Sorbonne Université
Master DAC

Projet - Réseau de neuronnes : DIY

Machine Learning

Dan SERRAF, Nikola KOSTADINOVIC

2022

Table des matières

Introduction

Ce projet avait pour objectif de développer une librairie **python** implémentant un réseau de neuronnes. L'implémentation est inspirée des anciennes versions de Pytorch et d'implémentation similaire permettant 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.

Par la suite, nous avons implémenté différent modules tel que le module ***Linear*** nécessaire à la régression linéaire mais aussi les modules ***TanH*** et ***Sigmoide***, permettant de se diriger vers de la non-linéarité.

Nous avons ensuite implémenté la classe **Sequentiel** qui nous permet d'ajouter des modules en série pour ne pas se contenter un travail fastidieux sur les opérations de chaînage entre modules lors de la descente de gradient. Cette classe Sequentiel automatise les procédures de forward et backward quel que soit le nombre de modules mis à la suite.

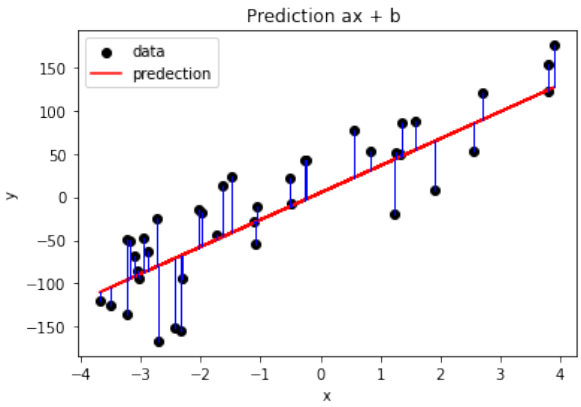
Nous devions ensuite passer au problème de Multi-Classe, pour cela nous avons rajouté une transformation ***SoftMax*** ainsi qu'un coût ***Cross-Entropique***.

Pour finir, nous sommes passés à de l'apprentissage non-supervisé grâce à l'***auto-encodeur***, qui est un réseau de neuronnes dont l'objectif est d'apprendre un encodage des données dans le but de réduire les dimensions. Nous implémentions aussi une couche convolutionnelle ***Conv1d*** et ***Conv2d*** pour de la classification d'images.

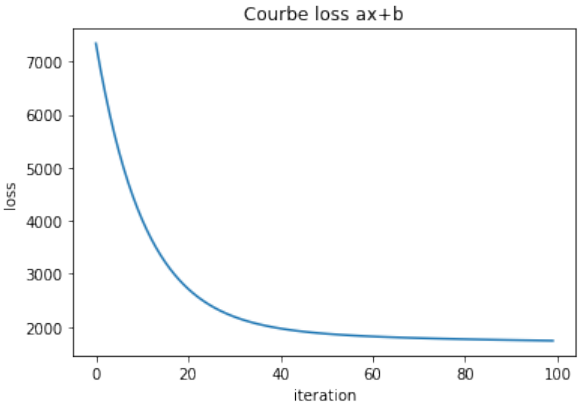
Mon premier est ...linéaire !

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 (Module ***Linear***). Pour ce faire, nous avons implémenté deux classes, la classe **Linear** héritant de **Module** et la classe **MSELoss** héritant de **Loss**.

2.0.1 Premier Test : Régression



(a) Droite obtenue avec le module **Linear** pour 1 seule sortie

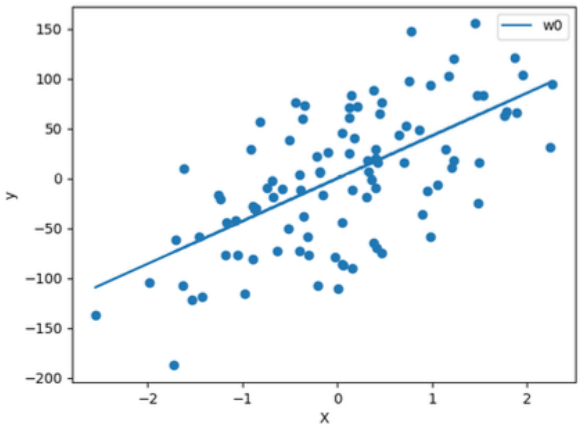


(b) Evaluation de la Loss avec le module **Linear** pour 1 seule sortie

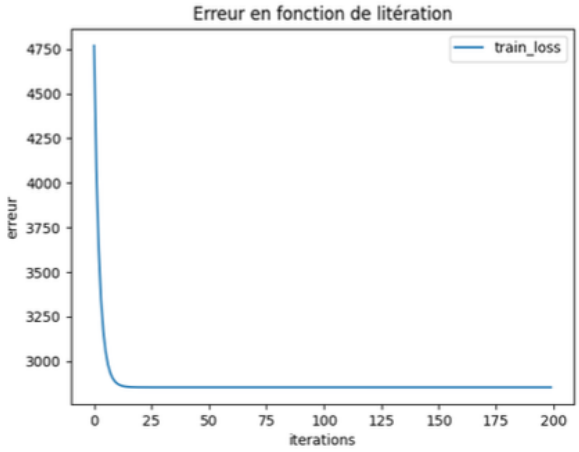
Dans la figure (2.1a), les points sont les données en entrée du réseau de neuronnes et la droite tracée correspond à $f(x) = ax$ où a est le paramètre w optimisé par le réseau. Le résultat trouvé correspond bien à ce que nous cherchions. On peut remarque que la loss décroit assez rapidement, environ 60 itérations pour converger.

Passons maintenant à des tests bruités.

2.0.2 Second Test : Regression with noise



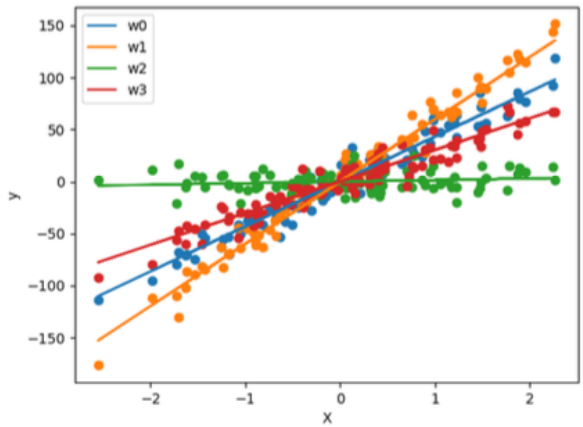
(a) Droite obtenue avec le module **Linear** pour 1 seule sortie et un bruit élevé



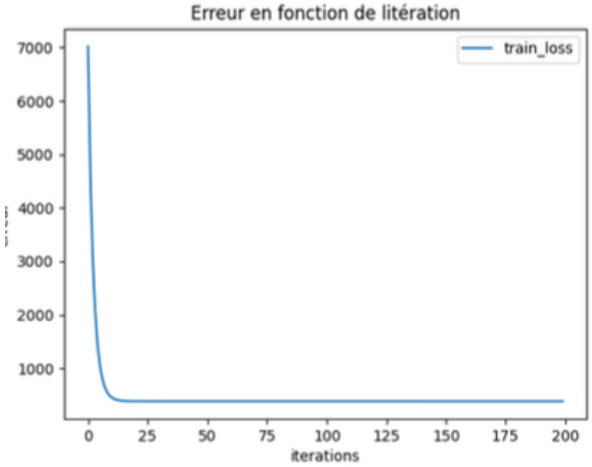
(b) Evaluation de la Loss avec le module **Linear** pour 1 seule sortie et un bruit élevé

Nous pouvons constater ici que même avec un bruit assez élevé, nous convergons assez rapidement ($\approx 20 itrations$).

2.0.3 Troisième Test : 4 Regression with noise



(a) Droites obtenue avec le module **Linear** pour 4 sorties et un bruit léger

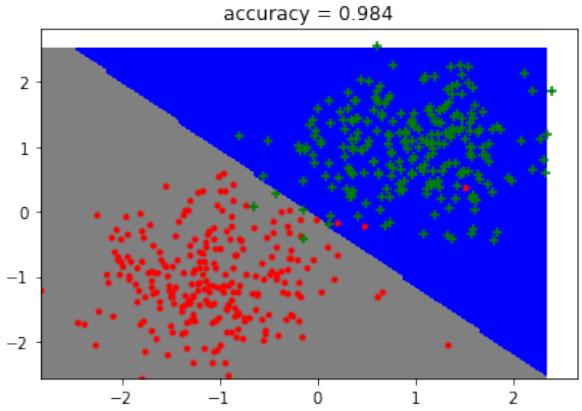


(b) Evaluation de la Loss avec le module **Linear** pour 4 sorties et un bruit léger

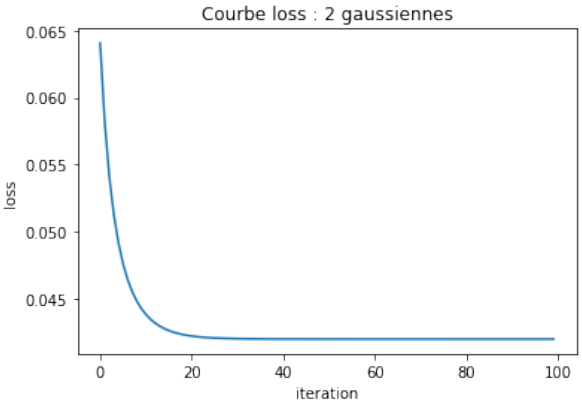
Comme avant, nous voyons que l'erreur diminue en fonction du nombre d'itération et nous avons réussi à tracer les 4 droites optimales assez rapidement ($\approx 15 itrations$).

Passons maintenant à une expérimentation sur de la classification linéaire.

2.0.4 Quatrième Test : Classification



(a) Accuracy



(b) Evaluation de la Loss

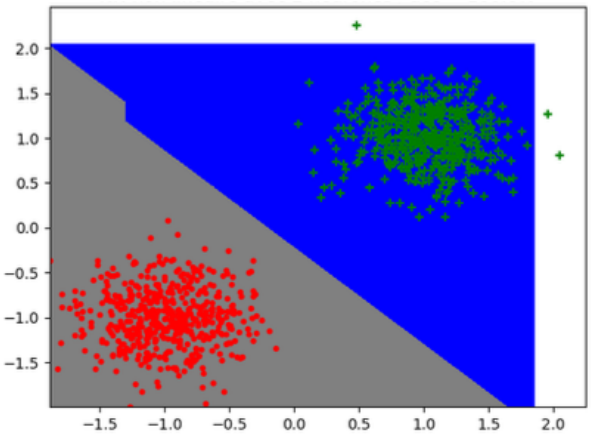
Nous avons ensuite fait un second test, ou nous testions la classification, on s'aperçoit bien avec 2.4a que notre modèle classe quasi parfaitement les données. La convergence quant-a elle est toujours aussi rapide (ici $\approx 17 itrations$).

Mon second est ...non-linéaire !

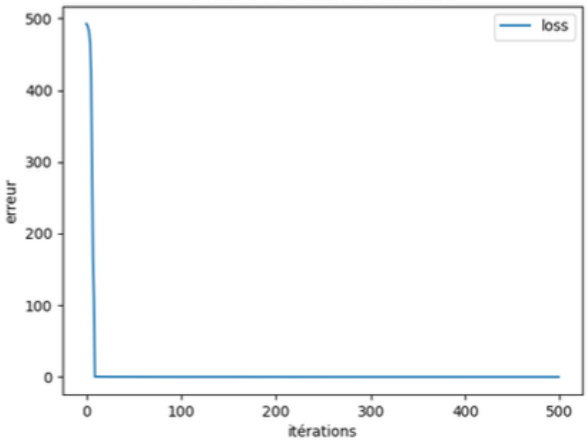
Passons maintenant à l'implémentation d'un réseau de neuronnes non-linéaire avec deux couches cachées et des fonctions d'activation qui sont, respectivement, la tangeante hyperbolique (Module ***TanH***) entre les deux couches et la sigmoide (Module ***Sigmoide***) à la sortie.

Commençons par un test linéairement séparable avec des données générées selon 2 gaussiennes.

3.0.1 Premier Test : Deux Gaussiennes



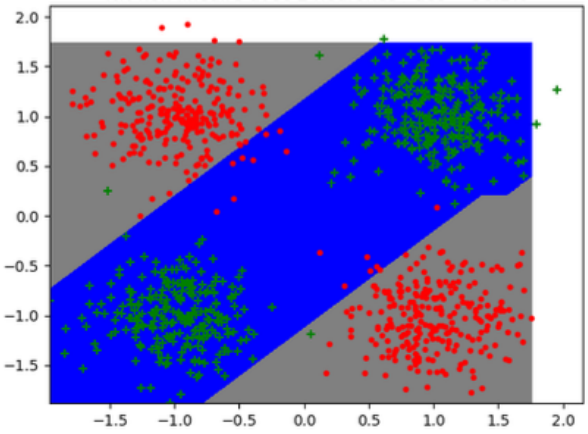
(a) Frontière de décision pour des données linéairement séparable



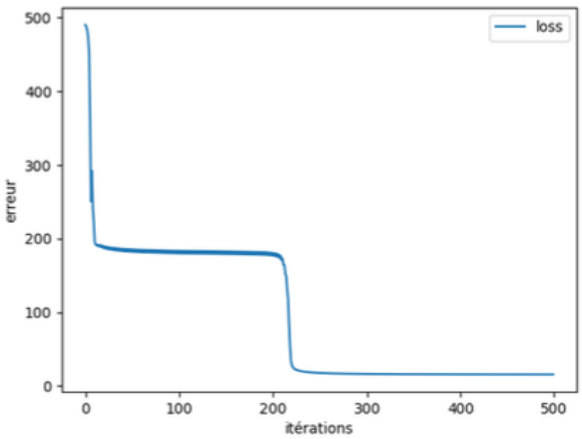
(b) Evolution de la loss

Nous obtenons un taux de bonne classification de 100%. Comme nous pouvons le voir, nous classons parfaitement les données avec notre réseau de neuronnes. Nous obtenons ici une convegence avoisinant les 20 itérations.

