

SORBONNE UNIVERSITÉ
SCIENCES ET INGÉNIERIE
Master Informatique

Réseau de neurones DIY

Antoine Théologien - 21400184



Parcours : DAC
Cours : Machine Learning
Date : April 17, 2025

1 Introduction

Ce rapport est consacré au projet du cours Machine-Learning et consiste à réaliser une implémentation de réseau de neurones from scratch en Python, en essayant de reproduire au mieux le comportement de la très célèbre bibliothèque PyTorch. Dans celui-ci, les différents détails d'implémentation et les différentes expérimentations seront apportés. Le plan suivra celui de l'énoncé du projet, disponible dans l'archive. Le code source est également disponible dans cette dernière, ainsi que les notebooks contenant les différents tests réalisés.

2 Prémices

Le projet se base ainsi sur une architecture modulaire, permettant d'assembler les différents éléments d'un réseau, facilitant la mise en place et le déploiement. On dispose ainsi de 2 classes principales qui serviront de classes mères pour la majorité des futures classes implémentées.

2.1 Classe Module

La première est la classe **Module**, qui, comme son nom l'indique, représente un module générique de notre réseau de neurones. Celle-ci contient toutes les méthodes permettant la bonne implémentation d'un réseau de neurones : forward, réinitialisation du gradient, backward, mise à jour des paramètres. Le diagramme de la classe est visible dans la figure 1.

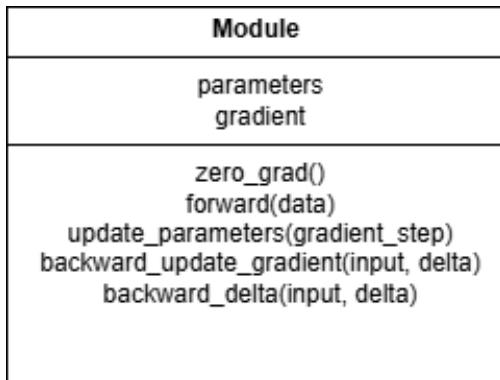


Figure 1: Diagramme de la classe Module

La majorité des fonctions de cette classe ne sont pour l'instant pas implémentées, à l'exception des méthodes *zero_grad* et *update_parameters*. Ainsi, dans la première, on commence par vérifier si la variable *gradient* n'est pas nulle, afin de s'assurer de pouvoir la manipuler. Si elle existe bien, on met à jour cette variable en mettant un zéro pour chaque paramètre contenu dans notre variable *parameters*. Dans *update_parameters*, on commence par vérifier cette fois-ci si les deux variables de notre classe sont bien définies, puis on met à jour la variable *parameters* en soustrayant le vecteur de la variable *gradient* à celui de nos paramètres, en veillant bien à multiplier celui-ci par le pas de gradient en paramètres. Le reste des méthodes seront implémentées dans les classes filles qui héritent de cette classe abstraite.

2.2 Classe Loss

Cette seconde classe abstraite nous permet donc de disposer des méthodes pour implémenter les différentes fonction de perte que nous souhaiterions utiliser dans notre réseau de neurones. Elle ne dispose que de deux méthodes : *forward*, qui prend en paramètre deux entrées et calcule le coût, et la méthode *backward*, qui va nous permettre de calculer le coup du gradient par rapport à une entrée. Il n'y a donc pas besoin de les implémenter pour l'instant : il faudra le faire spécifiquement en fonction des fonctions de coûts que nous souhaitons mettre en place.

3 Mon premier est ... linéaire !

3.1 Implémentation de la perte MSE

On commence donc par implémenter la fonction de perte Mean Squared Error (MSE), qui est la fonction la plus couramment utilisée pour les régressions linéaires.

$$\text{MSE}(y, \hat{y}) = (y_i - \hat{y}_i)^2$$

où y est le vecteur des valeurs réelles, \hat{y} est le vecteur des prédictions, et n est le nombre d'échantillons. Pour ceci, on commence par déclarer la méthode **forward**, qui va nous permettre de calculer la perte entre les deux vecteurs d'entrée. On commence par vérifier que les deux vecteurs sont bien de la même taille, puis on calcule la somme des carrés des différences entre les deux vecteurs, on pourra ainsi calculer la moyenne dans notre descente de gradient. On récupère ainsi notre perte. On implémente ensuite la méthode **backward**, qui correspond au calcul de gradient de notre perte, s'exprimant ainsi :

$$\frac{\partial \text{MSE}}{\partial \hat{y}} = -2(y - \hat{y})$$

Cela va nous permettre d'obtenir les valeurs des gradients de notre perte et donc de pouvoir effectuer notre backpropagation.

La classe **MSELoss** hérite de la classe **Loss** et implémente les méthodes **forward** et **backward** pour calculer respectivement la perte et le gradient.

3.2 Implémentation du module linéaire

Le module linéaire, est une brique fondamentale de notre implémentation de réseau de neurones. Celle-ci hérite de la classe **Module**, et possède comme paramètre une matrice de poids de dimension entrée/sortie. On ajoute une dimension de plus en entrée qui correspond à un vecteur de biais, facilitant l'implémentation de celui-ci. Pour la variable *gradient*, on crée une matrice remplie de zéro de la même taille.

Pour la méthode **forward**, on commence donc par ajouter un vecteur unitaire de biais dans nos données X , grâce à la fonction **hstack**. On effectue ensuite le produit matriciel grâce à l'opérateur **@**, qui est équivalent à la fonction **np.dot**.

On passe ensuite à la méthode **backward-update-gradient**, dans laquelle on commence par vérifier les bonnes dimensions de nos paramètres, mais aussi par ajouter un vecteur unitaire pour le biais. On ajoute ensuite notre gradient, obtenu par produit matriciel, dans la variable *self.gradient*.

3.3 Expérimentation : Régression linéaire

Avec ces deux éléments, nous pouvons maintenant implémenter une régression linéaire simple. Nous entraînerons un modèle pour prédire une sortie à partir d'une entrée donnée, en minimisant la perte MSE. Les premières expérimentations s'effectuent ainsi sur une simple fonction linéaire de la forme : $y = 3x+4$. Comme pour toutes les expérimentations dans ce projet, nous prenons soin de séparer les données d'entraînement et de test, afin de pouvoir évaluer la performance de notre modèle sur des données non vues, et donc d'éviter un phénomène de surapprentissage. Nous utilisons la fonction `train_test_split` de la bibliothèque `sklearn` pour cela.

On dresse ainsi un graphique montrant la valeur de la perte en fonction du nombre d'itérations, ainsi qu'un second graphique montrant la valeur de la sortie prédite par rapport à la sortie réelle. On peut voir que la perte finit bien par converger, et que les valeurs prédites sont très proches des valeurs réelles. 2.

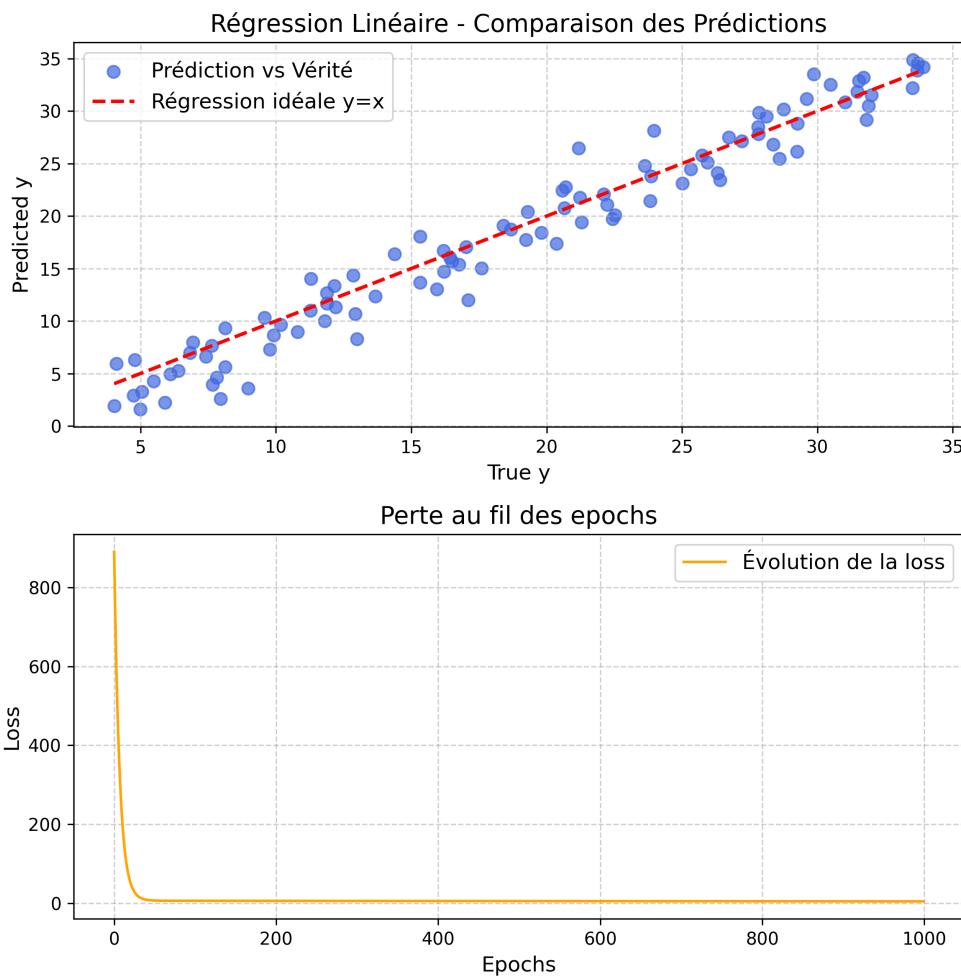


Figure 2: Loss et prédictions de la régression linéaire

On teste ensuite sur différents problèmes de classification, en utilisant la fonction `gen_arti` que nous avons déjà utilisé dans les différents TMEs de l'UE. On a donc trois jeux de données : le premier étant un mélange de deux gaussiennes, le deuxième un mélange de 4 gaussiennes (XOR), et enfin un échiquier. Les résultats de la classification sont visibles dans la figure 3.

