Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - Données Apprentissage et Connaissances

Machine Learning

Rapport de projet

Réseaux de neurones : DIY

Réalisé par :
TOUKAL Raouf
BENHADDAD Sabrina

Mai 2023

Table des matières

Table des figures

Introduction

Les réseaux de neurones sont une famille de modèles d'apprentissage automatique qui ont révolutionné le domaine de l'intelligence artificielle au cours des dernières décennies. Inspirés par le fonctionnement du cerveau humain, les réseaux de neurones sont capables d'apprendre à partir de données brutes, sans avoir besoin d'une programmation explicite. Puissants et polyvalents, ces derniers ont par ailleurs, permis des avancées significatives dans de nombreux domaines, de la reconnaissance d'images à la traduction automatique en passant par la prédiction de séquences temporelles.

L'objectif de ce projet est d'implémenter un réseau de neurones version modulaire. Nous allons en d'autres termes, construire et entraîner nos propres réseaux de neurones à l'aide des modules suivants :

1. **Le module linéaire**
2. **Le module non-linéaire**
3. **Le module séquentiel**
4. **Le module Multi-classe**
5. **Le module Auto-encodeur**
6. **Le module convolutionel**

A noter que l'implémentation est inspirée des anciennes version de *pytorch* et des implémentations analogues qui permettent d'avoir des réseaux génériques très modulaires. Ainsi, chaque couche du réseau est vue comme un module. Un réseau est donc constitué d'un ensemble de modules, en particulier les fonctions d'activation.

Chapitre 1

Module linéaire

Pour cette première étape, nous allons effectuer une tâche d'apprentissage à l'aide d'un module linéaire.
Ce dernier va nous permettre autrement dit, d'appliquer une transformation linéaire aux entrées à l'aide d'une régression sur les données : $y = \alpha x + \beta$ (y varie linéairement par rapport à x).

Cette section portera par conséquent sur les différentes évaluations et expérimentations requises du module. L'objectif étant ainsi, de valider son fonctionnement.

1.1 Expérimentations

1.1.1 Comparaison avec scikit-learn

Notre première phase de tests consiste en la comparaison de notre module linéaire avec le module linéaire de scikit-learn à savoir : *Linear Regression*[1]

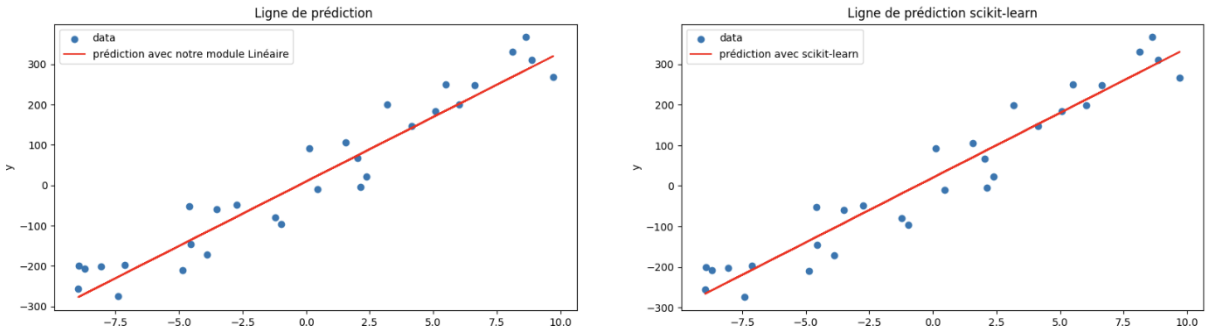

Figure 1.1 – Comparaison de notre module linéaire avec le module scikit-learn

Nous pouvons remarquer ainsi grâce à la figure, que les deux droites sont exactement les mêmes.

1.1.2 Évolution du coût MSELoss

L'objectif de cette évaluation est de suivre visuellement l'évolution de la fonction de coût "MSELoss" au fil des itérations lors de l'apprentissage par descente de gradient.

1. https ://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Nous pouvons constater que le coût chute drastiquement et atteint son minimum au bout de 20 itérations. Il converge donc assez rapidement.

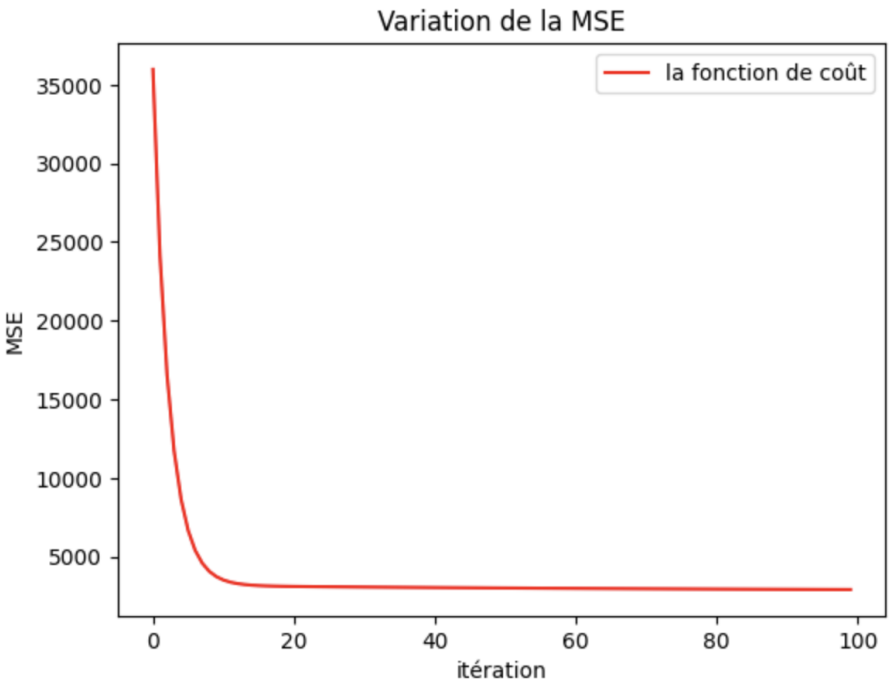

FIGURE 1.2 – Variation de la fonction de coût MSE

Il est important de souligner tout de même que les données utilisées lors de ces expérimentations sont linéairement séparables.
Ainsi, bien que le module linéaire puisse donner des résultats prometteurs, il n'est malheureusement pas applicable à tous les types de données.

C'est notamment le cas lorsque ces dernières ne peuvent pas être séparées de manière linéaire. C'est principalement pour cette raison que nous prévoyons de réaliser d'autres implémentations visant à résoudre ce problème et à généraliser donc l'utilisation de nos modules.

Chapitre 2

Module non-linéaire

Dans cette partie, nous allons nous intéresser davantage sur la classification binaire des données non linéairement séparables.
En effet, nous allons tester l'implémentation de notre module non-linéaire sur les deux types de données à savoir : les linéairement séparables et les non linéairement séparables, générées à l'aide de la fonction *gen_arti*[1].

2.1 Données linéairement séparables

La figure ci-dessous a été générée en utilisant le paramètre *type=0* de la fonction.
Étant donné que seul un hyperplan est nécessaire, un seul perceptron est suffisant pour classifier ce type de données.
Nous avons utilisé par conséquent, une seule couche linéaire avec une sortie unique, suivie d'une fonction sigmoide.

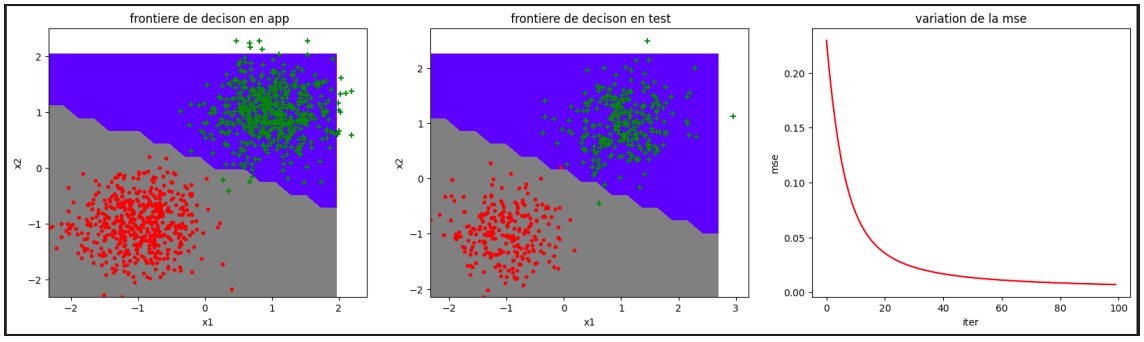

Figure 2.1 – Classification données linéairement séparables

La frontière de décision sépare très bien les données linéairement séparables.

2.2 Données non-linéairement séparables (type XOR)

La figure ci-contre a été générée quant à elle, en utilisant le paramètre *type=1* de la fonction précédente.
En ce qui concerne ce type de données (XOR), 2 perceptrons seront nécessaires pour cette classification binaire.

1. Le code de la fonction figure dans les scripts python joints au rapport

Étant donné le bruit, nous avons utilisé 4 sorties suivies d'une couche $tanH$, suivie d'une couche linéaire avec pour finir, une sortie suivie d'une sigmoide.

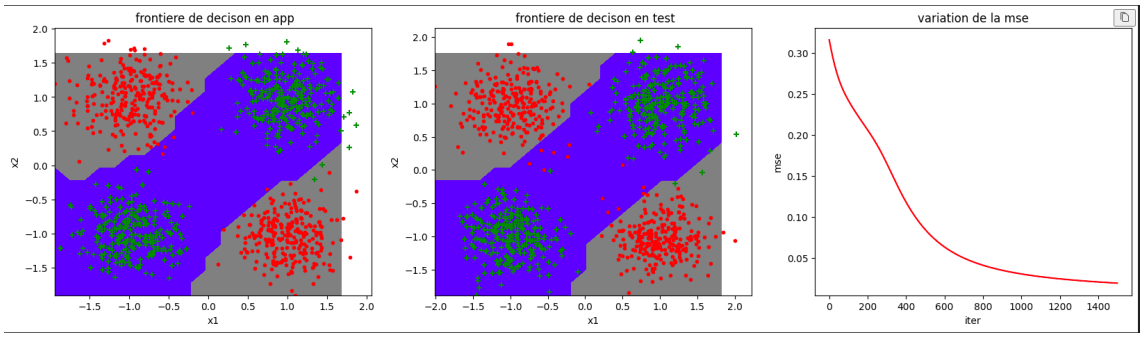

FIGURE 2.2 – Classification données non-linéairement séparables (de type XOR)

Nous pouvons remarquer que la frontière n'est en effet pas linéaire lorsqu'il s'agit de données de type XOR.
Nous arrivons toutefois à une très bonne classification avec une accuracy de 0.977

Chapitre 3

Module séquentiel

Dans cette nouvelle partie, nous développons un module séquentiel permettant d'ajouter les modules en série et de superposer les couches de notre réseau.
Nous implémentons également des fonctions supplémentaires s'inspirant de la bibliothèque *pytorch*[1] à l'image de SGD et OPTIM, dans le but d'optimiser notre modèle.

En ce qui concerne le processus d'expérimentation, nous commencerons par visualiser les frontières de décision générées par notre modèle. Ensuite, nous procéderons à des tests sur les fonctions d'optimisation en évaluant l'impact de la taille du lot (batch size) sur nos modèles.

3.1 Expérimentations

XOR Les expérimentations sont menées sur des données de *type=1*, autrement dit sur le même réseau utilisé précédemment.

Les tests quant à eux ont été réalisés en utilisant un batch stochastique, un batch et un mini-batch de taille 100, comme le montrent les figures ci-dessous.

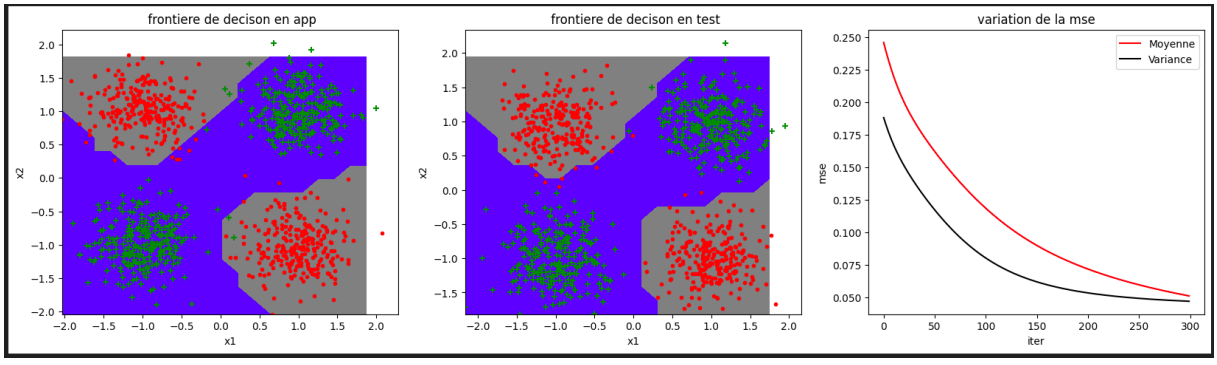

Figure 3.1 – Frontière XOR avec un nombre d'exemple du batch = 100

1. https ://pytorch.org/

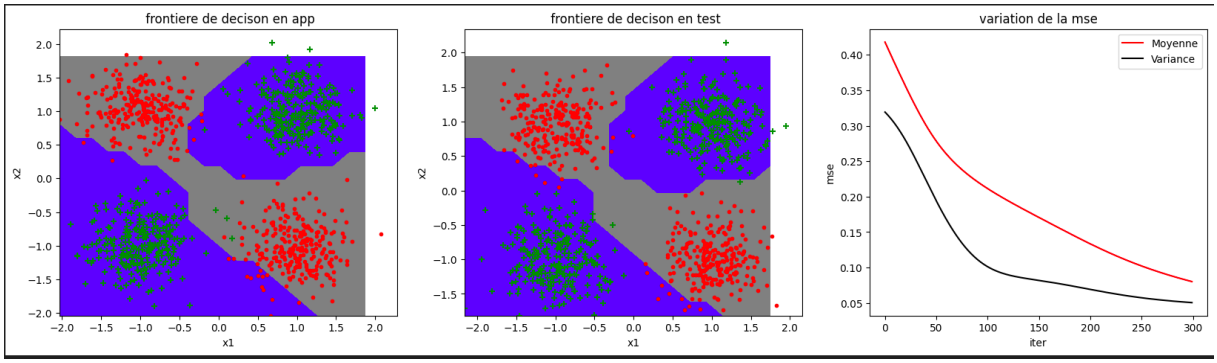

FIGURE 3.2 – Frontière XOR avec un nombre d'exemple du mini-batch = 100

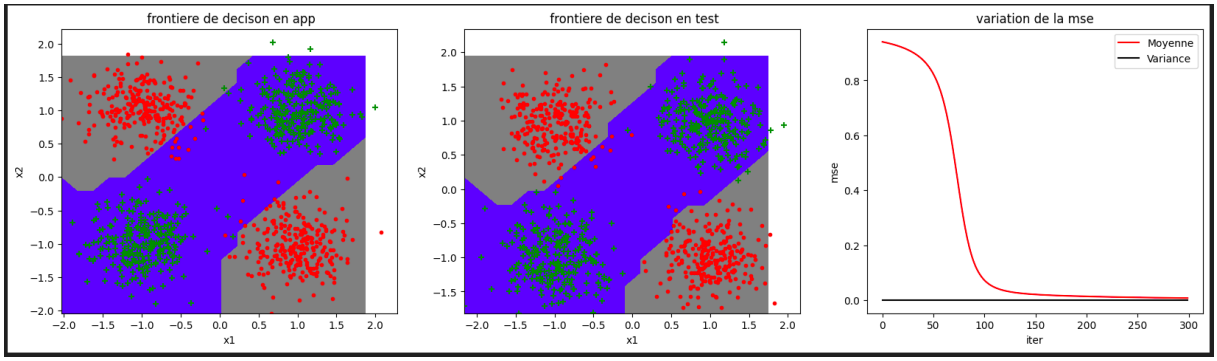

FIGURE 3.3 – Frontière XOR avec un batch stochastique

Échiquier Dans cette partie dédiée à l'échiquier, nous avons mené nos expérimentations en utilisant des données de *type = 2*.
Les figures suivantes présentent les résultats du plot de notre modèle séquentiel pour les trois types de batchs mentionnés précédemment.

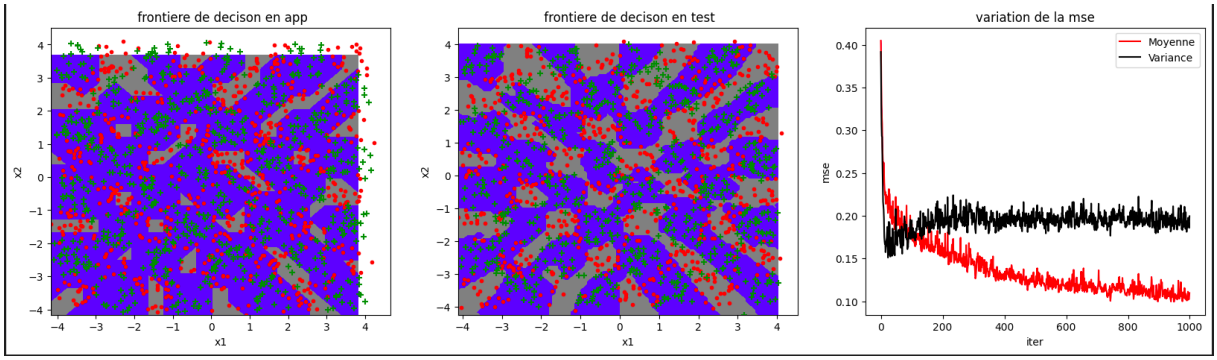

FIGURE 3.4 – Frontière échiquier avec un nombre d'exemple du batch = 100

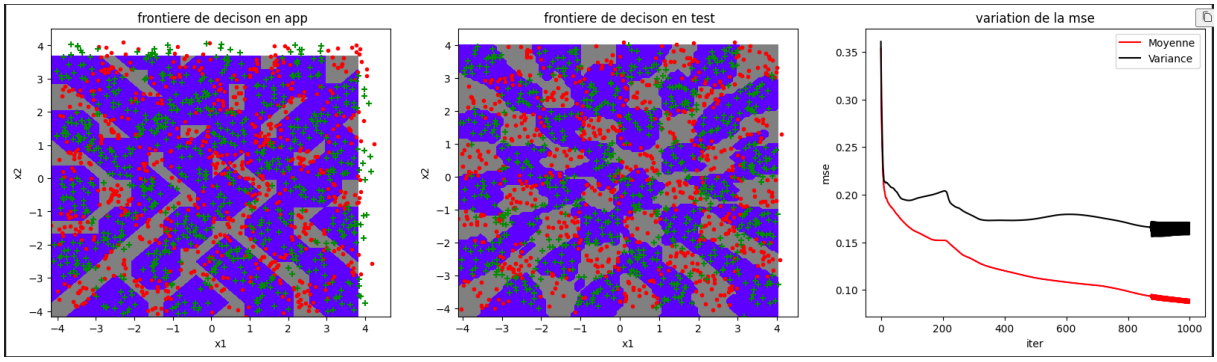

FIGURE 3.5 – Frontière échiquier avec un nombre d'exemple du mini-batch = 100

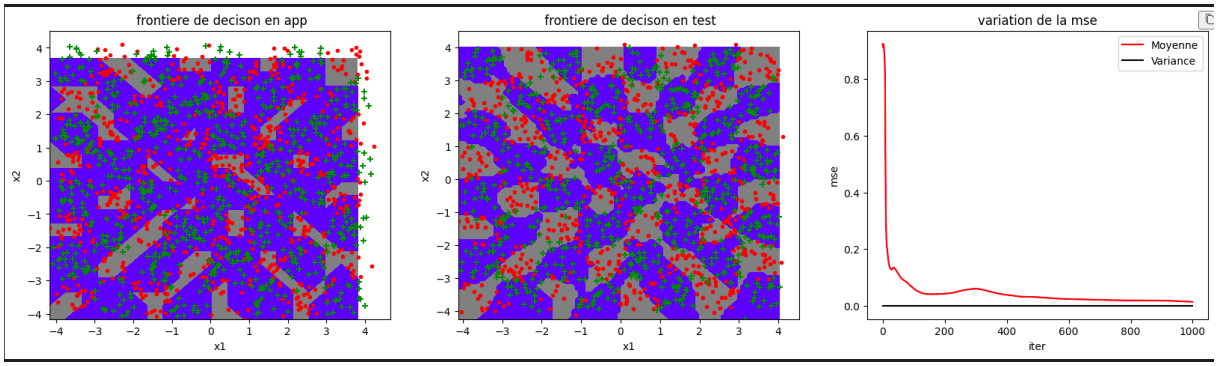

FIGURE 3.6 – Frontière échiquier avec un batch stochastique

Remarques

Nous pouvons observer dans un premier temps, que le batch stochastique est moins sensible au bruit car sa variance reste stable et égale à zéro pour les deux problèmes.

En comparant le batch et le mini-batch, nous constatons que pour le problème de l'échiquier, la variance augmente légèrement dans le batch, tandis qu'elle décroit dans le mini-batch. De plus, la variance dans le mini-batch est plus petite que dans le batch. Pour le problème du XOR, la variance dans le mini-batch chute plus rapidement qu'avec le batch, ce qui suggère que le mini-batch est plus stable que le batch.

Cependant, il convient de noter que plus nous nous rapprochons du stochastique, plus l'apprentissage devient lent. Cela peut être un compromis à prendre en compte lors du choix de l'approche d'apprentissage.

Chapitre 4

Module Multi-classe

Nous nous sommes concentrés jusqu'à présent sur des problèmes de classification binaire. L'objectif de cette section est d'explorer un exemple de classification à plusieurs classes.
Cela nous permettra d'élargir notre compréhension des réseaux de neurones et de leur capacité à résoudre des problèmes de classification plus complexes.

Pour ce faire, nous utiliserons la fonction d'activation Softmax $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ pour $i = 1, 2, ..., K$. Nous devons également utiliser une fonction de coût adapté à l'image de la fonction *Cross Entropy* $CE = -\sum_{c=1}^{M} y_c log(p_c)$.

4.1 Expérimentations

Les expérimentations à venir seront réalisées sur les données USPS [1]. Elles seront structurées en trois étapes essentielles :

Première partie

Nous avons testé pour commencer un réseau neuronal simple avec une couche cachée de 124/64, ainsi qu'un réseau plus profond et un réseau plus dense.
Nous avons comme mentionné plus haut utilisé les fonctions d'activation log softmax pour la classification et avons comparé deux fonction de coût : la cross-entropie et la MSE (erreur quadratique moyenne). L'apprentissage quant à lui a été effectué en utilisant des batchs avec 100 epochs.

	mse train	mse test	log softmax ce train	log softmax ce test
128/64	0.962694	0.908321	0.896585	0.845541
dense	0.088465	0.073244	0.738308	0.704534
profond	0.779454	0.735426	0.539844	0.528151

Figure 4.1 – Résultats des deux réseaux dense et profond

1. Données utilisées en TME

	moyenne
mse train	0.610204
mse test	0.572330
log softmax ce train	0.724912
log softmax ce test	0.692742

FIGURE 4.2 – Résultats du réseau simple

Nous constatons d'après les résultats obtenus ci-dessus que le réseau neuronal simple est meilleur que les deux autres modèles pour les deux fonction de coût, en raison de l'évaporation du gradient moins prononcée.
De plus, la moyenne de la cross-entropie s'est révélée être une meilleure mesure de performance que la MSE dans notre cas.

Seconde partie

Dans la deuxième partie de nos expérimentations, nous avons fait varier le nombre d'epochs pour le réseaude neurones avec une structure de (124/64). Nous avons fixé le taux d'apprentissage à $1e-4$.

	train	test
100	0.850363	0.801694
500	0.955150	0.891878
1000	0.973666	0.907823
5000	0.986147	0.915296
10000	0.989028	0.916791

FIGURE 4.3 – Résultats liés à la variation du nombre d'epochs

La figure 4.3 montre bien qu'en augmentant le nombre d'epochs avec un apprentissage par batch, nous ajoutons de la stabilité (chose qui manquait aupravant). Selon les résultats du tableau, il est notifiable que l'augmentation du nombre d'itérations risque de conduire à un surapprentissage. Entre 5000 et 10000 epochs, les gains obtenus sont minimes, ce qui suggère que nous nous approchons de la limite de performance du modèle.

Il est important de noter tout de même que l'augmentation du nombres d'epochs doit être réalisée avec précaution pour éviter le surapprentissage. Il est effectivement nécessaire

de trouver un équilibre entre l'ajout de stabilité et la capacité du modèle à généraliser efficacement les données.

Troisième partie

Dans la troisième et dernière partie, nous avons fait varier le pas d'apprentissage (learning rate) tout en maintenant le nombre d'epochs à 1000 avec batch.

Nous observons que lorsque le pas d'apprentissage est trop petit, le modèle ne converge pas efficacement vers le minimum de la fonction de coût. Cela est dû au fait que les pas effectués lors de la mise à jour des poids sont très petits, rendant difficile l'atteinte du minimum en un nombre d'epochs limité.
Avec un pas d'apprentissage trop grand en revanche, le modèle diverge et ne parvient pas à trouver un minimum stable.

Après avoir examiné les résultats du tableau ci-dessous, nous constatons que les meilleurs taux d'apprentissage étaient de l'ordre de $1e-3$ et $1e-4$. Ces valeurs ont effectivement permis d'obtenir une convergence stable du modèle sans le faire diverger.

	train	test
1e-1	0.088465	0.073244
1e-2	0.100261	0.098655
1e-3	0.983130	0.903837
1e-4	0.970100	0.906328
1e-5	0.840077	0.807673
1e-6	0.636264	0.611858
1e-7	0.098066	0.082212
1e-8	0.143190	0.141006
1e-9	0.052531	0.058794
1e-10	0.133315	0.124564

FIGURE 4.4 – Résultats liés à la variation du learning rate

Il est donc crucial de trouver le bon équilibre du taux ou pas d'apprentissage pour garantir une convergence efficace et stable du modèle.

Voici les résultats du modèles avec les paramètres suivants : learning rate à $1e-4$, un nombre d'epochs fixé à 1000 et un réseau de neurones structuré comme suit : 124 neurones

dans la couche cachée et 64 dans la couche de sortie.

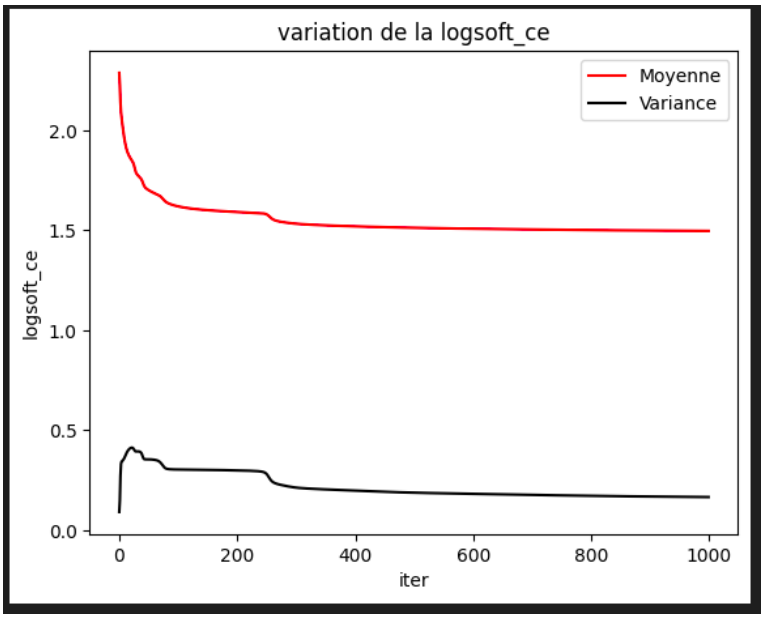

FIGURE 4.5 – Variation de la logsoft_CE

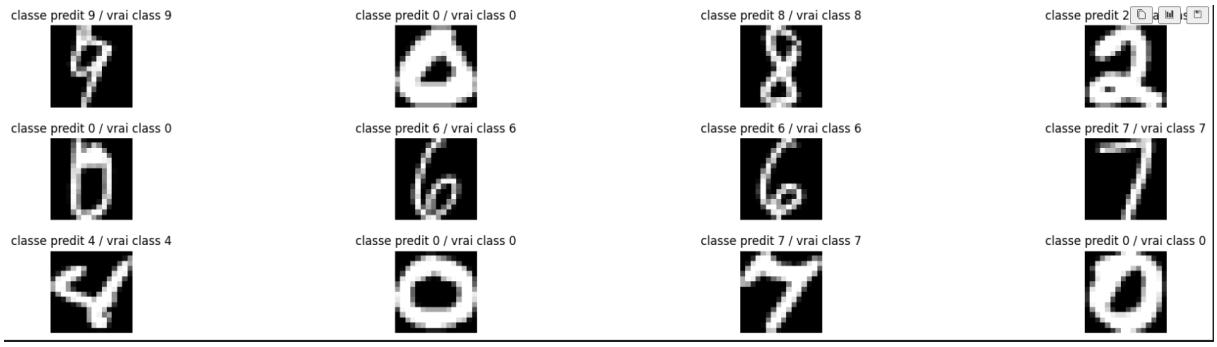

FIGURE 4.6 – Prédictions sur données USPS avec un pas $= 1e-4$

Chapitre 5

Module Auto-encoder

Dans cette partie, nous nous concentrerons sur la construction d'un auto-encodeur, un algorithme non supervisé qui vise à apprendre une représentation compressée des données et à les reconstruire.
Notre objectif sera de réduire les dimensions des données en comprimant leur représentation.

5.1 Expérimentations

Nous utiliserons les données MNIST [1] pour mener différentes expérimentations. Ces expérimentations nous permettront d'explorer les performances de l'auto-encodeur en fonction de différents paramètres et architectures, et d'évaluer sa capacité à reconstruire efficacement les images MNIST.

5.1.1 Reconstruction de données non bruitées avec un auto-encoder

Dans cette première partie, nous avons réalisé des expérimentations sur plusieurs auto-encodeurs symétriques en variant le nombre de couches, le nombre de sorties et les fonctions d'activation utilisées.

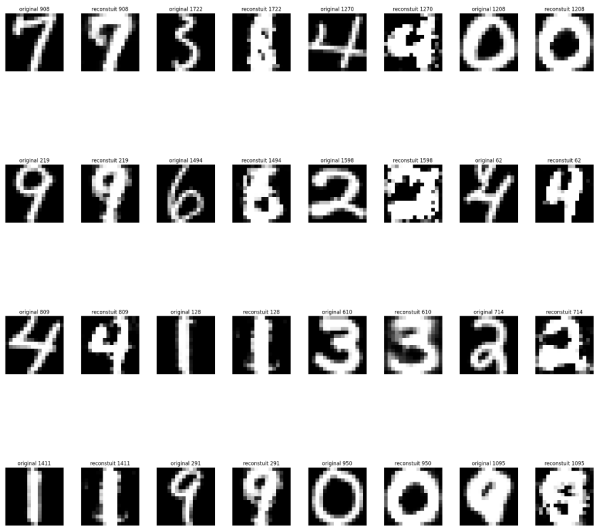

Figure 5.1 – Test de reconstruction de données - USPS Dataset

1. https ://keras.io/api/datasets/mnist/

TanH Lorsque nous avons utilisé la fonction d'activation tanH, nous avons observé différentes configurations, telles que 2 couches $(1- > 128, 2- > 16)$ et 3 couches $(1- > 256, 2- > 128, 3- > 16)$.
Nous avons constaté que l'auto-encodeur à 2 couches donnait des reconstructions plus lisses, mais avec moins d'expressivité que celui à 3 couches. De plus, la fonction d'activation tanH produisait des résultats plus lisses comparativement à la fonction ReLU.

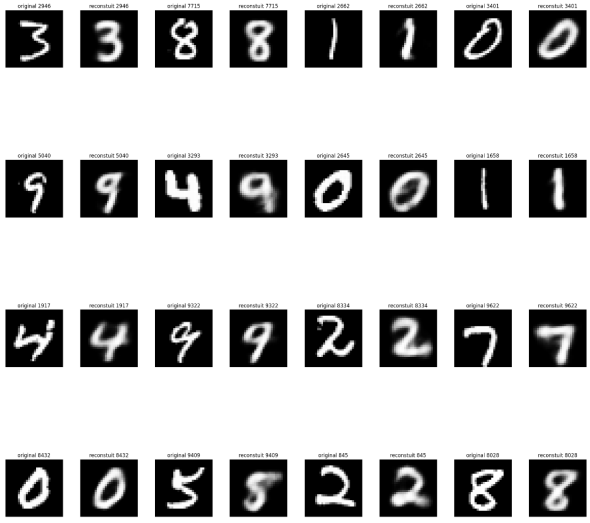

FIGURE 5.2 – Reconstruction des données de test MNIST avec la fonction d'activation TanH - encoder 2 couches 128_16

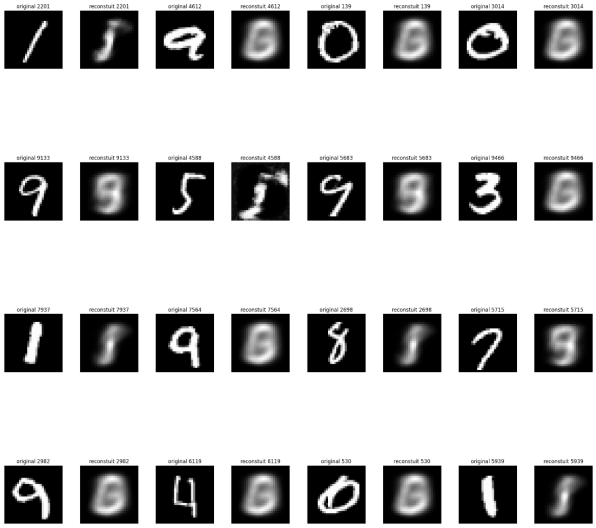

FIGURE 5.3 – Reconstruction des données de test MNIST avec la fonction d'activation TanH - encoder 2 couches 256_2

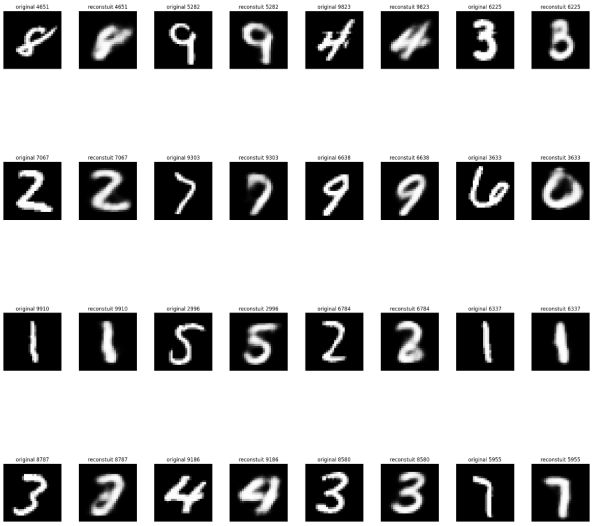

FIGURE 5.4 – Reconstruction des données de test MNIST avec la fonction d'activation TanH - encoder 3 couches 256_128_16

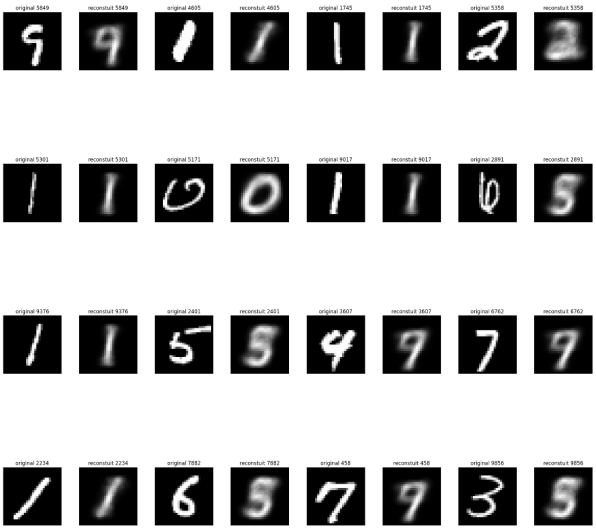

FIGURE 5.5 – Reconstruction des données de test MNIST avec la fonction d'activation TanH - encoder 3 couches 512_256_2

ReLU En ce qui concerne l'utilisation de la fonction d'activation ReLU, nous avons testé des auto-encodeurs avec 2 couches ($1- > 128, 2- > 16$) et 3 couches ($1- > 256, 2- > 128, 3- > 16$).
Les résultats ont montré que les reconstructions étaient plus brutes, avec des transitions abruptes entre les valeurs. Toutefois, l'auto-encodeur à 3 couches présentait une reconstruction moins brutale que celui à 2 couches, grâce à une plus grande expressivité du modèle.

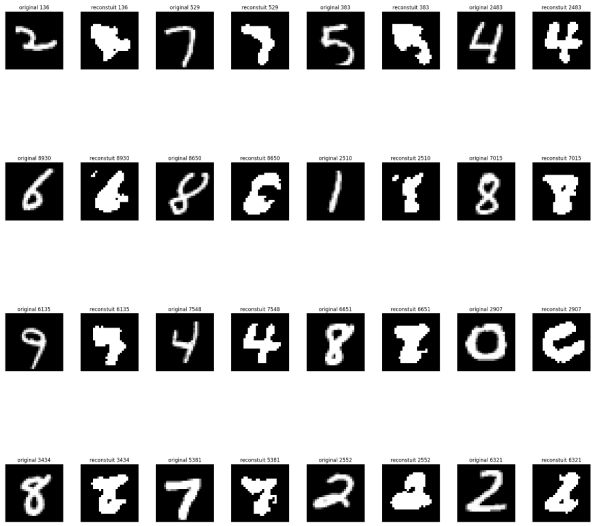

FIGURE 5.6 – Reconstruction des données de test MNIST avec la fonction d'activation ReLU- encoder 2 couches 128_16

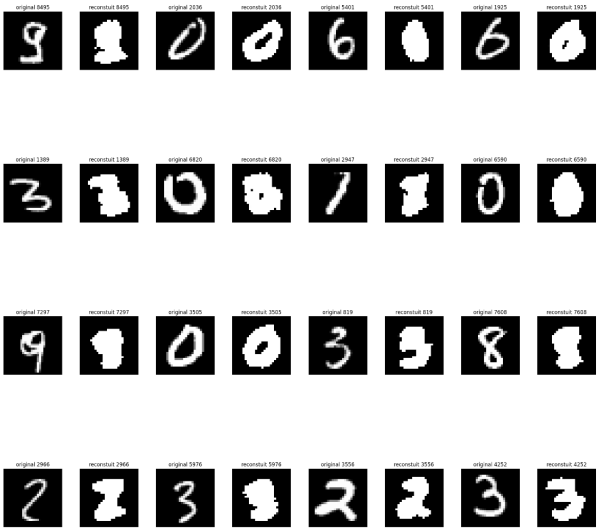

FIGURE 5.7 – Reconstruction des données de test MNIST avec la fonction d'activation ReLu - encoder 3 couches 256_128_16

Combinaison des deux Finalement, nous avons opté pour une combinaison de ReLU et tanH.
Cette approche consiste à utiliser tanH comme fonction d'activation dans l'encodeur et ReLU dans le décodeur. Cette combinaison nous a permis d'éviter une trop grande quantité de zéros dans le décodeur, ce qui aurait pu nuire à la performance.

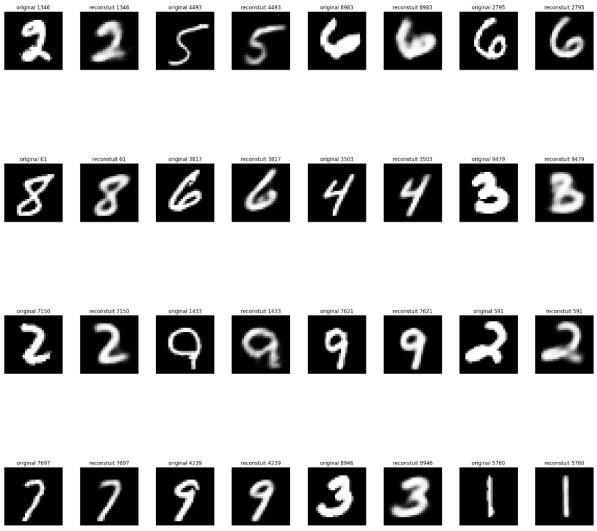

FIGURE 5.8 – Reconstruction des données de test MNIST avec la combinaison des deux - encoder 3 couches 256_128_16

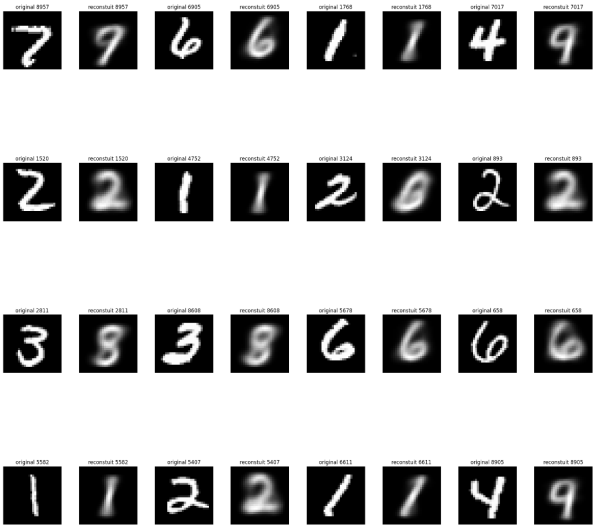

FIGURE 5.9 – Reconstruction des données de test MNIST avec la combinaison des deux - encoder 3 couches 512_256_2

Les résultats ont démontré que cette combinaison était particulièrement efficace, en particulier pour des espaces latents de dimension 2 et 16, offrant des reconstructions plus claires et détaillées par rapport à l'utilisation exclusive de ReLU ou tanH.

5.1.2 Classification avec les espaces latents

Dans la deuxième partie de notre étude, nous avons exploré la possibilité de réaliser une classification en utilisant les espaces latents créés par nos auto-encodeurs. Nous avons projeté nos caractéristiques dans des espaces de dimension 16 et 2 en utilisant l'encodeur de nos modèles.

Cependant, les résultats obtenus avec une dimension de 2 n'étaient pas satisfaisants, tandis que ceux avec une dimension de 16 étaient acceptables.

Ci-joint les résultats détaillés :

	train	test
16	0.92472	0.8326
2	0.38458	0.3796

FIGURE 5.10 – Résultats de la classification latente

5.1.3 Visualisation tSNE

Dans cette troisième partie d'expérimentation, nous avons examiné si notre auto-encodeur était capable de comprimer les informations dans les espaces latents de manière à pouvoir distinguer les différentes classes.

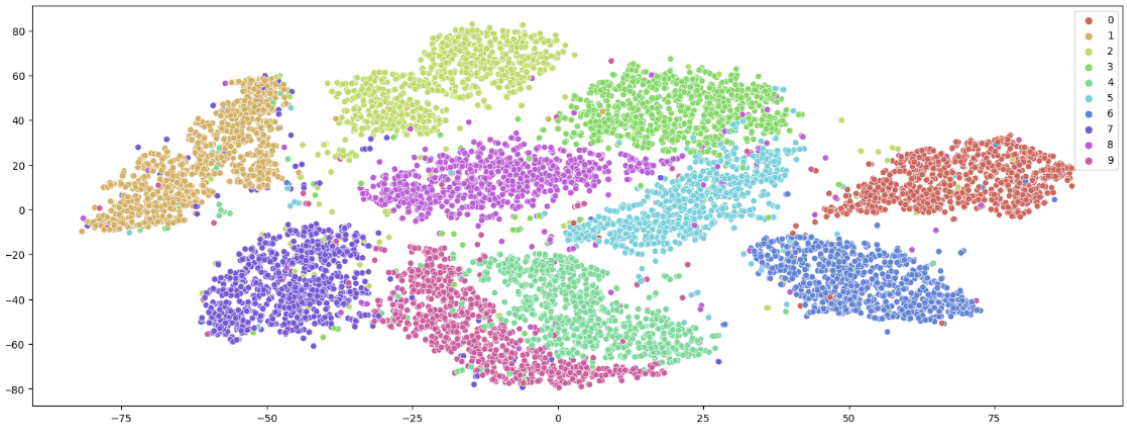

FIGURE 5.11 – Visualisation tSNE des données d'origine

En utilisant un encodeur avec une configuration de 256_128_16 et une combinaison de ReLU et tanH, nous avons constaté que nos espaces latents étaient capables de compresser les images de manière à pouvoir les distinguer.
Cela est clairement visible dans la figure ci-dessous qui représente les espaces latents de dimension 16.