3.0.2 Second Test : Quatre Gaussiennes, problème du XOR



(a) Frontière de décision pour le **XOR**

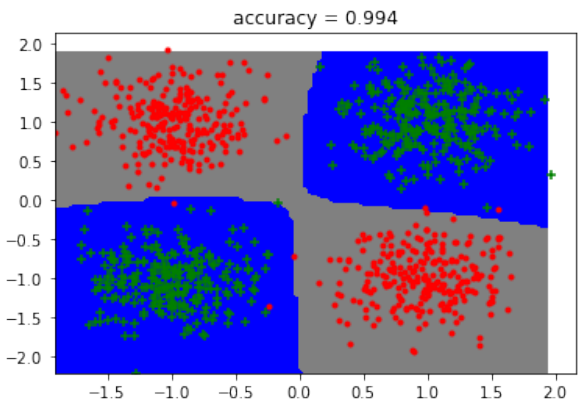


(b) Evolution de la loss

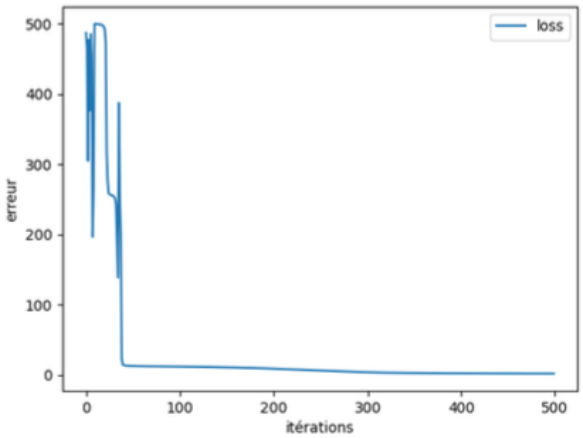
Nous obtenons un taux de bonne classification de 98.4%. Nous pouvons aussi observer la Loss chutée très fortement au début puis converger à partir de la $\approx 200^{ieme} itrations$.

Passons maintenant, toujours au problème du XOR, mais maintenant, nous allons le résoudre avec 10 neuronnes.

3.0.3 Troisième Test : Problème du XOR avec 10 neuronnes



(a) Problème du **XOR** avec 10 neuronnes



(b) Evolution de la Loss

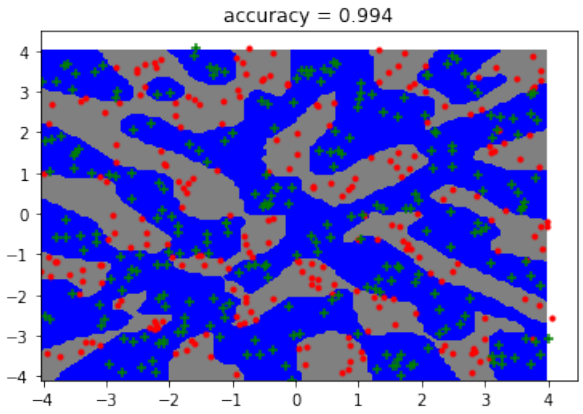
On obtient ici, un taux de bonne classification de 99.4% Concernant l'erreur, elle fluctue beaucoup avant avant de converger au alentours des 60 itérations. On peut donc supposer que l'augmentation du nombre de neuronnes permettent la meilleure précision de notre modèle.

Mon troisième est un encapsulage

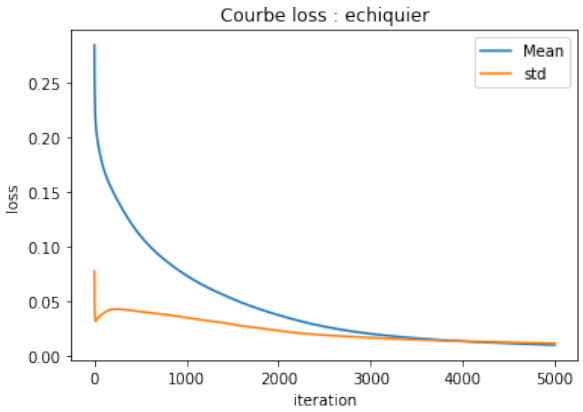
Pour commencer, nous avons implémenté deux classes très importantes, Sequentiel et Optim, nous permettant d'automatiser le processus. Nous les avons testé en utilisant les mêmes paramètres que pour la partie ***Mon Second est ...non-linéaire***, nous obtenons bien des résultats identiques, ce qui confirme la bonne implémentation des classes.

4.0.1 Test : Échiquier with Optim

Ici, nous utilisons nos classe Sequentiel et Optim pour essayer de résoudre le problème de l'échiquier



(a) Frontière de décision pour le problème de l'échiquier avec Optim

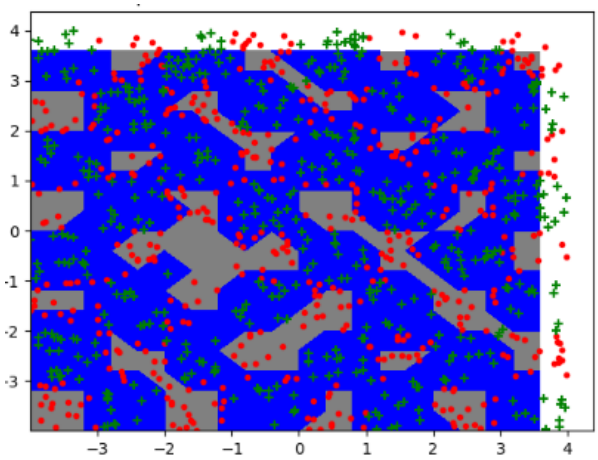


(b) Evaluation de la Loss

On obtient ici, un taux de bonne classification de 99.4% Nous avons presque un taux de 100% ainsi qu'une loss quasiment nul. On remarque que plus on a un réseau profond, plus le taux de bonne classification augmente.

