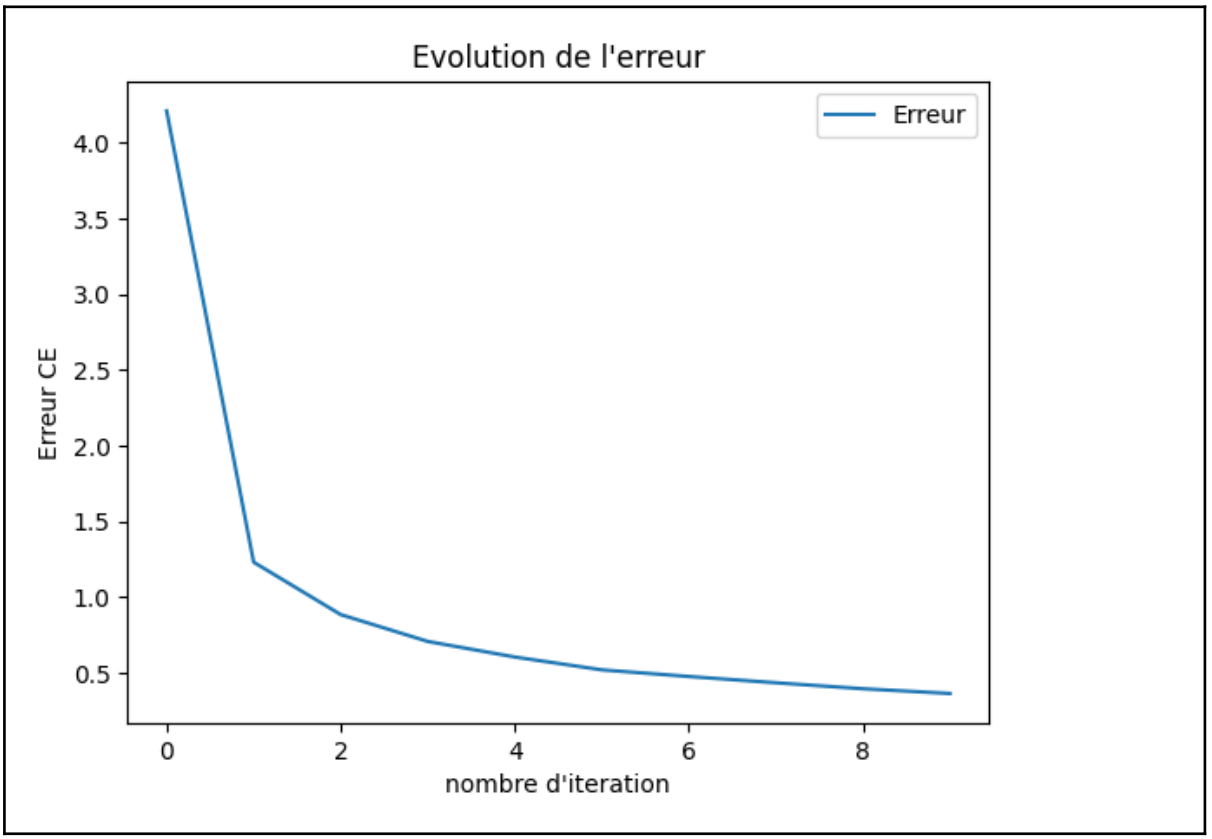
Nous observons que plus le nombre d'itérations est important et plus le reconstruction des données se rapproche des données réelles.

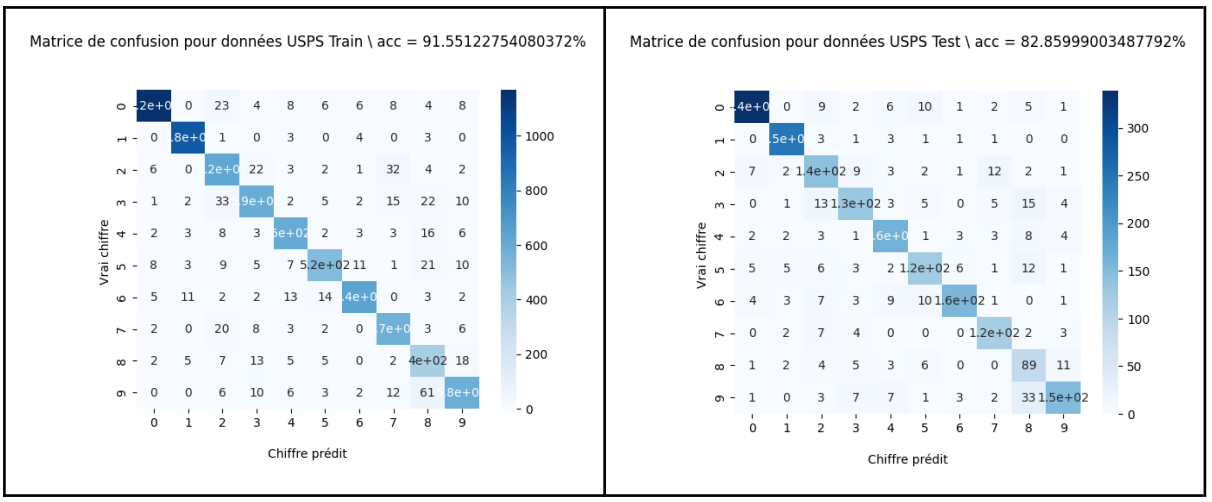
Pour la tâche de débruitage des données, nous observons également que plus le nombre d'itérations est important et plus le débruitage est efficace.

Mon sixième réseau … se convole

La couche convolution est le standard en classification d'image. Elle permet d'appliquer un même filtre sur différentes positions de l'image et de sortir une valeur par position. Les sorties associées à un filtre sont une nouvelle image (de taille différente en fonction du nombre de positions où l'opération de convolution a été effectuée) qui est appelée feature map.

Test de la couche convolution:





Mon tout s'améliore.

Les convolutions 1D implémentées permettent de reconnaître des motifs horizontaux dans les images.
À partir des modules déjà implémentés, il est facile de reconnaître également des motifs verticaux : il suffit pour cela de transposer l'image.

Nous allons donc implémenter un réseau qui reconnaît des motifs verticaux et horizontaux en concaténant la sortie de deux couches convolutionnelles. Nous allons également considérer du Average Pooling (qui fait la moyenne sur la fenêtre temporelle) plutôt que du Max Pooling.

Résultats des expériences: