Rapport de Projet

ML

Réseau de neurones

Étudiants :
Anyes TAFOUGHALT
Racha Nadine DJEGHALI

Encadré par :
Nicolas BASKIOTIS

24 mai 2024

Table des matières

1 Introduction :

L'objectif de ce projet est d'implémenter un réseau de neurones inspiré des anciennes versions de PyTorch (en Lua, avant l'introduction de l'autograd). L'implémentation vise à créer des réseaux de neurones modulaires et génériques, où chaque couche du réseau est représentée comme un module indépendant. Cette approche permet de concevoir des réseaux flexibles et facilement extensibles, facilitant l'expérimentation avec différentes architectures et fonctions d'activation.

Dans cette approche, un réseau est constitué de plusieurs modules, chaque module représentant une couche du réseau. Les fonctions d'activation sont également considérées comme des modules à part entière, ce qui permet une modularité maximale.Chaque module M_h prend une entrée z_{h-1} et des paramètres W_h pour produire une sortie z_h.

$$z_h = M_h(z_{h-1}, W_h)$$

La fonction de coût $L(y, \hat{y})$ évalue la différence entre la sortie prévue $\hat{y}$ et la sortie réelle y. Pour optimiser les paramètres des modules via rétropropagation, il est nécessaire de calculer le gradient de la fonction de coût par rapport aux paramètres de chaque module, $\nabla_{W_h} L$. Ce gradient est calculé en utilisant la dérivation en chaîne et la rétropropagation, en s'appuyant sur deux gradients spécifiques :

1. Le gradient du module par rapport à ses paramètres $\nabla_{W_h} M_h$, qui peut être calculé indépendamment du reste du réseau, en fonction des caractéristiques du module.
2. Le gradient de l'erreur par rapport aux sorties du module $\nabla_{z_h} L$, qui représente l'erreur à corriger lors de la rétropropagation et est fourni par les modules suivants dans le réseau par induction.

$$\nabla_{W_h} L = \left(\frac{\partial z_h}{\partial W_h}\right)^T \nabla_{z_h} L$$

Pour chaque module M_h, les poids W_h sont "aplatis" en une dimension, ce qui permet de simplifier le calcul des dérivées partielles. Les équations de la rétropropagation s'expriment alors comme suit :

$$\nabla_{W_h} L = \left(\frac{\partial z_h}{\partial W_h}\right)^T \nabla_{z_h} L$$

$$\nabla_{z_{h-1}} L = \left(\frac{\partial z_h}{\partial z_{h-1}}\right)^T \nabla_{z_h} L$$

Ces équations montrent que, pour appliquer la rétropropagation, chaque module doit être capable de calculer sa dérivée par rapport à ses paramètres et sa dérivée par rapport à ses entrées. Cela garantit que chaque module peut être optimisé indépendamment tout en contribuant à l'optimisation globale du réseau.

En résumé, ce projet vise à développer une implémentation modulaire et flexible des réseaux de neurones, en mettant l'accent sur la capacité de chaque module à calculer les gradients nécessaires pour la rétropropagation et l'optimisation des paramètres. Cette approche permet de construire des réseaux de neurones robustes et facilement adaptables à différentes tâches et architectures.

Résumé de l'approche

Cette section présente la librairie centrée sur la classe abstraite `Module`, représentant un module générique du réseau de neurones. La classe `Module` comprend :

— `_parameters` : stocke les paramètres du module.

— `forward(data)` : calcule les sorties du module.

— `_gradient` : accumule le gradient calculé.

— `zero_grad()` : réinitialise le gradient.

— `backward_update_gradient(input, delta)` : calcule et ajoute le gradient du coût par rapport aux paramètres.

— `backward_delta(input, delta)` : calcule le gradient du coût par rapport aux entrées.

— `update_parameters(gradient_step)` : met à jour les paramètres selon le gradient accumulé.

En série, chaque module appelle `forward` pour la passe avant et `backward` pour la passe arrière. Les paramètres sont mis à jour avec `update_parameters`.

La classe `Loss` contient deux méthodes :

— `forward(y, yhat)` : calcule le coût.

— `backward(y, yhat)` : calcule le gradient du coût.

Après avoir implémenté la classe `Module`, nous avons développé plusieurs modules et méthodes en suivant les étapes ci-dessous :

1. **Modèles linéaires** :
   — Nous avons implémenté les classes `MSELoss` et `Linear` pour réaliser des régressions linéaires.
2. **Modèles non linéaires** :
   — Nous sommes passés aux modèles non linéaires en utilisant des fonctions d'activation et en implémentant les modules `TanH` et `Sigmoid`.
3. **Encapsulation de modules** :
   — Nous avons encapsulé la séquence des modules dans un objet de type `Sequential` pour éviter les actions répétitives et fastidieuses.
4. **Modèles multi-classes** :
   — Nous nous sommes intéressés aux modèles multi-classes en implémentant la fonction de perte `CrossEntropy` et la transformation `Softmax`.
5. **Autoencodeur** :
   — Nous avons implémenté un autoencodeur pour compresser des données et avons réalisé plusieurs expérimentations avec ce modèle.
6. **Couches convolutionnelles** :
   — Enfin, nous avons implémenté des couches convolutionnelles, le standard en classification d'image, pour améliorer les performances de nos modèles.

2 Module linéaire :

Pour tester et manipuler un modèle linéaire, nous avons besoin de données synthétiques. Ces données peuvent être générées en utilisant une relation linéaire avec un certain niveau de bruit aléatoire. Dans cette démonstration, nous allons utiliser Python pour générer ces données.

2.1 Régression linéaire :

2.1.1 Génération des données :

- **Fixation de la graine aléatoire** : `np.random.seed(0)` fixe la graine pour le générateur de nombres aléatoires afin de rendre les résultats reproductibles.
- **Génération de X** : `X = 2 * np.random.rand(100, 1)` génère 100 points aléatoires pour la variable indépendante X, répartis uniformément dans l'intervalle $[0, 2]$.
- **Définition des paramètres du modèle** : Nous définissons les coefficients a et b du modèle linéaire $y = aX + b$.
- **Ajout de bruit** : `noise = np.random.randn(100, 1)` génère du bruit selon une distribution normale centrée réduite.
- **Calcul de y** : `y = 8 * X + 2 + noise` calcule les valeurs de la variable dépendante y en appliquant le modèle linéaire et en ajoutant le bruit.
- **Visualisation** : Les données sont ensuite visualisées à l'aide d'un graphique de dispersion pour vérifier la distribution linéaire des points.

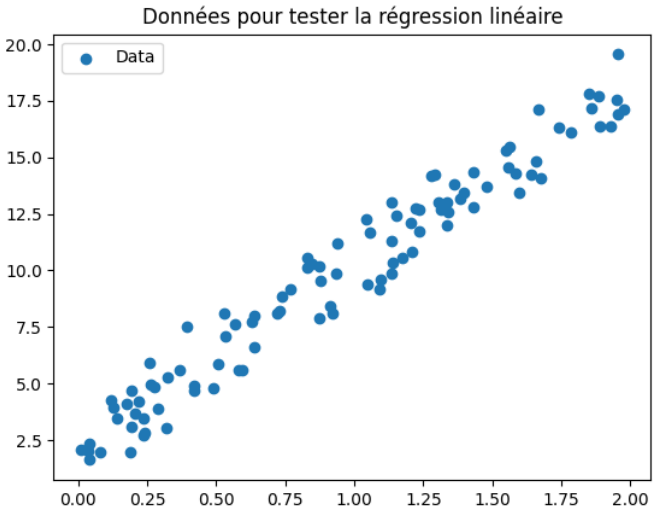


FIGURE 1 – Données générées

2.1.2 Entrainement et évaluations du modèle :

Pour entraîner notre modèle linéaire, nous avons utilisé 200 époques. À chaque époque, nous avons effectué une passe avant (forward pass) pour prédire les valeurs de sortie, calculé la perte, et effectué une passe arrière (backward pass) pour mettre à jour les

paramètres du modèle. Nous avons également évalué la perte sur l'ensemble de test pour suivre les performances au fil des époques.

En suivant ce processus sur 200 époques, nous avons pu observer et enregistrer les variations des pertes sur les ensembles d'entraînement et de test. Cela nous a permis de suivre l'amélioration du modèle au fil du temps et de sauvegarder les meilleurs paramètres obtenus durant l'entraînement.

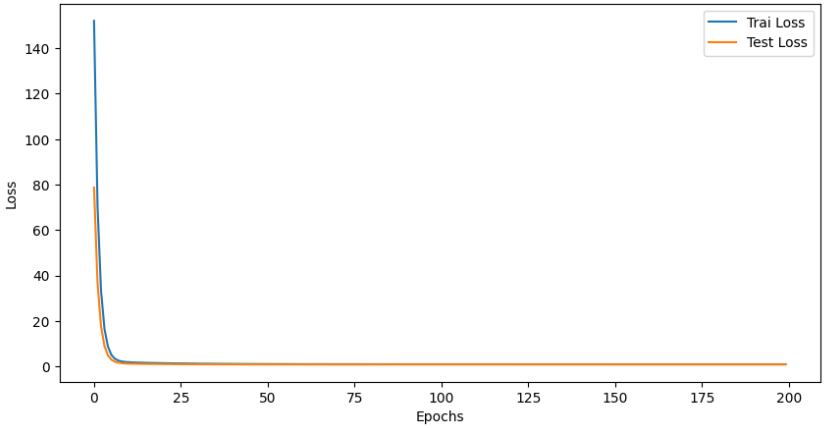


FIGURE 2 – Loss par epoch

2.1.3 Visualisation du résultat :

Maintenant que nous avons sauvegardé les meilleurs paramètres du modèle, nous pouvons afficher la régression en utilisant ces paramètres.

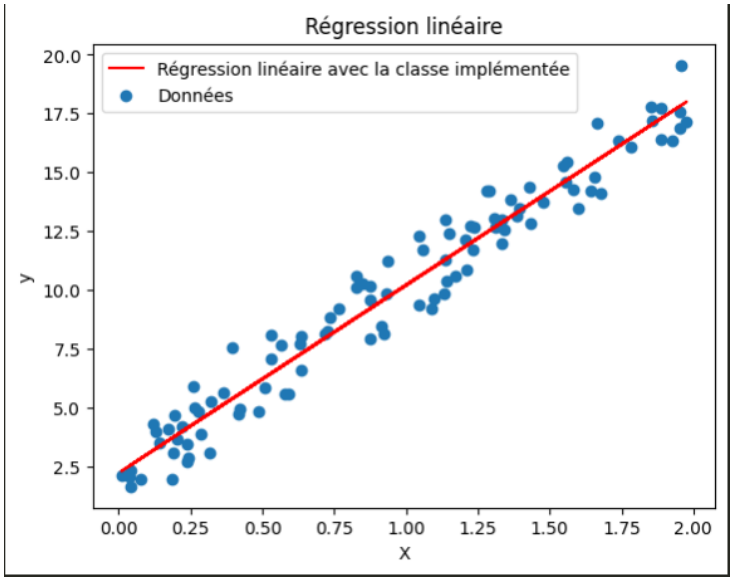


FIGURE 3 – Régression linéaire avec notre modèle linéaire

L'ajustement visuel de la droite de régression montre que le modèle linéaire est adéquat pour ces données synthétiques. Les résidus semblent aléatoires et bien répartis autour de la droite, ce qui est un bon signe que le modèle est approprié pour cette tâche de régression.

2.1.4 Comparaison avec scikit-learn :

Nous avons également comparer notre modèle personnalisé avec le modèle de régression linéaire de `scikit-learn` :

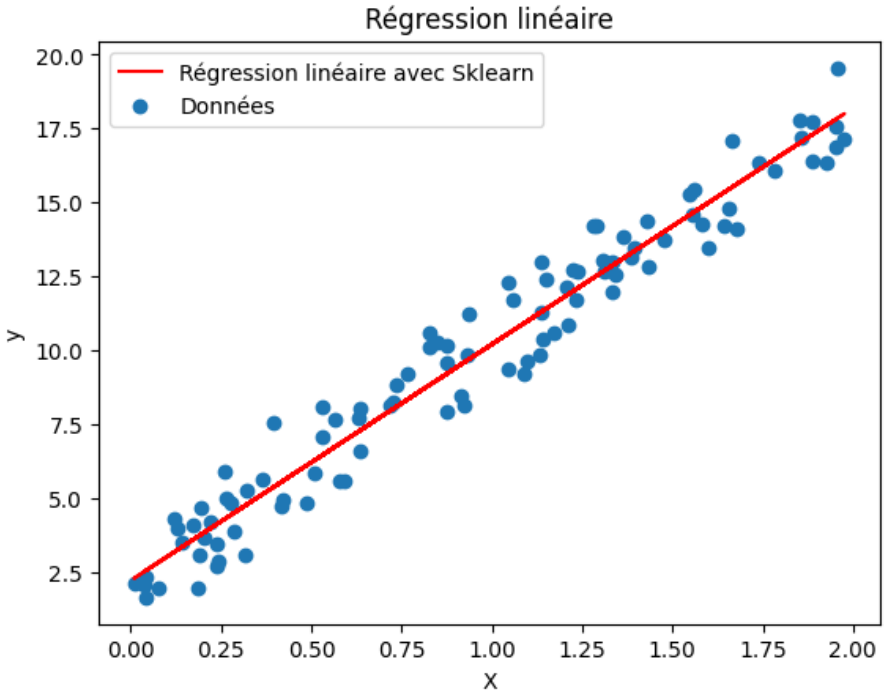


FIGURE 4 – Régression linéaire avec scikit-learn

Qualité de la régression : Les deux droites suivent de près la trajectoire des points de données, indiquant que les deux modèles capturent bien la relation entre les variables X et y.

En conclusion, les résultats montrent que notre modèle de régression linéaire personnalisé et le modèle `scikit-learn` fournissent des performances similaires. Cela confirme que notre implémentation est correcte et qu'elle capture efficacement la tendance linéaire des données.

2.2 Probème de classification :

2.2.1 Génération des données :

Pour la partie classification, nous avons généré deux catégories de points séparables linéairement. Nous avons ensuite appris un classificateur linéaire en minimisant la MSE-Loss jusqu'à obtenir un modèle qui dessine une frontière de séparation entre les deux clusters de points.

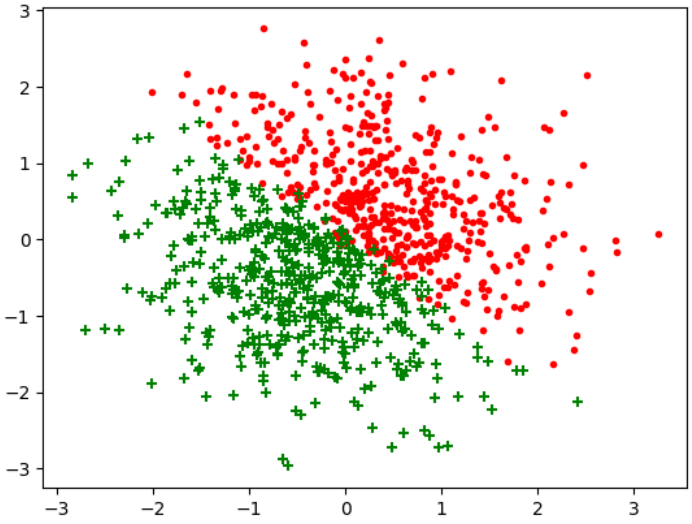


FIGURE 5 – Données séparables linéairement

2.2.2 Entrainement et évaluations du modèle :

Nous avons également suivi l'évolution de la perte (loss MSE) en train et en test au fil des époques, ce qui nous a permis de nous assurer que le modèle s'améliorait progressivement et que les performances étaient stables.

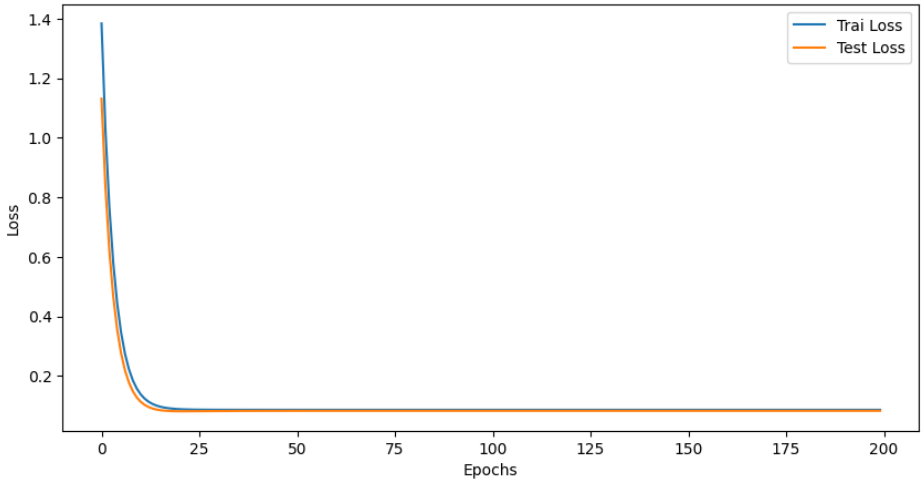


FIGURE 6 – Loss par epoch

2.2.3 Visualisation du résultat :

Maintenant que nous avons sauvegardé les meilleurs paramètres du modèle, nous pouvons afficher notre frontière de décision :

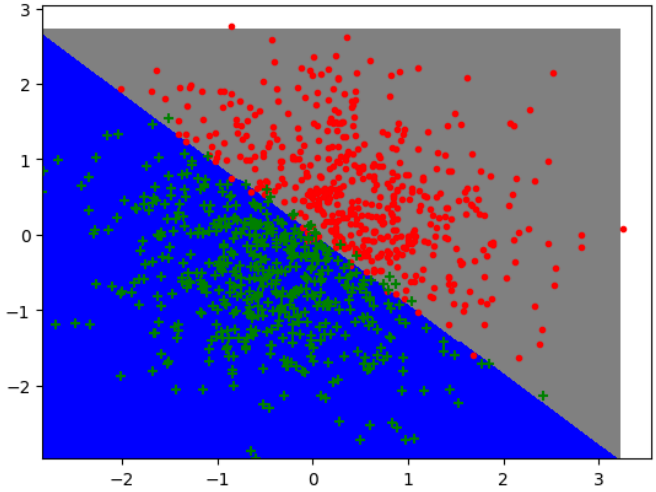


FIGURE 7 – Séparatrice par notre modèle linéaire

2.2.4 Comparaison avec scikit-learn :

Pour comparer notre classificateur linéaire avec celui de `scikit-learn`, nous avons affiché la frontière de séparation :

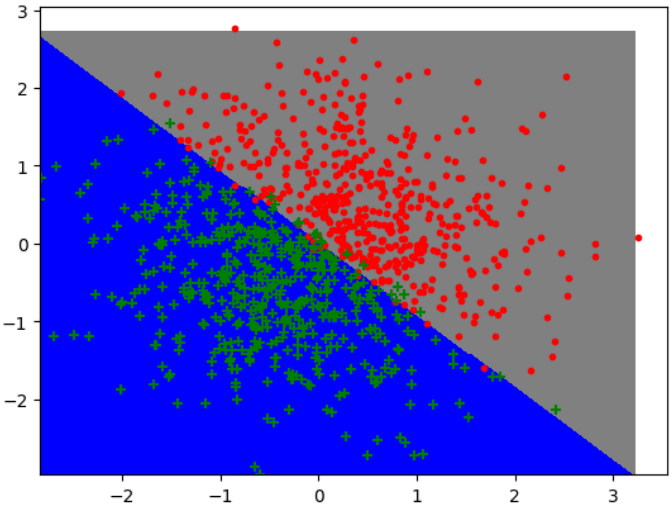


FIGURE 8 – Séparatrice par scikit_learn

En conclusion, notre classificateur linéaire personnalisé a réussi à séparer les deux catégories de points, démontrant l'efficacité de notre approche de classification. La comparaison avec `scikit-learn` valide la qualité de notre implémentation.

3 Module non linéaire :

Pour cette section, nous passons à un modèle non linéaire.

3.1 Données linéairemnt séparables :

Pour tester notre modèle nous avons commencé par vérifier s'il fonctionne sur des données linéairement séparables. Cette étape nous a permis de valider le bon fonctionnement de notre modèle de base avant de l'appliquer à des données plus complexes.

Pour ce faire nous avons adopté un modèle composé de deux couches linéaires entrecoupées d'activations non linéaires. La première couche, de dimension d'entrée **2** et de sortie **3**, est suivie d'une activation TanH. Ensuite, une seconde couche linéaire, de dimension d'entrée **3** et de sortie **1**, est suivie d'une activation Sigmoïde. Pour évaluer les performances de ce modèle, nous avons utilisé une fonction de perte Mean Squared Error (MSE) et un taux d'apprentissage de **0.01**.

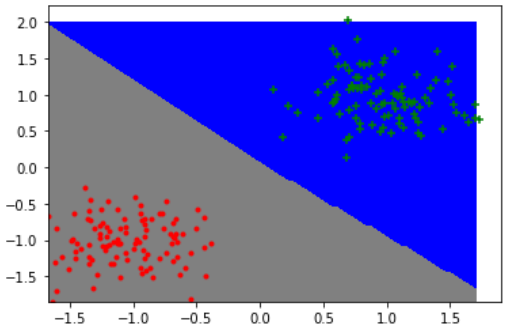


FIGURE 9 – Séparatrice

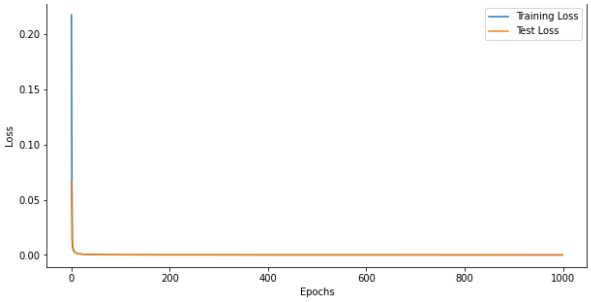


FIGURE 10 – Loss par epoch

Observation : Nous pouvons donc conclure que le modèle perfome assez bien sur ce type de données.

3.2 Données non linéairemnt séparables :

Nous avons également testé ces modules avec des données générées à partir de quatre nuages de points gaussiens, ce qui signifie que les données ne sont pas linéairement séparables.

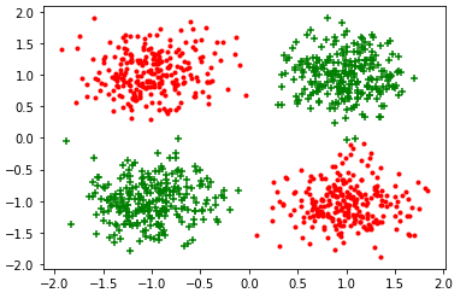


FIGURE 11 – Données non séparables linéairement

3.2.1 Entrainement et évaluations du modèle :

Afin de concevoir un modèle qui capture des relations non linéaires on a utilisé deux couches linéaires intercalées avec des activations non linéaires. La première couche, avec une entrée de dimension **2** et une sortie de dimension **50**, est suivie d'une activation Tangente Hyperbolique (TanH). Ensuite, une deuxième couche linéaire avec une entrée de dimension **50** et une sortie de dimension **1** est suivie d'une activation Sigmoïde. Nous avons utilisé la même fonction de perte Mean Squared Error (MSE) et un taux d'apprentissage de **0.0001**.

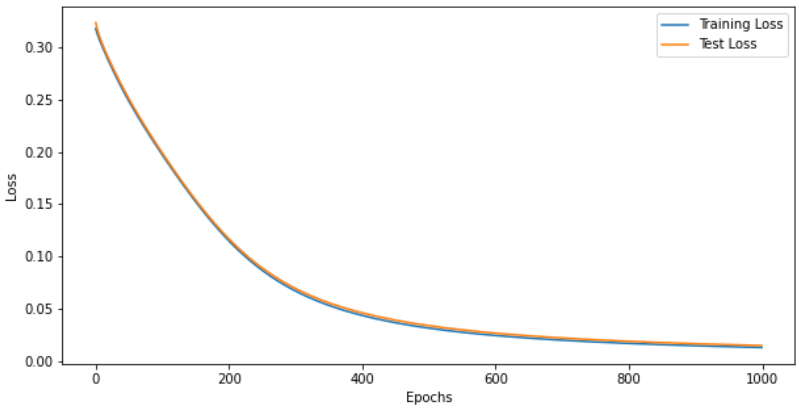


FIGURE 12 – Loss par epoch

Nous avons également suivi l'évolution de la MSELoss sur le train et test au fil des 1000 epochs, comme illustré dans la figure ci-dessus.

Nous avons remarqué que la loss (MSELoss) décroît exponentiellement jusqu'à atteindre une valeur où elle devient constante (Le modéle atteint convergence).

3.2.2 Visualisation du résultat :

Maintenant que nous avons sauvegardé les meilleurs paramètres du modèle, nous pouvons afficher nos frontières de décision obtenues :

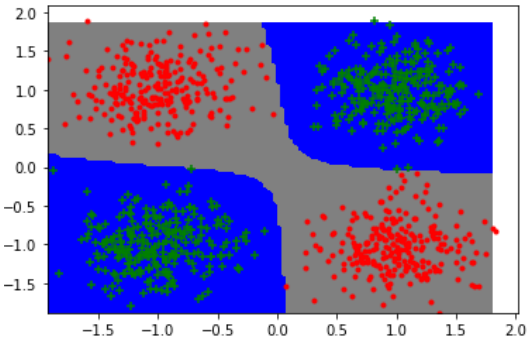


FIGURE 13 – Séparatrice par notre modèle non linéaire sur toutes les données

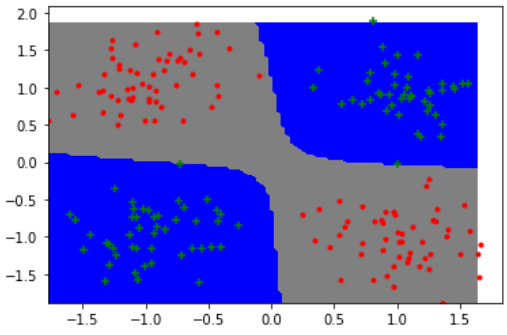


FIGURE 14 – Séparatrice par notre modèle non linéaire sur les données de test uniquement

Observation : Nous observons que le modèle sépare bien les données.

Enfin nous avons aussi étudié les variantions du score d'accuracy :

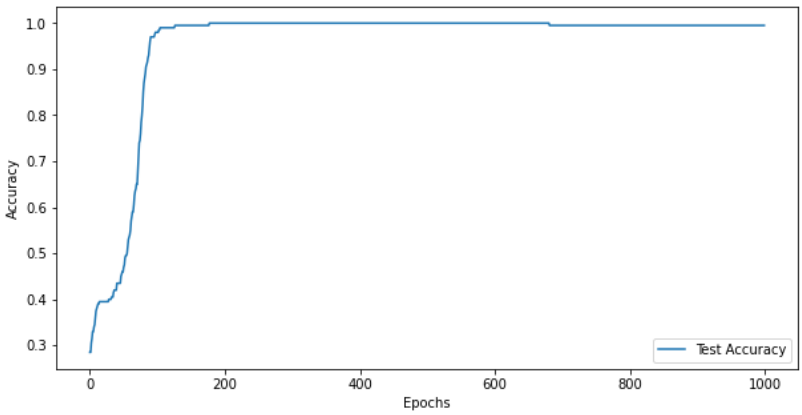


FIGURE 15 – Accuracy par epoch

Observation : Nous observons une tendance où l'accuracy augmente progressivement au fur et à mesure des epochs jusqu'à atteindre un seuil où elle devient relativement constante. De plus, au-delà de 600 itérations, nous remarquons une légère baisse sur la courbe. Cela suggère que le modèle commence à sur-apprendre les données, ce qui rend moins intéressant de poursuivre l'entraînement au-delà de ce point.

4 Module Séquentiel :

Comme les opérations de chaînage entre modules étaient répétitives lors de la descente de gradient, que ce soit pour la passe forward ou backward. Il serait fastidieux de les écrire pour un grand nombre de modules. Pour résoudre ce problème, nous avons implémenté une classe `Sequential` qui permet d'ajouter des modules en série et qui automatise les procédures de forward et backward quel que soit le nombre de modules mis à la suite.

Après avoir testé cette classe, nous avons créé une classe `Optim` pour condenser une itération de gradient. Cette classe prend dans son constructeur un réseau, une fonction de coût et un pas, et contient une méthode `step` qui calcule la sortie du réseau sur un lot de données, calcule le coût par rapport aux étiquettes, exécute la passe backward et met à jour les paramètres du réseau.

En outre, nous avons mis en place une fonction `SGD` qui, entre autres, prend en entrée un réseau, un jeu de données, une taille de lot et un nombre d'itérations. Cette fonction se charge du découpage en lots du jeu de données et de l'apprentissage du réseau pendant le nombre d'itérations spécifié.

4.1 Génération des données :

Données générées sous forme d'échequier :

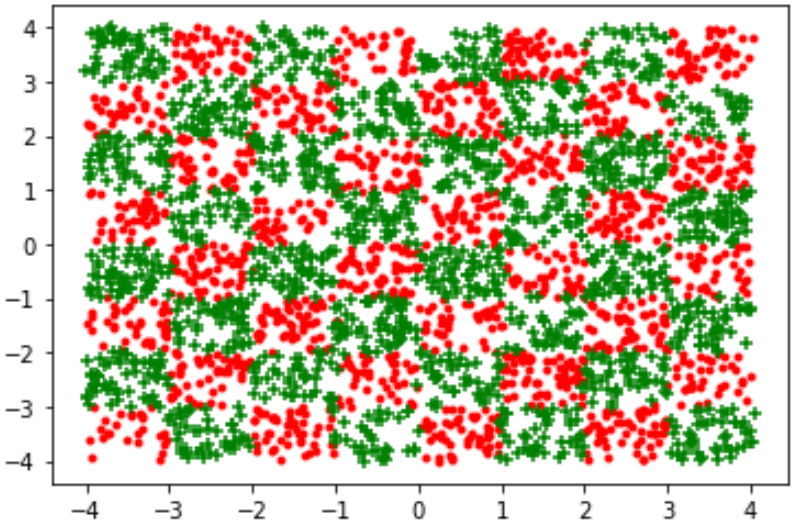


FIGURE 16 – Données sous forme d'échequier

4.2 Entrainement et évaluations du modèle :

Pour tester notre implémentation, nous avons utilisé des données d'échiquier qui ont nécessité trois couches linéaires pour être correctement séparées. Les données ont été divisées en ensembles d'entraînement et de test en utilisant la fonction train_test_split avec une taille de test de 20% et une graine aléatoire de 42. Pour notre fonction de coût, nous avons utilisé la Mean Squared Error (MSE). Les paramètres de l'optimiseur étaient définis comme suit : un lot de taille 100, 10 000 epochs et un pas de 0.001.

graLe modèle utilisé consistait en trois couches linéaires intercalées avec des activations non linéaires. La première couche avait une entrée de dimension **2** et une sortie de dimension **50**, suivie d'une activation TanH. Ensuite, une deuxième couche avait une entrée de dimension **50** et une sortie de dimension **20**, également suivie d'une activation TanH. Enfin, une troisième couche avec une entrée de dimension **20** et une sortie de dimension **1**, était suivie d'une activation Sigmoïde. Le modèle a été encapsulé dans une instance de la classe `Sequential` pour simplifier les opérations de forward et backward, puis passé à une instance de la classe `Optim` pour effectuer l'apprentissage en utilisant la descente de gradient stochastique (SGD).

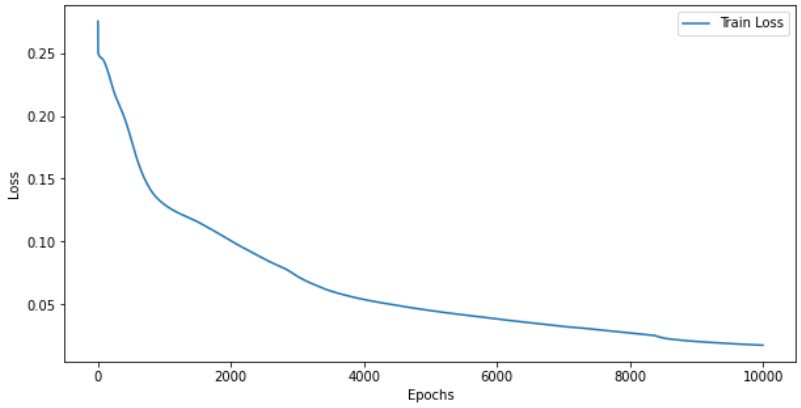


FIGURE 17 – Loss par epoch

n **Observation :**Nous avons observé une évolution favorable de la perte (loss) au fil des epochs, démontrant ainsi l'amélioration continue du modèle.

4.3 Visualisation du résultat :

Maintenant que nous avons sauvegardé les meilleurs paramètres du modèle, nous pouvons afficher nos frontières séparatrices :

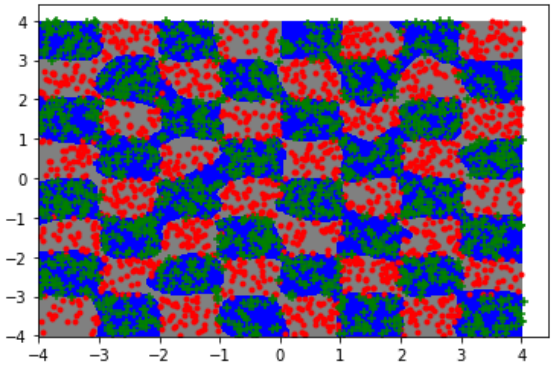


FIGURE 18 – Séparatrice par notre modèle toutes les données

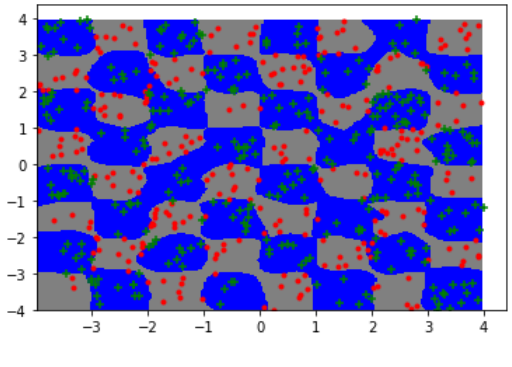


FIGURE 19 – Séparatrice par notre modèle sur les données de test uniquement

Observation : Nous avons obtenu une précision (**Accuracy**) de **0.935** sur l'ensemble de test et de **0.975** sur l'ensemble d'entraînement, ce qui indique que le modèle a bien performé.

5 Module Multi_classes

Dans cette partie du projet, nous avons implémenté un modèle de classification multi-classes en utilisant la fonction d'activation softmax et la fonction de coût cross-entropy loss. La fonction softmax convertit les scores bruts du modèle en probabilités pour chaque

classe, facilitant ainsi l'interprétation des résultats, selon la formule :

$$\text{Softmax}(z_j) = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}.$$

La fonction de coût cross-entropy loss, donnée par

$$\text{CE} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} y_{ij} \log(\hat{y}_{ij}),$$

mesure la divergence entre les probabilités prédites et les étiquettes réelles, encourageant des prédictions précises. Ensemble, ces fonctions permettent de construire un modèle robuste et précis pour les tâches de classification multi-classes.

5.1 Expérimentation

Afin de réaliser des expérimentations sur cette partie, nous avons utilisé les données USPS pour essayer de les classifier à l'aide du réseau de neurones suivant :

Network = Sequential(Linear(input, 128), Tanh(), Linear(128, 64), Tanh(), Linear(64, output), Softmax())

Dans un premier temps nous avons comparé les performances du modéles en utilisant comme fonction de cout la MSE d'une part , et la CE d'autre part et le tableau suivant résume l'accuracy des 2 modéle. d'aprés ces résultat on peut dire que la CE est mieux adapté pour ce cas de classification.

Loss	**Accuracy**
MSE	84.85%
CE	86.04%

TABLE 1 – Résultats des accuracy en fonction de la loss utilisé

Ensuite, nous avons fixé le nombre de batchs à 100 et avons varié le nombre maximum d'itérations, en calculant l'accuracy à chaque fois. Voici un tableau résumant les résultats des accuracy :

Nombre d'itérations	**Accuracy**
100	81.86%
500	89.09%
1000	90.14%

TABLE 2 – Résultats des accuracy en fonction du nombre d'itérations

Ce tableau montre comment l'accuracy de classification s'améliore en augmentant le nombre d'itérations lors de l'entraînement du modèle.

Ensuite, pour un nombre d'itérations égal à 100, nous avons varié le pas de gradient (learning rate) afin d'observer son impact sur l'accuracy. Voici les résultats résumés dans un tableau :

Pas de gradient	Accuracy
1e-1	83.15%
1e-2	86.04%
1e-3	90.14%
1e-4	65.62%
1e-6	12.25%

TABLE 3 – Résultats des accuracy en fonction du pas de gradient

Le graphique suivant représente l'évolution de la moyenne des pertes (losses) pendant la phase d'apprentissage pour chaque pas de gradient :

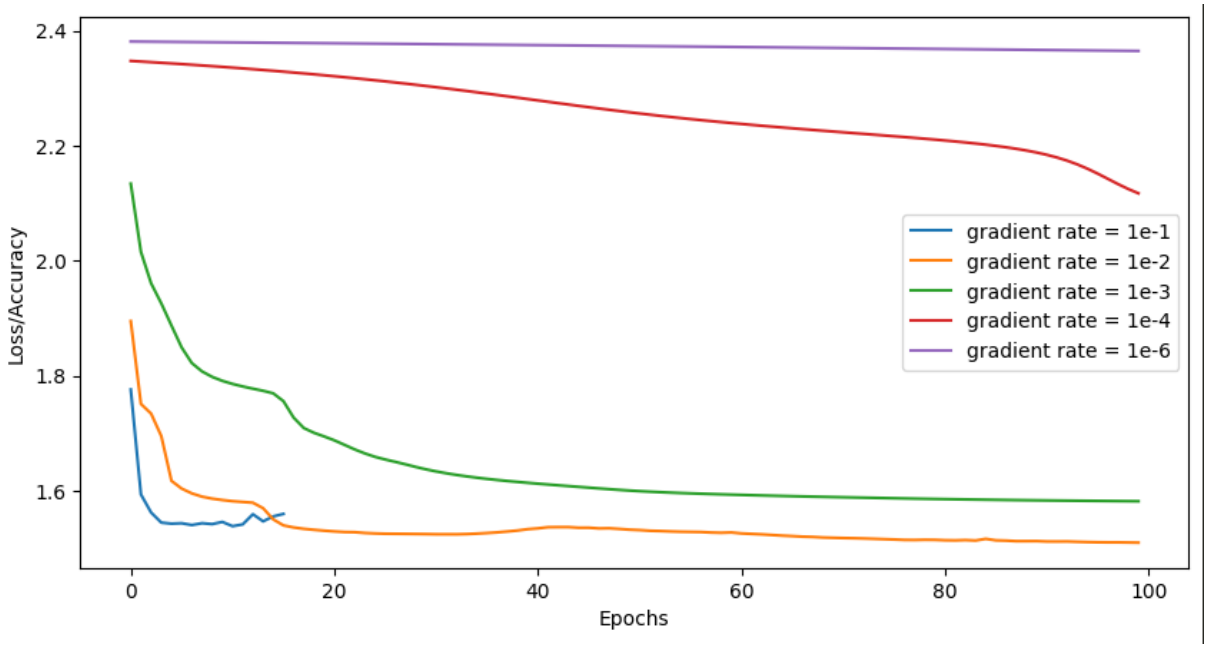


FIGURE 20 – Évolution de la moyenne des pertes pendant l'apprentissage en fonction du pas de gradient

Ces résultats montrent comment l'accuracy de classification et la moyenne des pertes évoluent en fonction du pas de gradient lors de l'entraînement du modèle. En analysant ces résultats, on peut voir que les pas de gradient optimaux (1e-2 et 1e-3) non seulement améliorent l'accuracy, mais permettent également une diminution plus rapide et plus stable des pertes. À l'inverse, des pas de gradient trop grands ou trop petits entraînent une diminution plus lente des pertes ou les maintiennent élevées, ce qui se traduit par une accuracy plus faible.

Par la suite, nous avons également testé différentes fonctions d'activation pour les couches cachées, incluant la tangente hyperbolique (Tanh) et la fonction sigmoïde. La fonction sigmoïde a donné une accuracy de 65.86%, tandis que la fonction Tanh a atteint une accuracy de 89.08%.

6 Module auto_encodeur

Dans cette section, nous avons mis en place un auto-encodeur, composé de deux parties distinctes :

— **Encodeur** : Ce composant prend une donnée en entrée et la transforme dans un espace de dimension réduite pour obtenir le code (ou la représentation latente) de l'exemple. Cette transformation est réalisée à l'aide de couches linéaires successives, de tailles décroissantes, alternées avec des fonctions d'activation non linéaires.

— **Décodeur** : Le décodeur reçoit en entrée le code de l'exemple et, généralement, utilise un réseau symétrique à l'encodeur pour reconstruire l'exemple vers sa forme originale.

L'objectif principal de ce réseau de neurones est de minimiser la fonction de coût Binary Cross Entropy, définie par :

$$\text{BCE}(y, \hat{y}) = -(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y}))$$

Lors de cette phase, nous avons réalisé les expérimentations suivantes.

6.1 Reconstruction d'images :

Dans cette partie, nous avons effectué des éxpérimentations et avons visualiser des images reconstruites par l'auto-encodeur en testant l'effet de quelques hypeparamétre : La premère ligne représente les images initiales La deuxième représente les images reconstruites après encodage :

— **Avec dimension latente pas trés réduite :**

FIGURE 21 – Visualisation des images reconstruites

Les images reconstruites sont presque identiques aux images originales, indiquant une perte d'information négligeable lors de la compression. Cela est dû aux dimensions bien choisies de la couche latente, avec `input_size = alltrainx.shape[1]`, `hidden_1 = input_size // 2`, et `output_size = hidden_1 // 2`, permettant une représentation efficace tout en préservant les détails essentiels.

— **Variation de la taille des sorties(dimension de l'espace latent) :** Lors de cette expérimentation, nous avons réduit nos images de taille 256 en une couche de dimension 100, puis en une couche de dimension 10. Cela peut expliquer les différences entre les images reconstruites, car il y a eu une perte d'information(Réduction exéssive on est descendu à une très petite dimension).

FIGURE 22 – Visualisation desimages reconstruites en utilisant une taille de sortie plus reduite

— **Variation du nombre de couches :** Ici, dans cette figure, comme les couches intermédiaires des 3 couches utilisées pour l'encodeur ne diminuent pas de manière excessive (division par 2), les images sont restées bonnes et il n'y a pas eu beaucoup de perte d'information. Cependant, quand on a utilisé plusieurs couches intermédiaires de taille très réduite (10, 15), nous avons observé une perte d'information importante et les images reconstruites ne ressemblent pas aux images initiales.

FIGURE 23 – Visualisation des images reconstruites avec plusieurs couches

— **Taille du Batch :** Nous avons exploré différentes tailles de batch lors de l'entraînement de l'auto-encodeur.Ici pour cette éxpérimentation vous avant augmenter les batch à **50** (10 avant) et les images reconstruites lors de l'utilisation d'un batch plus grand montrent une amélioration de la qualité. En augmentant la taille du batch, nous avons observé une meilleure estimation des gradients, ce qui a conduit à une convergence plus stable et plus rapide de l'autoencodeur. En conséquence, les mises à jour des poids étant moins bruitées, les reconstructions sont plus fidèles aux images originales. De plus, une plus grande taille de batch a permis d'inclure une plus grande diversité d'échantillons à chaque étape d'entraînement, aidant ainsi l'autoencodeur à mieux généraliser et à produire des reconstructions de meilleure qualité.

FIGURE 24 – Visualisation des images reconstruites en utilisant un batch de taille plus grande

— **Nombre Maximal d'Itérations :** Lors de cette expérimentation, nous avons doublé le nombre d'époques de 100 à 200, et nous remarquons que le modèle a eu un peu plus de difficulté et nous avons observé une perte d'information. Cela pourrait être justifié par le fait qu'il n'arrive pas à généraliser et qu'il a surappris.

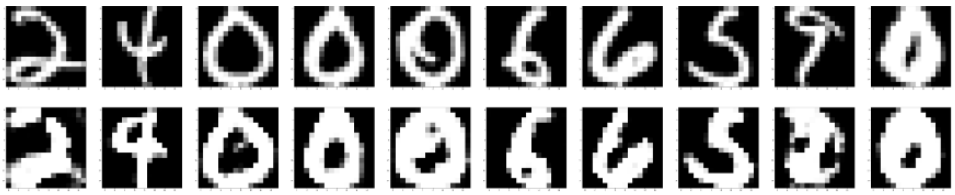


FIGURE 25 – Visualisation des images reconstruites en utilisant un nombre d'epoch élevé

6.2 K-means et T_SNE

Pour explorer le clustering, nous avons comparé la visualisation des clusters d'images à l'aide de la t-SNE avec la représentation obtenue par l'auto-encodeur. Nous avons observé qu'après encodage et décodage des images, les données ont été considérablement affectées et la distribution entre les groupes n'est plus aussi claire que sur les données initiales. Cette altération est probablement due à la perte de données consécutive à la forte compression.

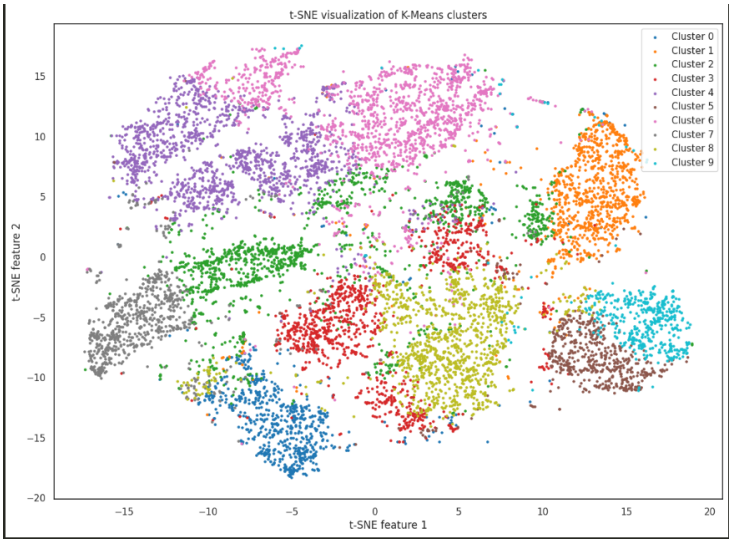


FIGURE 26 – Visualisation des clusters sur les images originales

Nous avons également visualisé la représentation t-SNE des clusters induits dans l'espace latent de l'auto-encodeur. Nous avons remarqué que cette représentation ressemblait quelque peu aux images initiales, bien que des différences subsistent en raison de la transformation appliquée par l'auto-encodeur.

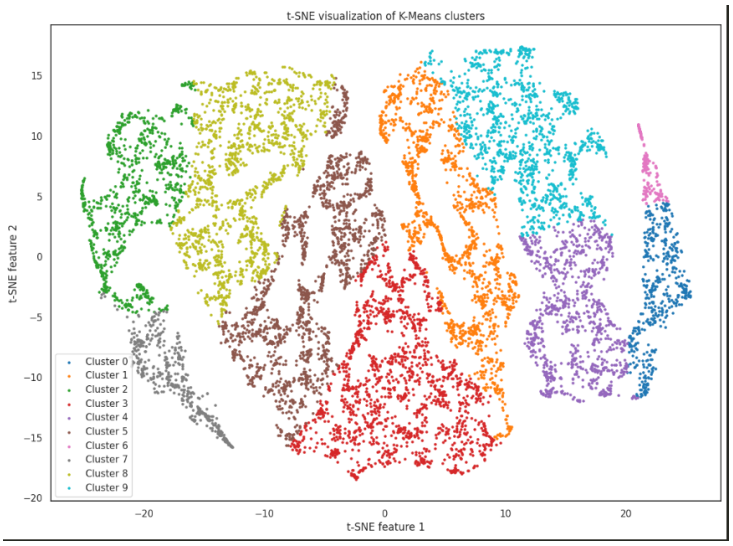


FIGURE 27 – Visualisation des clusters des représentations obtenues par l'auto-encodeur.

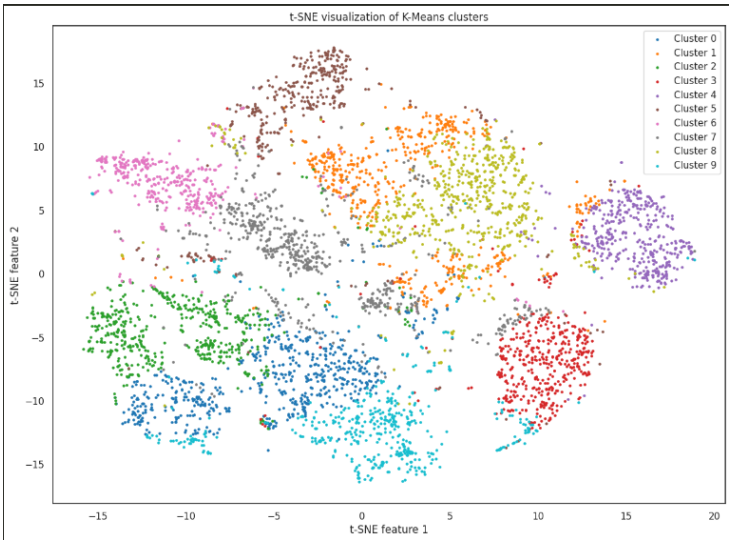
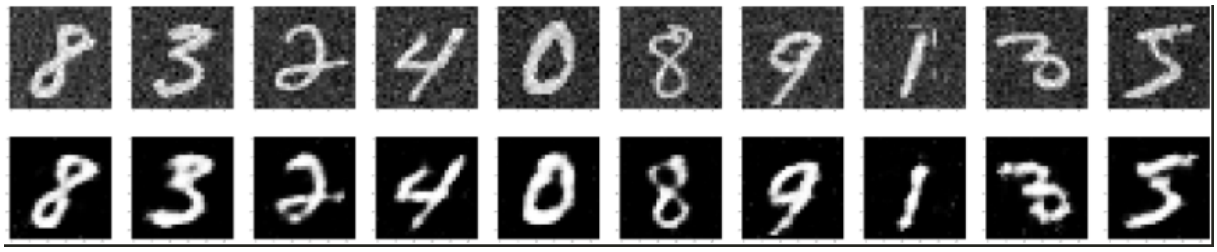


FIGURE 28 – Visualisation des clusters induits dans l'espace latent de l'auto-encodeur.