Nous avons ensuite implémenté une fonction ***SGD*** qui prend en entrée entre autre un réseau, un jeu de données, une taille de batch et un nombre d'itération et s'occupe du découpage en batch du jeu de données et de l'apprentissage du réseau pendant le nombre d'itérations spécifié.

4.0.2 Test : Échiquier with SGD



(a) Frontière de décision pour le problème de l'échiquier avec SGD

On voit que l'impact de la taille du mini batch (taille du batch / 10) à une importance sur la précision de notre modèle avec 99.9% taux de bonne classification contre 99.4% en 4.1a.

Mon quatrième est multi-classe

Nous allons maintenant nous intéressé au cas multiclasse, en effet, jusqu'a présent nous n'avons traité que les problèmes binaires. 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 allons prédire la classe qui maximise les probabilités.
Le coût utilisé est le coût Cross-Entropique qui s'écrit :

$$CE(y, \hat{y}) = -\hat{y}_y + \log \sum_{i=1}^{K} e^{\hat{y}_i}$$

Notre réseau est maintenant composé d'une seule couche caché, avec pour fonctions d'activations :

— une tangeante hyperbolique (Module ***TanH***)
— un softmax (Module ***SoftMax***)

Dans la continuité de la partie précédente, nous avons fait en sorte que le réseau soit capable de gérer les problèmes multi-classes. C'est à dire distinguer les différentes classes parmi un ensemble de plus de 2 classes. Pour se faire nous avons utilisé les données MNIST. Pour vérifier cette propriété nous avons testé si l'algorithmes était capable de reconnaître les 9 classes de chiffres entre-elles.

5.0.1 Paramètres de l'experience

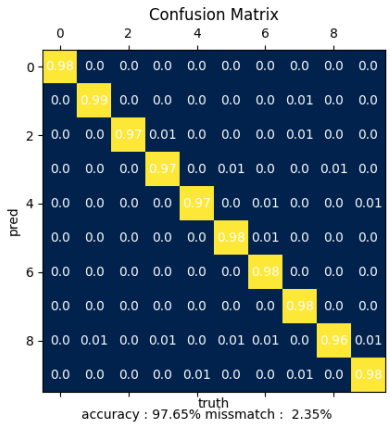
le réseau utilisé pour cette experience :

— 1 : **input** = 256 , **output** = 256 , **fonction d'activation** = Tanh
— 2 : **input** = 256 , **output** = 16 , **fonction d'activation** = Tanh
— 3 : **input** = 16 , **output** = 10 , **fonction d'activation** = Sigmoïde

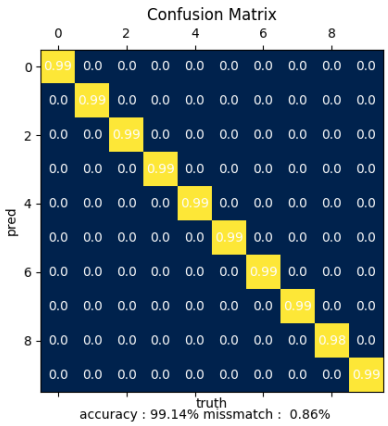
Les hyper-paramètres utilisés sont : $\epsilon = 1e-3$ $max_{iterations} = 100$ Loss utilisée : BCE.

Dans le cas présent, l'input représentera un pixel d'une image de MNIST de dimension 16 par 16. L'output du réseau est une couche renvoyant l'énergie prédite pour chaque classe. Pour la prédiction, à la différence d'un classifier binaire où on regarde le maximum des deux sorties, ici on considèrera l'argmax pour avoir l'intuition de la classe disposant de la plus haute énergie pour un exemple donné.