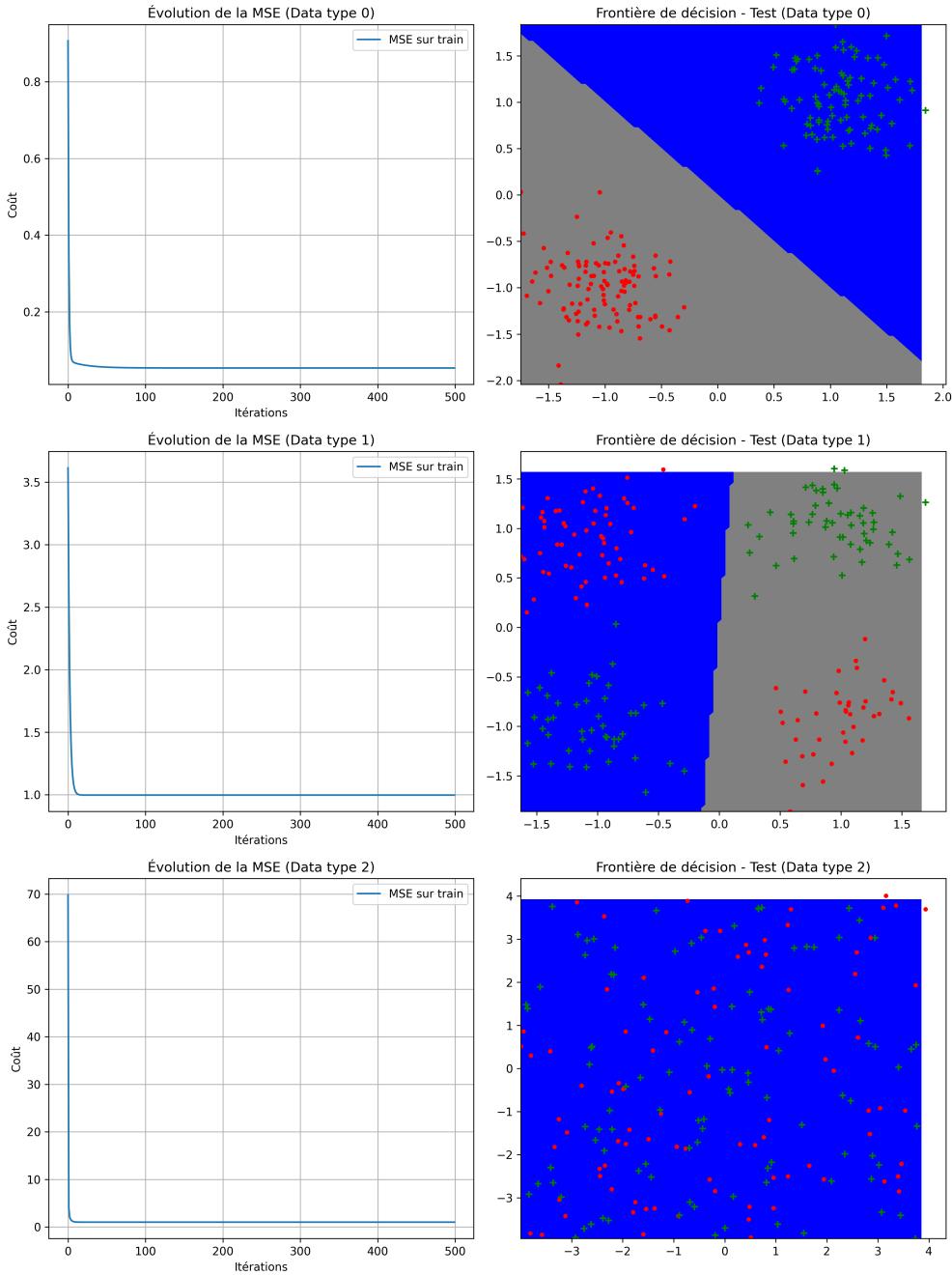


Figure 3: Loss et prédictions de la régression linéaire

On peut donc voir que la régression linéaire fonctionne bien sur la séparation des deux gaussiennes, ce qui est logique puisque la séparation est linéaire. Pour les 4 gaussiennes, la loss converge vers des valeurs beaucoup plus haute, une régression linéaire simple ne pouvant pas suffire à résoudre un tel problème, ce qui est encore plus visible sur notre problème d'échiquier.

4 Mon second est ... non linéaire !

4.1 Implémentations

On commence donc par implémenter la fonction d'activation **TanH**, qui hérite de la classe module. Il ne suffit que d'implémenter la méthode **forward**, qui consiste simplement en un appel de la fonction *tanh* de la bibliothèque **numpy**. On implémente ensuite la méthode **backward_delta**, qui va nous permettre de calculer le gradient de notre fonction d'activation. Pour cela, on utilise la dérivée de notre tangente hyperbolique, ce qui nous donne, en python :

$$\text{delta} = \text{delta} * (1 - np.tanh(X)^2)$$

Pour la fonction sigmoïde, on procède de la même manière, à l'exception qu'il n'y a pas de fonction sigmoïde déjà implémentée dans numpy, on doit donc la définir par nous-même. On implémente ensuite la dérivée de celle-ci dans la méthode **backward_delta**, qui s'exprime ainsi :

$$\text{delta} = \text{delta} * \text{sig} * (1 - \text{sig}), \text{ avec : } \text{sig} = 1/(1 + np.exp(-X))$$

Enfin, afin de tester notre premier réseau de neurones, nous avons mis en place la classe **SimpleNN**, qui va nous permettre de créer un réseau constitué d'une couche linéaire, puis d'une fonction d'activation **tanh**, puis d'une seconde couche linéaire et enfin d'une fonction **sigmoïde**. On implémente la méthode **forward**, qui va nous permettre de faire passer nos données dans le réseau, en appelant les différentes méthodes **forward** de chaque module, en prenant en paramètre le retour de la fonction forward de la couche précédente. Pour la méthode **backward**, on commence par calculer le gradient de la loss, puis on va calculer tous les deltas dont nous avons besoin par rétropropagation, en appelant la méthode **backward_delta** de chaque module, en prenant en paramètre le retour de la fonction backward de la couche suivante. Enfin, on appelle la méthode **backward_gradient_update** sur chacune des couches linéaires. Pour les deux autres méthodes que sont **zero_grad** et **update_parameters**, on appelle simplement la méthode de la classe mère pour chacune des couches linéaires.

4.2 Expérimentation : Réseau de neurones simple

On teste donc sur les mêmes jeux de données que précédemment, mais cette fois-ci avec un réseau de neurones simple, constitué d'une couche linéaire, d'une fonction d'activation **tanh**, d'une seconde couche linéaire et enfin d'une fonction **sigmoïde**. Les résultats sont visibles dans la figure 4.

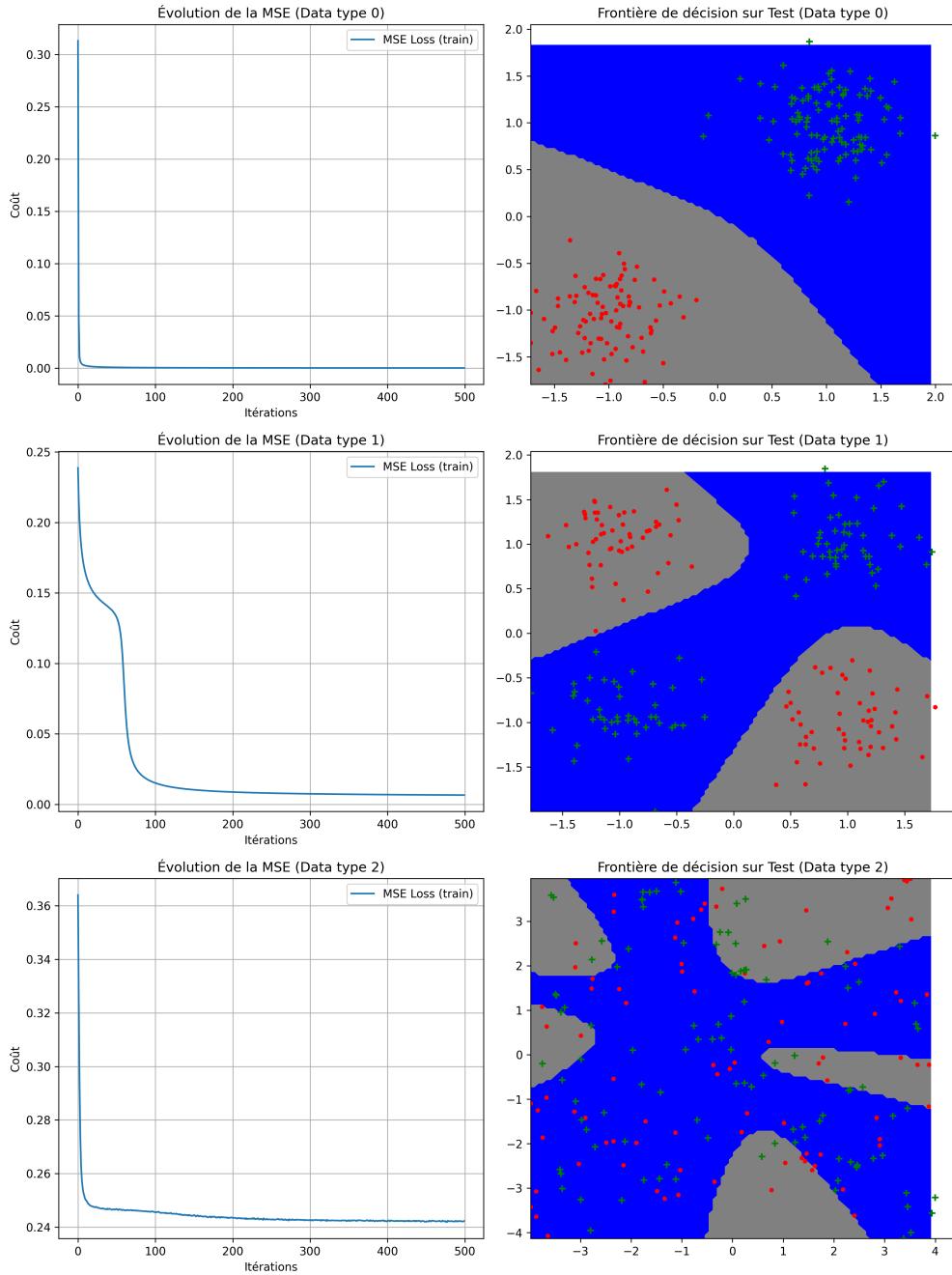


Figure 4: Loss et frontières avec une couche cachée

On peut voir que la loss converge bien plus rapidement que pour la régression linéaire, et que les résultats sont bien meilleurs. On peut voir que le réseau arrive à bien séparer les deux gaussiennes, mais aussi le XOR. En revanche, il n'arrive pas à séparer l'échiquier, ce problème nécessitant de mettre en place un réseau avec plus de couches.

5 Mon troisième est un encapsulage

5.1 Classe Sequentiel

Pour la classe **Sequentiel**, on suit le même principe que pour la classe **SimpleNN**, mais cette fois en généralisant afin de pouvoir construire un réseau de neurones plus grand. Cette classe ne possède ainsi qu'un seul paramètre, une liste de modules. Pour la méthode **forward**, on parcours donc notre liste en faisant appel à cette même méthode pour chacun des modules du réseau. Pour la **backward**, on parcours la liste dans le sens inverse, et on update les gradients de chacune des couches linéaires, en calculant également les gradients de delta. Pour les deux dernières méthodes, on appelle simplement la méthode de la classe mère pour chacune des couches linéaires.

Pour expérimenter notre réseau, on teste à la fois sur les mêmes données que précédemment, mais aussi sur d'autres répartition de données, afin de mettre en avant l'efficacité de notre implémentation. De plus, on compare l'évolution de la loss en entraînement ainsi que la frontière de décision avec la bibliothèque **PyTorch**, afin de voir si notre implémentation est correcte. Les résultats sont visibles dans les figures suivantes. On teste ici avec 500 epochs, avec une batch size de 32.

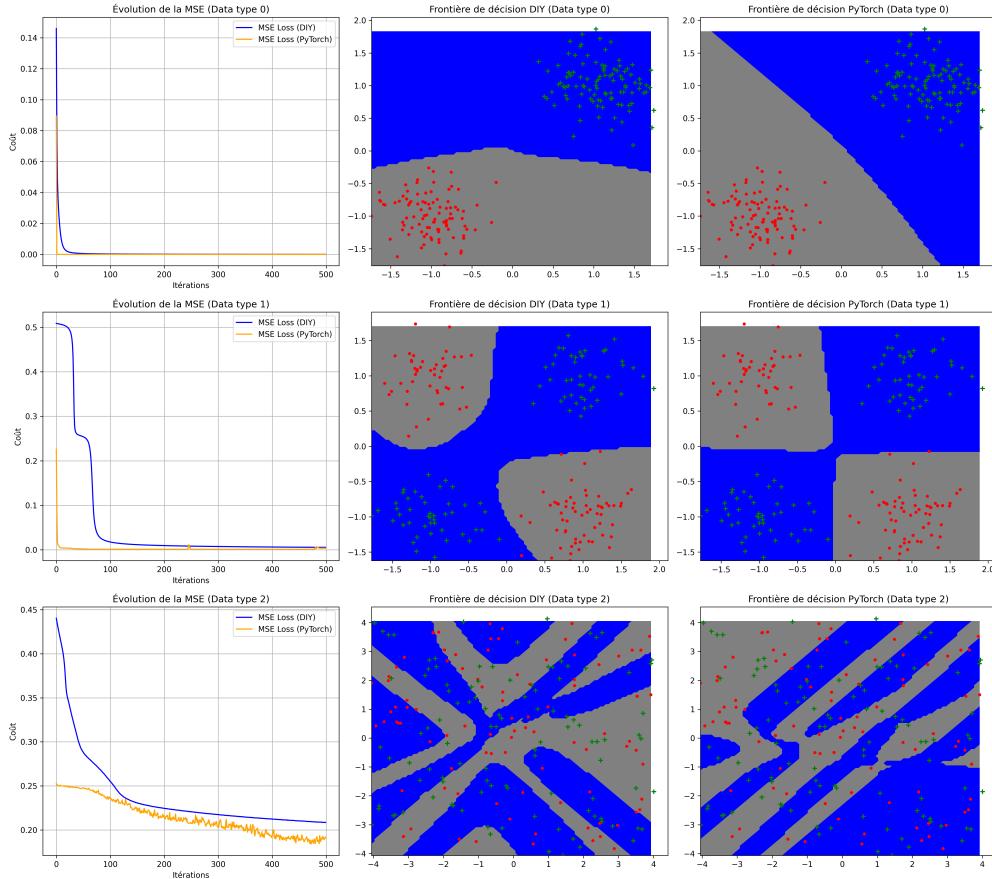


Figure 5: Loss et frontières avec la classe Sequentiel (premières données)

On teste ici avec un réseau constitué de la manière suivante : Une couche linéaire (2,5), puis une autre de (5,10), (10,20), puis (20,1), en séparant les couches internes par des fonction de tangente hyperbolique, et en ajoutant une fonction sigmoïde à la fin. On remarque que la loss converge bien plus rapidement que pour le

réseau de neurones simple, et que les résultats sont bien meilleurs. On peut voir que le réseau arrive à bien séparer les deux gaussiennes, mais aussi le XOR. En revanche, il n'arrive pas à séparer l'échiquier, ce problème nécessitant probablement de mettre en place un réseau avec plus de couches. La différence d'évolution entre la loss du réseau DIY et la loss de PyTorch est peut-être due à la non utilisation d'optimiseur pour l'instant, point que nous verrons dans la partie suivante.

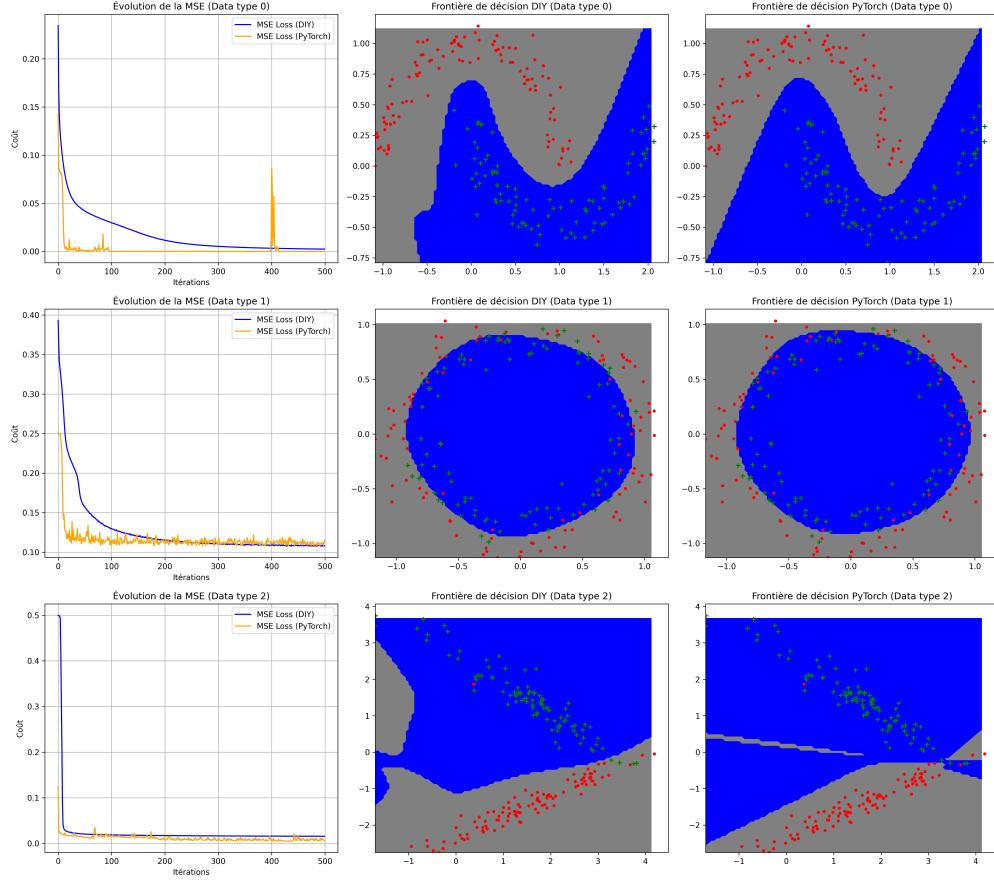


Figure 6: Loss et frontières avec la classe Sequentiel (secondes données)

Avec le même réseau, on arrive ici à résoudre des problèmes de classification plus complexes que nos deux premiers, témoignant de l'efficacité des réseaux de neurones.

5.2 Optim et SGD

Afin de nous rapprocher le plus possible de la bibliothèque **PyTorch**, on met en place la classe **Optim**, conformément aux informations de l'énoncé. On implémente également la fonction SGD, qui correspond à la fonction de train que nous utilisions précédemment en utilisant cette fois-ci notre optimiseur. On expérimente ainsi sur les mêmes données que précédemment, dont les résultats sont visibles dans la figure 7 et 8.

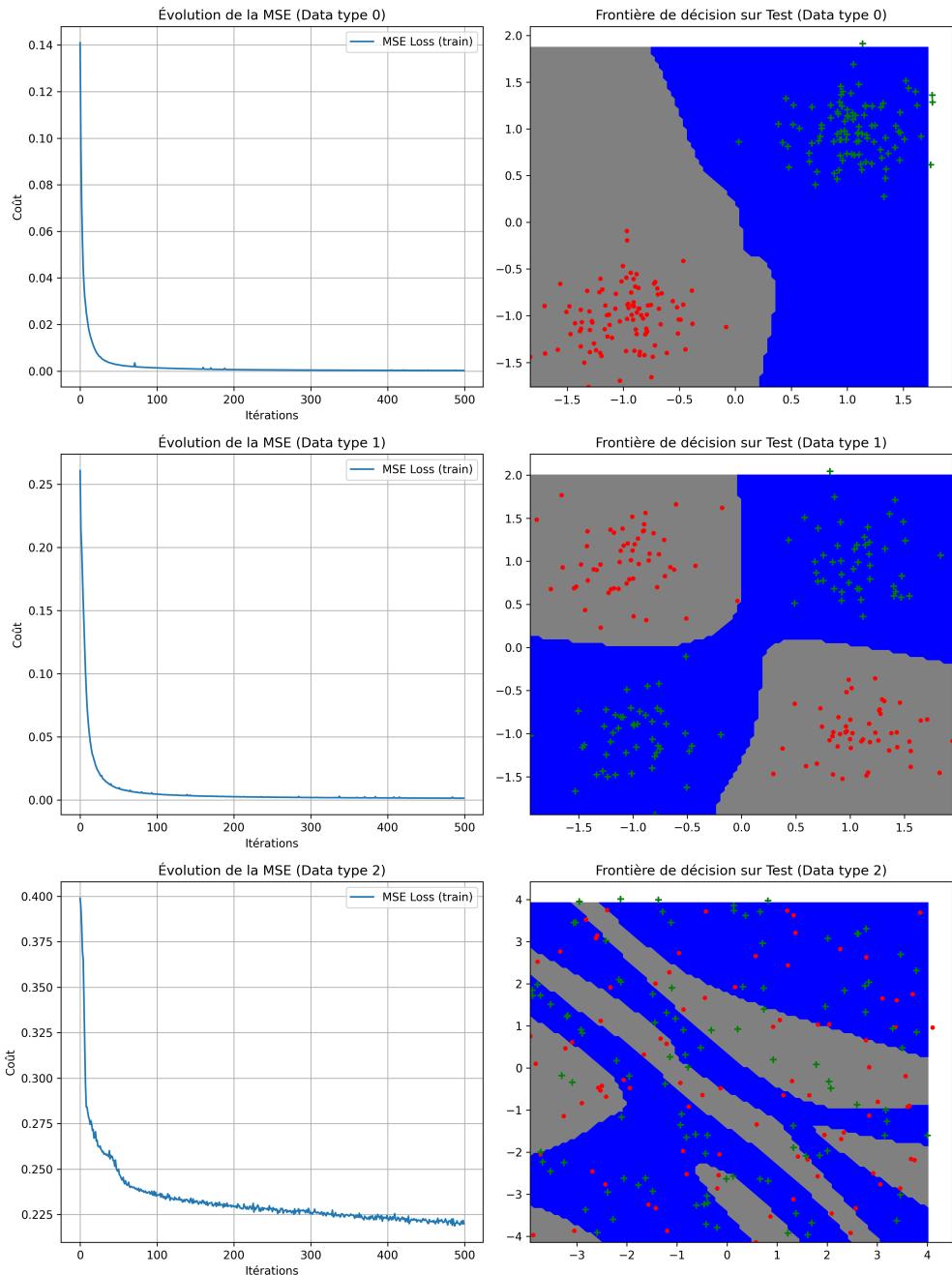


Figure 7: Loss et frontières avec optimisation (premières données)

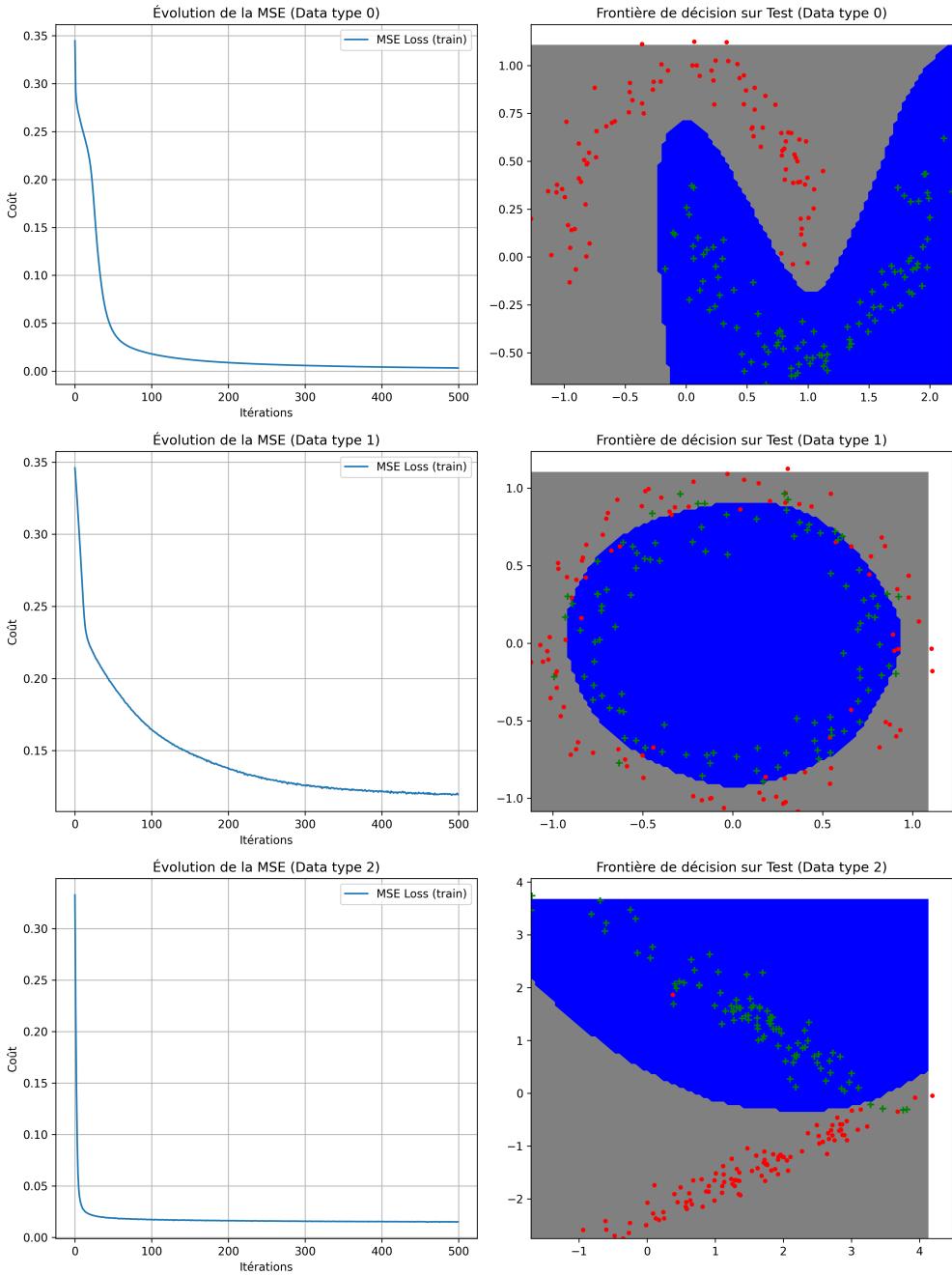


Figure 8: Loss et frontières avec optimisation (secondes données)

On remarque ainsi que nos notre loss atteint une valeur plus basse, améliorant légèrement les résultats grâce à l'utilisation de l'optimiseur.

6 Mon quatrième est multi-classe

On commence donc par implémenter notre module **SoftMax**, qui hérite donc de la classe module. On retranscrit ainsi la formule de la fonction softmax, en python, dans la méthode **forward**. De plus, afin de

pouvoir gérer le multi-classe, on ajoute la perte cross-entropique, définie comme un maximum de vraisemblance. On prend ainsi soin d'éviter les valeurs nulles, afin de ne pas avoir de problème avec notre logarithme. On expérimente ainsi sur de nouvelles données, celles des chiffres manuscrits, **USPS**. Les données sont déjà séparées en train et en test, il nous suffit donc d'entraîner notre modèle. Notre réseau de neurones est donc constitué de deux couches cachées, auxquelles on fini par appliquer un softmax. On utilise également la classe CrossEntropyLoss que nous venons de créer. Les résultats sont visibles dans la figure 9.

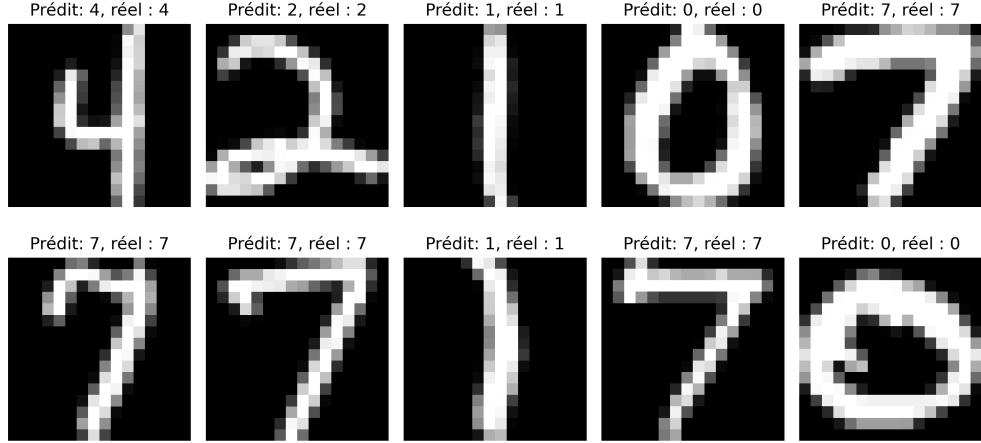


Figure 9: 10 prédictions de la classification multi-classe (200 epochs)

En testant ainsi pour 200 epochs, on obtient une accuracy d'environ 78 %, ce qui est un résultat plutôt encourageant. Les chiffres sont donc assez bien prédits, mais il reste encore environ 2 chiffres sur 10 qui sont mal classés. On peut améliorer ce résultat en utilisant la LogSoftMax et la LogCrossEntropy. Pour implémenter ces deux fonctions, on implémente simplement la Cross Entropy en intégrant le logsoftmax dans ces fonctions **forward** et **backward**. Lorsque l'on construit notre réseau, on ne met donc pas de fonction d'activation à la fin, celle-ci s'appliquant désormais directement lors du calcul de notre loss. Les résultats expérimentaux sont visibles dans la figure 10.

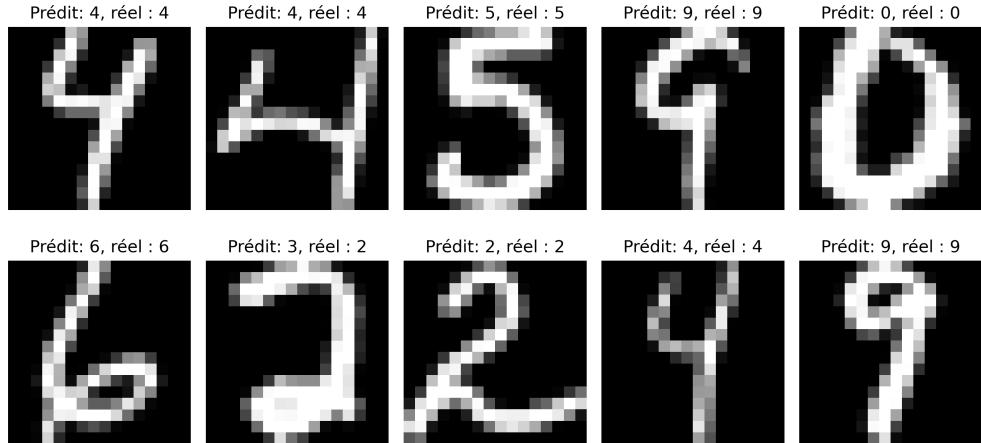


Figure 10: 10 prédictions de la classification multi-classe (200 epochs)

Nous avons cette fois-ci une accuracy de 92 %, ce qui montre l'efficacité de la LogCrossEntropy. Les chiffres sont ainsi en grande partie bien prédits, nous sommes donc désormais capables de résoudre des problèmes de classification multi-classe.

7 Mon cinquième se compresse (et s'expérimente beaucoup)

Nous allons donc mettre en place notre auto-encodeur. Pour cela, on modifie légèrement notre classe **Sequentiel**, afin de pouvoir construire un objet de cette classe à partir de deux autres objets de celle-ci. Ceci va nous permettre de créer un réseau de type encodeur, puis un autre de type décodeur. Il suffit alors de construire l'objet **Sequentiel** en mettant les deux éléments dans une liste. Notre réseau est alors construit de la manière suivante : 2 couches par partie de l'encodeur, la sortie de la première et l'entrée de la seconde étant en dimension 10, une pour chaque chiffre.

On crée ensuite notre **BCELoss**, en appliquant simplement la formule de celle-ci. Cela nous permettra d'obtenir de meilleures performances qu'avec la MSE. On entraîne ensuite notre autoencodeur sur les données **USPS**, en utilisant la fonction d'optimisation SGD. Les résultats sont visibles dans la figure 11.

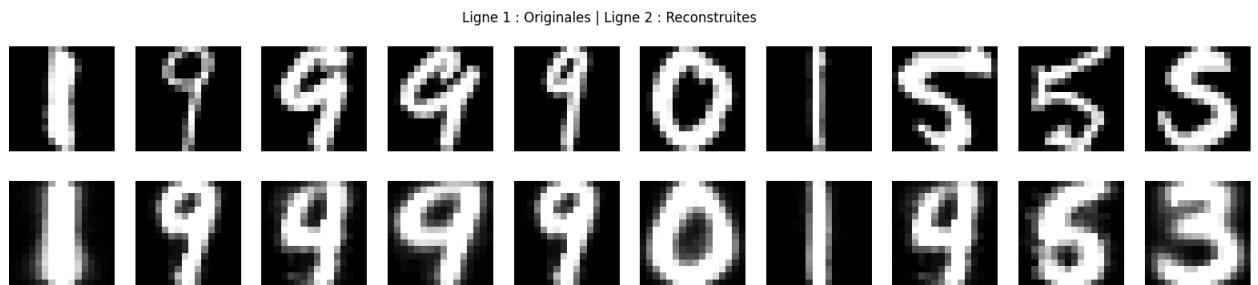


Figure 11: 10 prédictions de l'autoencodeur (1000 epochs)

Parmi les 10 chiffres tirés au hasard, on remarque que certains sont très bien recréés, tandis que d'autres, comme le 5 et le 6, sont confondus avec d'autres chiffres. Pour vérifier ce phénomène, on applique donc l'algorithme **TSNE**. Le résultat est visible dans la figure 12.

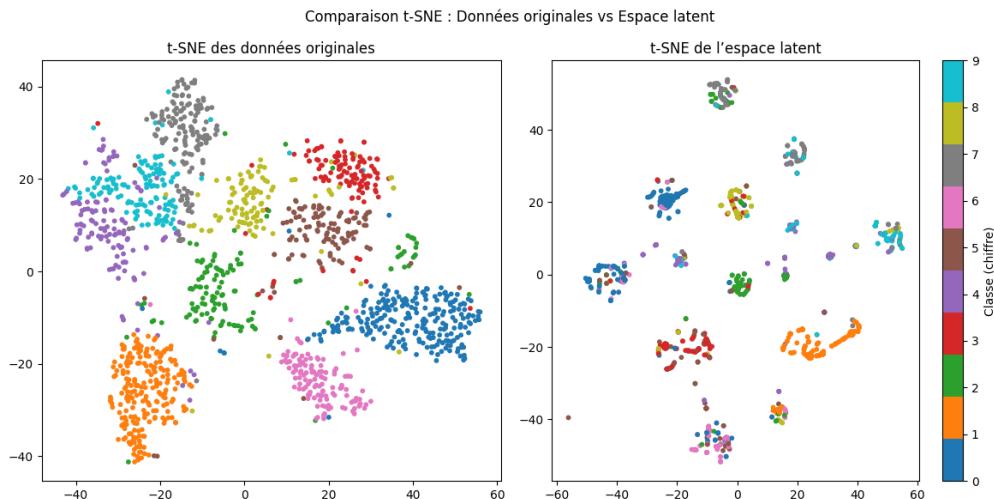


Figure 12: Résultat de l'algorithme TSNE

On ajoute à cela un test avec les k-means, afin d'évaluer le clustering de nos chiffres manuscrits. Une représentation t-sne est visible dans le figure 13. On peut voir que les clusters sont bien séparés, mais que certains se dissipent dans d'autres clusters, empêchant une reconstruction parfaite.

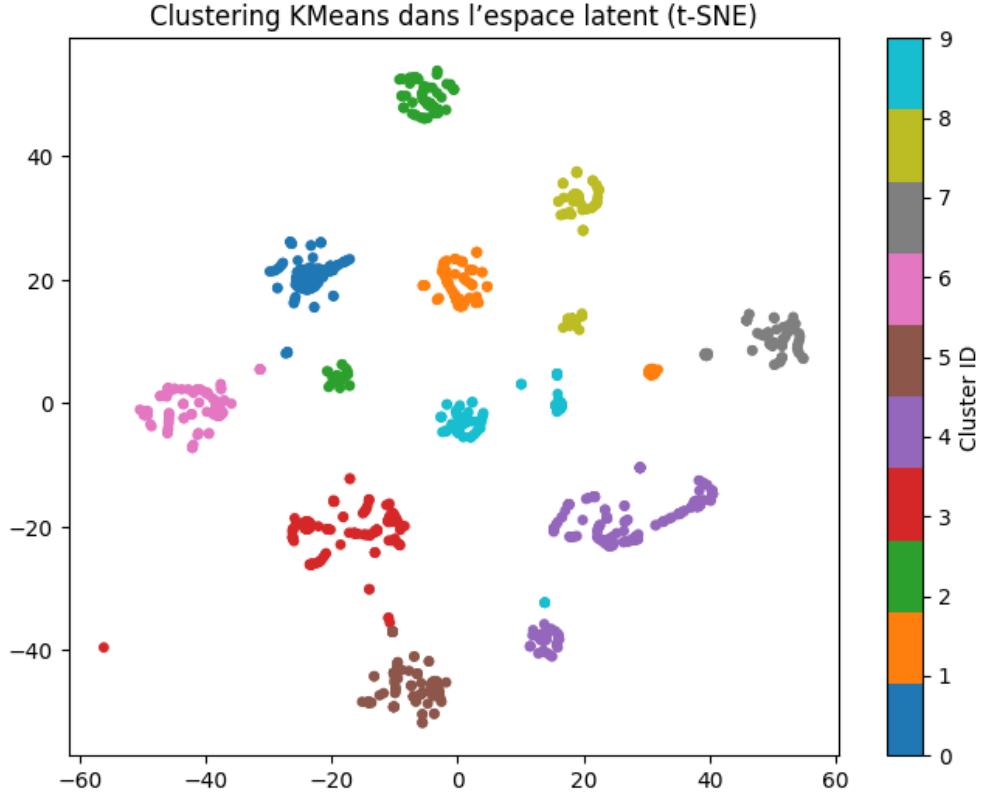


Figure 13: Résultat de l'algorithme K-means

En utilisant les métrique **Silhouette** et **ARI**, on a bien la confirmation de cette analyse : les clusters sont bien séparés mais ne sont pas alignés par rapport aux originaux. On remarque ainsi que les clusters ne sont pas les mêmes, et que les chiffres dont les formes sont proches ont tendance à se mélanger dans l'espace latent, ce qui explique nos problèmes de reproduction totalement fidèle des chiffres. Nous avons également mis en place une grid-search afin d'optimiser les valeurs des hyperparamètres pour le learning rate et la batch-size. Il s'est avéré que les meilleures valeurs étaient celles utilisées dans les expérimentations précédentes, du moins dans notre structure actuelle.

Nous avons ensuite testé les mêmes choses en mettant cette fois en place des réseaux différents, tant dans le nombre de couche et de neurones que dans les fonction d'activation. Les résultats sont visibles dans la suite du rapport.