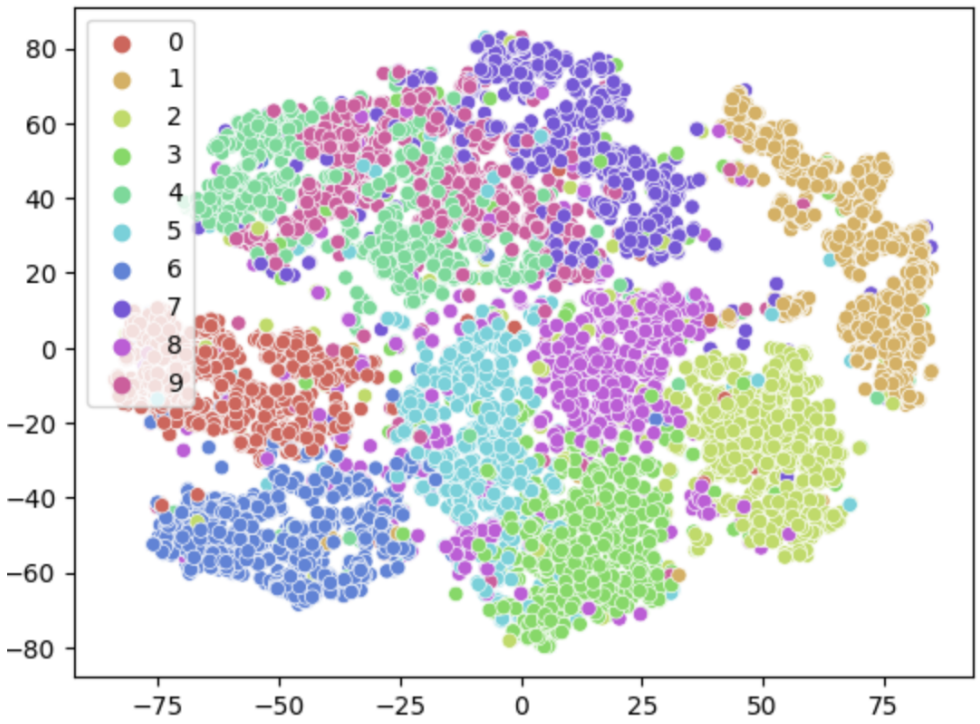

FIGURE 5.12 – Visualisation tSNE des espaces latents de dimension 16

De même, en utilisant un encodeur avec une configuration de 512_256_2 et la même combinaison de ReLU et tanH, nous avons observé que nos espaces latents étaient également capables de compresser les images de manière à pouvoir les distinguer.

D'après les résultats de cette classification, nous avons obtenu un score de 30%. Cela signifie que notre encodeur est capable d'encoder correctement certaines images de chaque classe, mais il a du mal tout de même, à encoder les autres images de la même classe.
En conséquence, lors de la visualisation avec t-SNE, nous avons observé neuf clusters distincts.
Ceci est illustré dans la figure suivante qui représente les espaces latents de dimension 2.

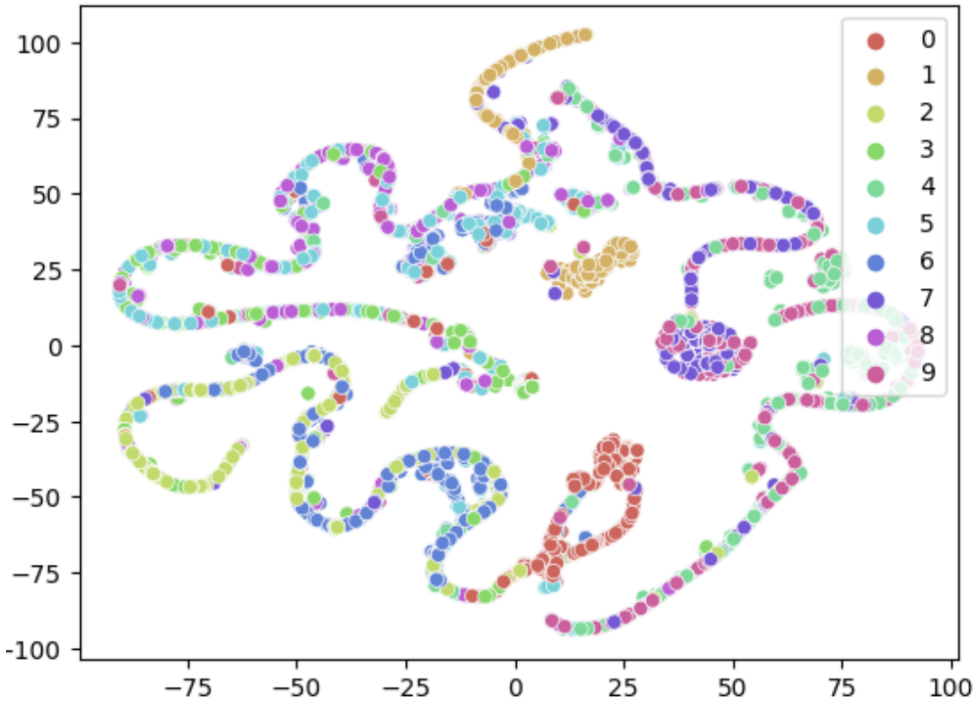

FIGURE 5.13 – Visualisation tSNE des espaces latents de dimension 2

La performance de notre modèle peut être améliorée en ajustant différents aspects de l'apprentissage, tels que l'architecture du réseau, les hyperparamètres, ou en utilisant des techniques plus avancées comme la régularisation ou le transfert learning.
Il est également possible que les données elles-mêmes présentent des difficultés de séparabilité ou de similarité entre les classes, ce qui peut affecter les performances du modèle.
Une analyse plus approfondie de ces aspects pourrait nous aider à améliorer les résultats de la classification avec les espaces latents.

5.1.4 Reconstruction de donnes bruitées avec un auto-encoder

Cette méthode vise à réduire le sur-apprentissage et à atténuer le risque d'évaporation du gradient.

Nous avons réalisé des expérimentations en utilisant un auto-encodeur avec une configuration de 256_128_16 et en ajoutant un bruit gaussien aux données d'entrée.
Plus précisément, nous avons testé deux niveaux de bruit gaussien : 0.8 et 1.5.
Dans les deux cas, nous avons observé que notre modèle était capable de reconstruire efficacement les images d'origine, comme le montrent les figures suivantes.

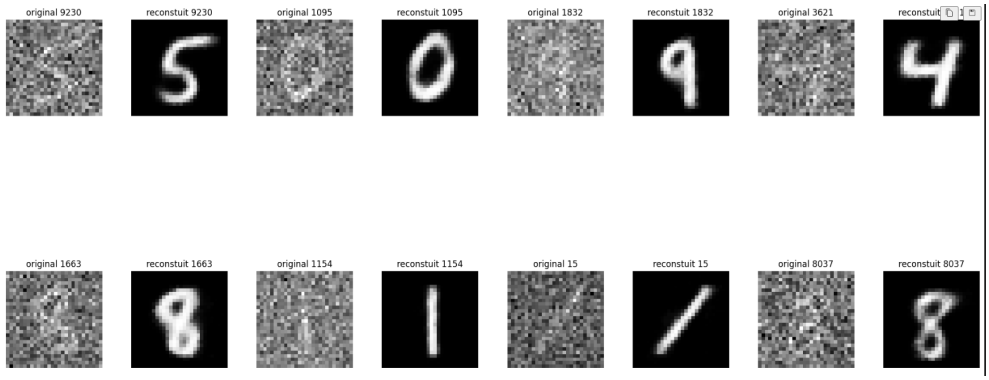

FIGURE 5.14 – Reconstruction avec bruit gaussien = 0.8

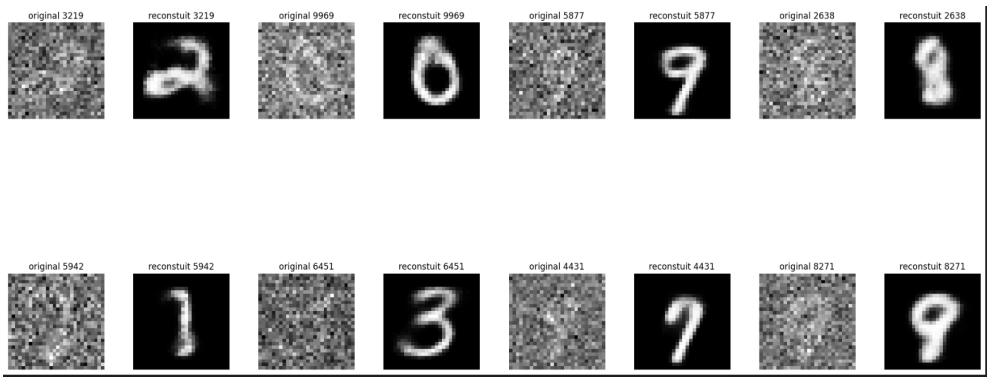

FIGURE 5.15 – Reconstruction avec bruit gaussien = 1.5

Ces résultats indiquent que l'ajout d'un bruit gaussien aux données d'entrée permet de construire des reconstructions précises, ce qui peut contribuer à réduire le sur-apprentissage et à prévenir l'évaporation du gradient.