5.0.2 10 VS 10 : stratégie OVR

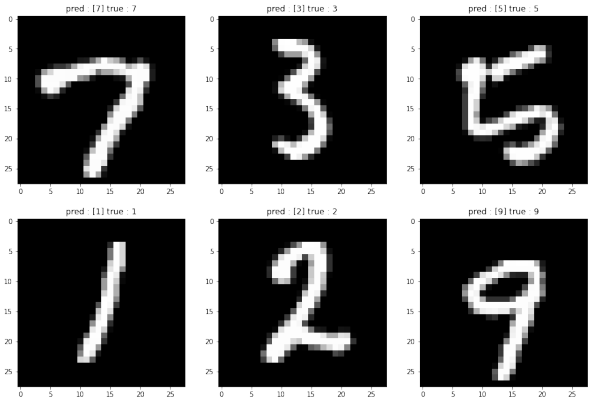


(a) Matrices de confusion de Test

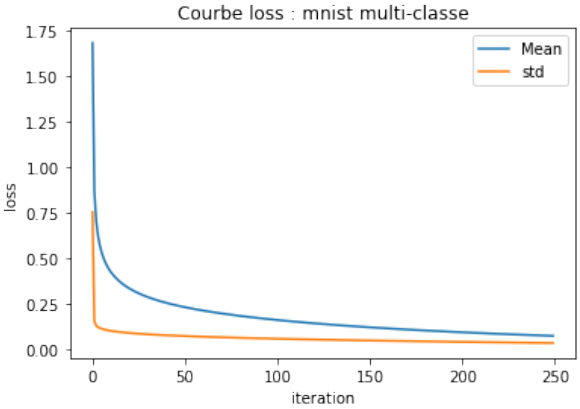


(b) Matrices de confusion de Train

Nous obtenons respectivement 97.57% et 99.04% d'accuracy.



(c) Accuracy



(d) Evaluation de la Loss

Nous obtenons ici, un taux de bonne classification de 94,36%

Dans ce cas ci, une MSE n'aurait pas pu nous permettre d'avoir des résultats aussi précis. La BCE ou **binary cross-entropy** utilise une stratégie de type One Versus Rest alors qu'une MSE sans aucune modification se serait basée sur la puissance renvoyée par chaque classe pour faire le calcul de la loss et la **back-propagation**. La *binary cross-entropy* ou la *cross-entropy* est un type d'entropie très utilisée pour la classification multi-classe notamment dans **Pytorch** puisque son utilisation permet, par l'utilisation d'un softmax, de noramliser l'énergie prédite dans chaque classe.

Mon cinquième se compresse

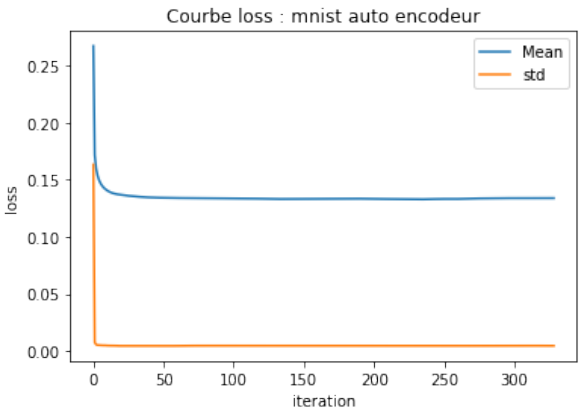
6.0.1 Premier Test : Auto-Encodeur

La 5ème partie de ce projet a conduit à mettre en oeuvre un encodeur et un décodeur. Nous avons réalisé plusieurs experiences à partir des données MNIST afin de pouvoir :

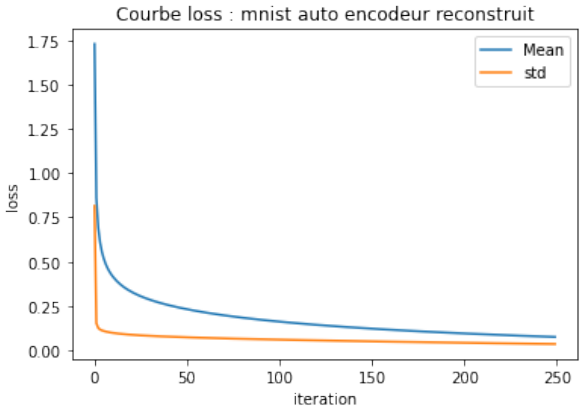
- étudier l'influence de la compression puis décompression sur la qualité des images.
- observer les performances en débruitage d'une compression puis décompression d'images.

La taille des images utilisées est 28x28 soit 789 pixels.

(a) Auto-Encodeur
Architecture :256->100N->10N



(a) Evaluation de la Loss Auto-Encodeur



(b) Evaluation de la Loss
Auto-Encodeur_Reconstruit

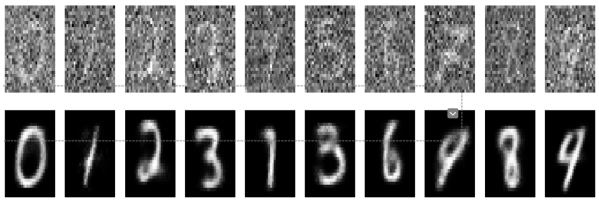
On obtient ici, un taux de bonne classification de 81.12%

Nous avons utilisé la loss **BCE** plutôt que **MSE** car nous avions remarqué qu'elle permettait d'observer plus rapidement des images avec un tracé "net".

On a pu remarquer que plus la taille de la compression est petite, plus l'algorithme a tendance à mélanger les classes qui se ressemblent. Lorsque la taille de la compression augmente on observe l'apparition de nouvelles classes et l'augmentation de la précision du tracé des classes déjà existantes. Mais aussi, qu'a partir d"une certaine taille de compression la qualité de la compression n'est plus visible à l'oeil nu.

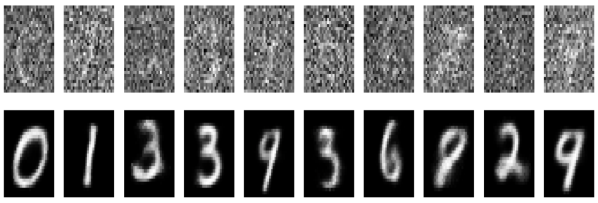
Passons maintenant à la compression et décompression d'images avec des données bruitées. Nous allons appliquer deux types de bruits, un bruit **gaussien** et un bruit **salt and pepper**. Nous allons essayer jusqu'à quel degré nos modèles peuvent retirer les informations encombrantes.

6.0.2 Gaussian Noise



(a) Images bruitées **(gaussien 0.2)**

Avec un bruit peu présents nous arrivons à extraire les chiffres avec une certaine difficulté pour certaine valeur.

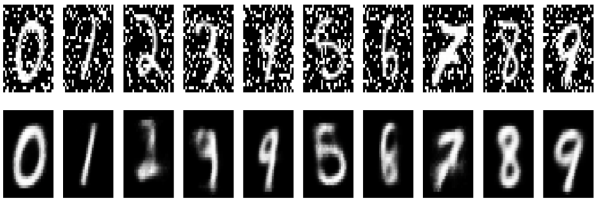


(a) Images bruitées **(gaussien 0.8)**

Avec un bruit beaucoup plus présent, on peut directement voir les difficultés, tout d'abord on remarque des chiffres qui se répète. Cela nous permet aussi d'en conclure que certains chiffres ont des patterns qui sont assez proches.

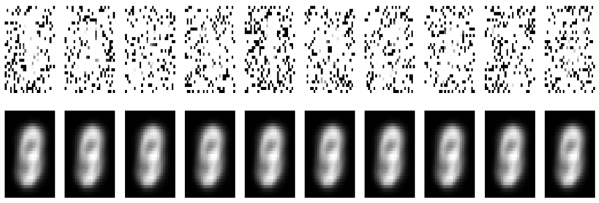
6.0.3 Salt and Pepper Noise

Passons maintenant au bruit **Salt and Pepper**.



(a) Images bruitées **(salt and pepper 0.2)**

On constate qu'en générale, on arrive à distinguer quasiment toute les valeurs.



(a) Images bruitées **(salt and pepper 0.8)**

Ici nous remarquons qu'il est bien trop difficile de distinguer des chiffres et d'en extraire des informations. Cela est pareil pour un bruit à 60%. On en conclue qu'il faudrait augmenter le nombre d'epochs et/ou de rajouter des données.

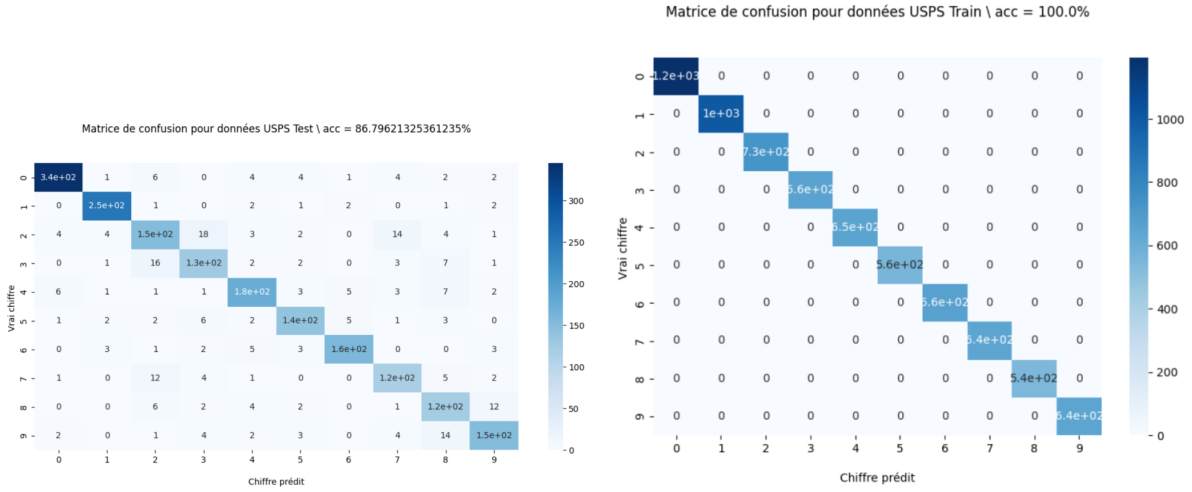
Mon sixième se convole

Le CNN est une méthode d'apprentissage développée pour la classification et la reconnaissance d'images. Largement utilisé dans le domaine de la reconnaissance des expressions faciales aujourd'hui car il permet de réduire la complexité du modèle et extraire avec précision les caractéristiques de l'image en utilisant plusieurs couches (Convolution, ReLu, Pooling..) qui vont être développer dans la section précédente.