Chapitre 6

Module convolutionnel

Un réseau neuronal convolutionnel, également connu sous le nom de CNN ou ConvNet, est un type de réseau neuronal spécialement conçu pour traiter des données disposées sous la forme d'une grille, principalement des images. Les images numériques sont représentées par des données binaires.

Un CNN est généralement composé de trois types de couches : une couche de convolution, une couche de pooling et une couche entièrement connectée. La couche de convolution effectue principalement des opérations de multiplication matricielle entre des filtres et la matrice d'entrée pour extraire des caractéristiques. La couche de pooling réduit la dimension de la représentation en effectuant une opération de sous-échantillonnage. Enfin, la couche entièrement connectée est utilisée pour la classification des données.

Dans cette section, nous présentons les résultats de l'implémentation des modules Conv1D et Conv2D. Il est important de noter que ces modules, en raison des nombreuses opérations de multiplication qu'ils effectuent, peuvent nécessiter un temps significatif pour le traitement, notamment en l'absence d'un GPU. L'utilisation d'un GPU peut accélérer les calculs et rendre l'exécution plus rapide.

Conv1D Dans cette section, nous présentons les résultats de l'application du module Conv1D sur les données USPS.
Nous avons utilisé le réseau suivant :
Sequential(Conv1D(3, 1, 32), MaxPool1D(2, 2), Flatten(), Linear(4064, 100), ReLU(), Linear(100, 10), SoftMax())

Les performances obtenues avec ce modèle sont excellentes, avec un score de 0.98. De plus, la Figure 6.1 illustre l'évolution de la moyenne et de la variance au fur et à mesure des itérations.

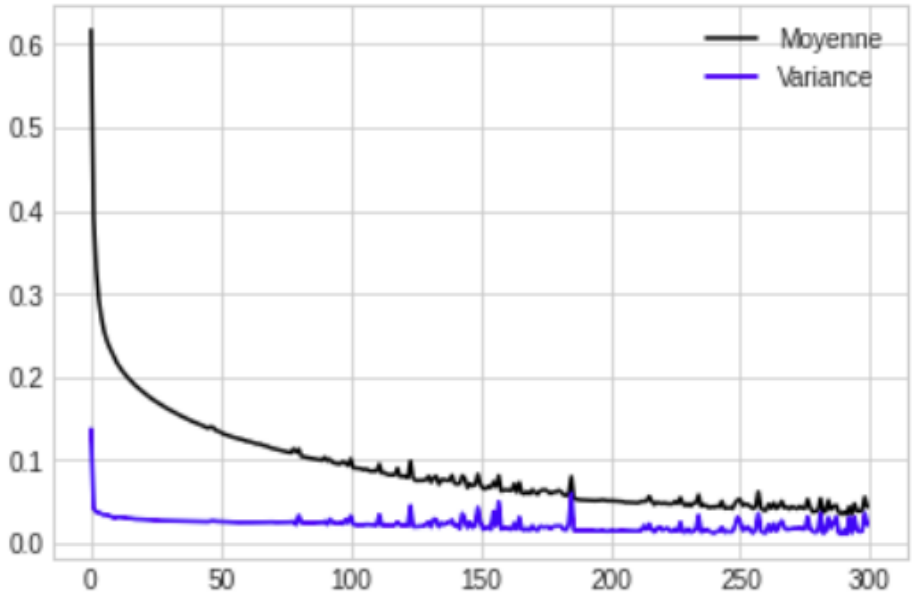

FIGURE 6.1 – Conv1D : Évolution de la moyenne et de la variance durant le train

Nous observons une diminution de ces valeurs, ce qui indique une convergence du modèle.

Conv2 Nous présentons dans cette section les résultats du module Conv2D sur les données USPS.
Nous reprenons le réseau suivant :
Sequential(Conv2D(3, 1, 32), MaxP ool2D(2, 2), Flatten(), Linear(1568, 100), ReLU(),Linear(100, 10), SoftMax())

Ce modèle a également obtenu d'excellentes performances, avec un score de 0.96. De plus, la Figure 6.2 illustre l'évolution de la moyenne et de la variance au fur et à mesure des itérations.

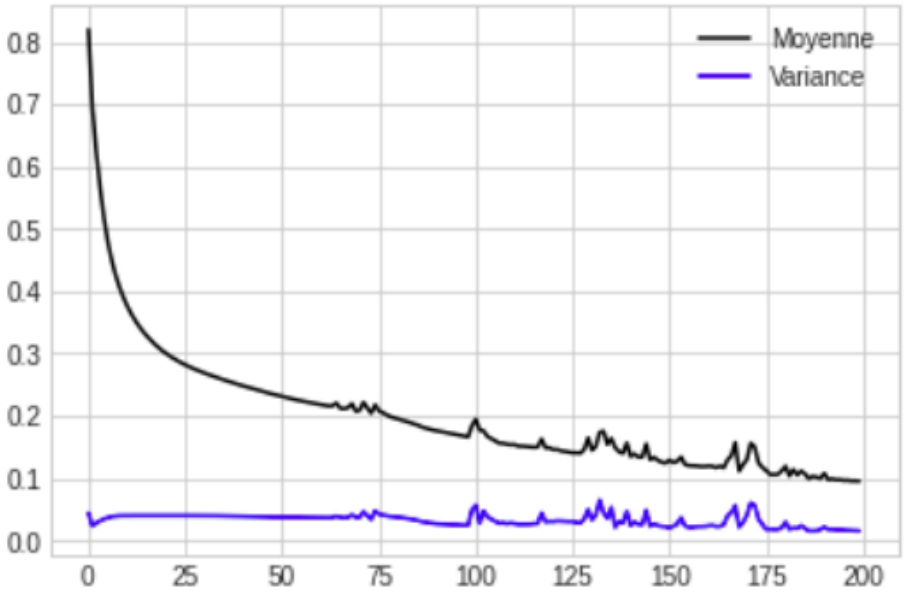

FIGURE 6.2 – Conv2D : Évolution de la moyenne et de la variance durant le train

Nous constatons une diminution de ces valeurs, ce qui indique une convergence du modèle.

Conclusion

Dans ce projet, nous avons mis en œuvre et expérimenté différents modules d'un réseau neuronal, en nous inspirant des anciennes implémentations de la bibliothèque PyTorch. Nous avons appliqué ce réseau à différentes tâches d'apprentissage, notamment la classification de données linéairement séparables avec des fonctions d'activation linéaires, ainsi que des données non linéairement séparables avec des fonctions d'activation telles que la sigmoïde et TanH.

Nous avons également abordé la classification binaire et multi-classe, et avons effectué des expérimentations avec un auto-encodeur, notamment pour le débruitage des données, la reconstruction et la visualisation avec t-SNE.

Enfin, nous avons examiné les résultats de nos implémentations des modules Conv1D, MaxPool1D, Flatten, MaxPool2D, ainsi que Conv2D. Ces modules sont utilisés dans le traitement des données de type grille, comme les images, et nous avons évalué leurs performances dans notre projet.