Pour commencer, nous avons implémenté les différents modules de la partie convolution de l'énoncé ; à savoir Conv1D, MaxPool1D, Flatten et ReLU.

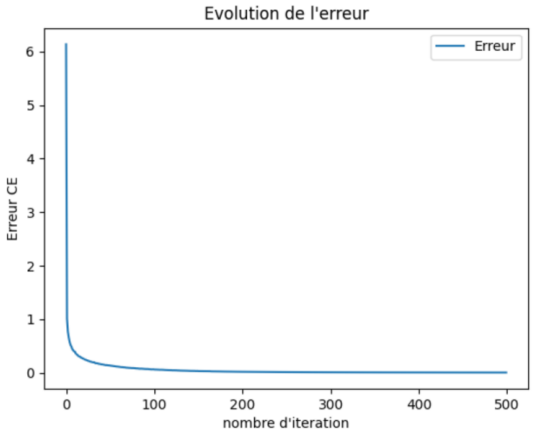
7.0.1 Premier Test : Convolution 1D : MaxPool

Les paramètres utilisés sont : $nbiter = 500$, $gradient\_step = 1e^{-3}$, $batch\_size = 1/300$



(a) Matrice de Confusion en Test

(b) Matrice de Confusion en Train

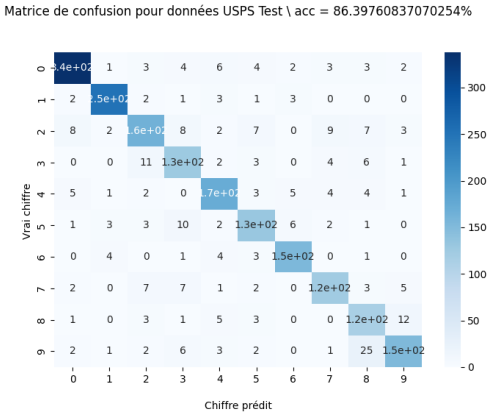


(a) Evaluation de la Loss

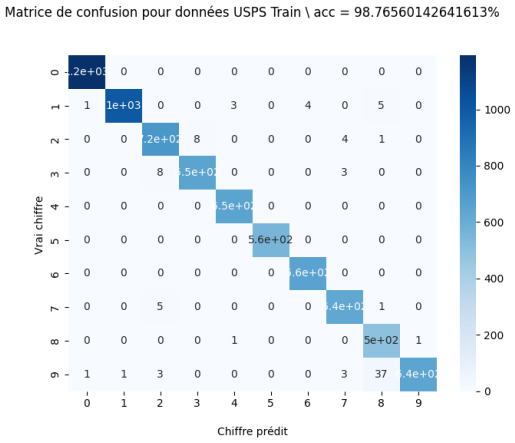
Ici le coût utilisé est : $CELogSoftMax$

7.0.2 Second Test : Convolution 1D : AvgPool

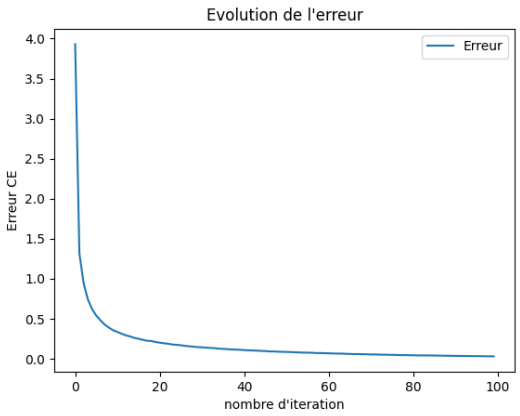
Les paramètres utilisés sont : $nbiter = 100$, $gradient\_step = 1e^{-3}$, $batch\_size = 1/10$



(a) Matrice de Confusion en Test



(b) Matrice de Confusion en Train



(a) Evaluation de la Loss

Ici le coût utilisé est : $CELogSoftMax$

Concernant les convolutions, on constate que des convolutions uniquement en 1 dimension permettent déjà d'avoir de très bons résultats.

Nous pouvons aussi bien constater la différence entre les convolutions avec les filtres horizontaux et les filtres horizontaux-verticaux, cette différence est bien nette, et c'est bien ce qui était prévu.

Nous pouvons aussi noter la différence entre lee MaxPooling et le AvgPooling, qui est, la loss. En effet, la loss du MaxPooling converge assez rapidement en chutant rapidement au début, puis continuer à converger petit à petit vers 0, alors que la loss du AvgPooling converge lentement mais de manière constante.