6.3 Débruitage d'images

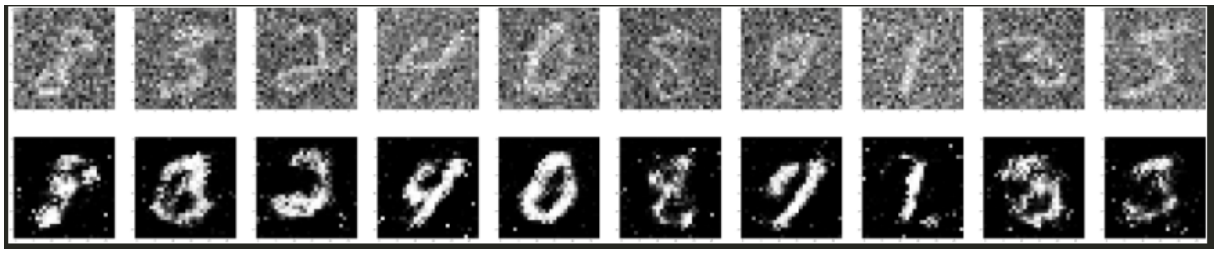
Dans cette section, nous avons étudié les performances de débruitage d'un décodeur sur des images bruitées avec divers niveaux de bruit. Nous avons d'abord entraîné le modèle sur des données bruitées, puis nous l'avons entraîné sur des données non bruitées. Cela nous a permis de comparer l'efficacité du décodeur dans les deux scénarios et d'analyser son aptitude à éliminer le bruit des images.

Après avoir analysé les résultats de ces expérimentations, nous avons constaté que le modèle entraîné sur des images bruitées distingue mieux les images par rapport au modèle entraîné sur des données non bruitées. Par exemple, pour un niveau de bruit de 0,5, le modèle entraîné sur des images bruitées parvient à reconnaître les chiffres, tandis que le modèle entraîné sur des données non bruitées n'y parvient presque pas. En outre, pour les images fortement bruitées (niveau de bruit de 0,8), il est nécessaire d'utiliser plus d'époques pour que le modèle converge lors de l'entraînement.

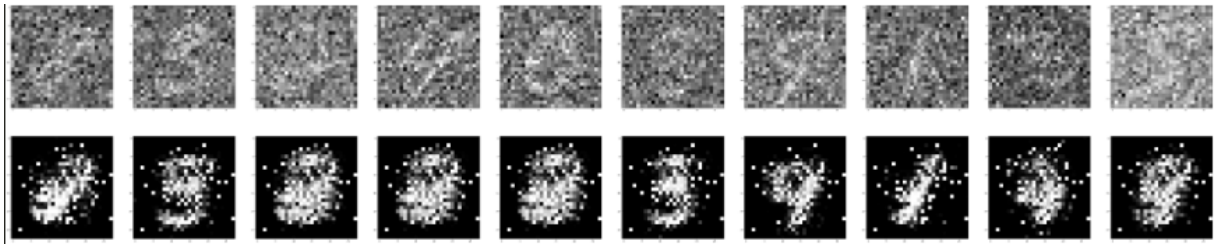


(a) noise = O.1

(b) noise = O.3

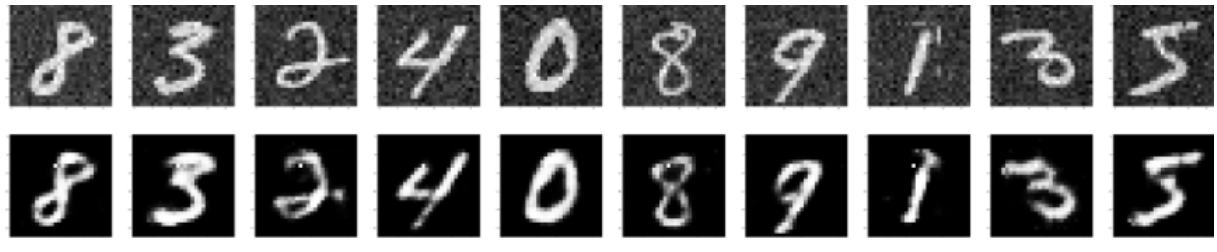


(c) noise = O.5

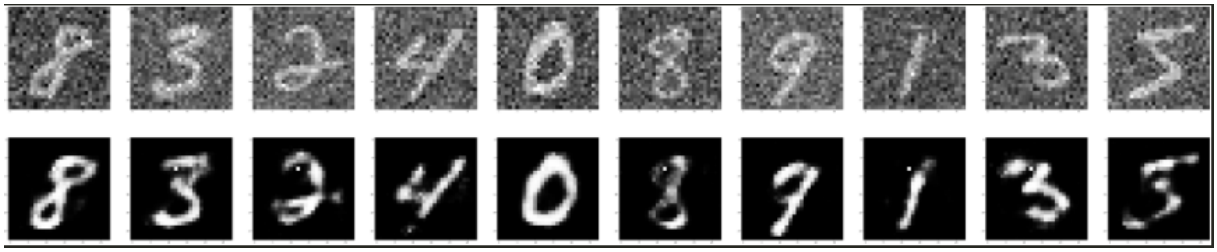


(d) noise = O.8

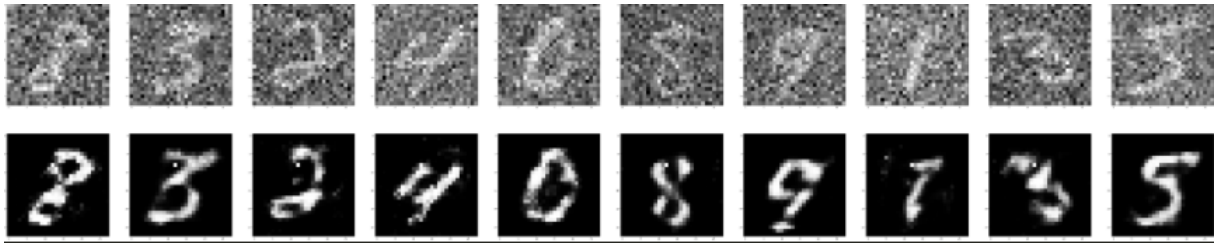
FIGURE 29 – Visualisation de le prediction des images avec un encodeur entrainé sur des images non bruitées



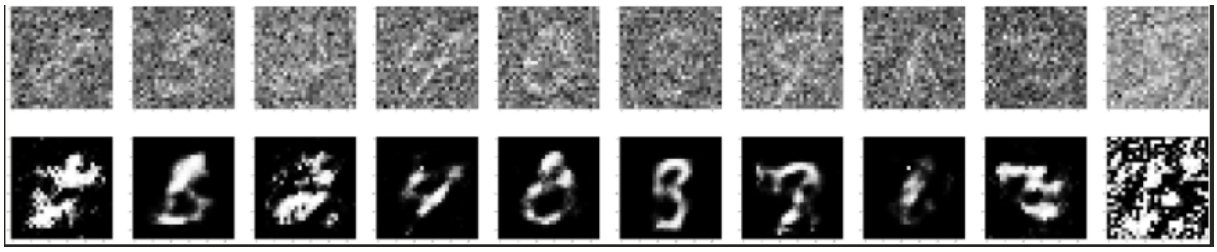
(a) noise = O.1



(b) noise = O.3



(c) noise = O.5



(d) noise = O.8

FIGURE 30 – Visualisation de le prediction des images avec un encodeur entrainé sur des images bruitées

7 Module Convolutionnelle

Dans cette partie du projet, nous avons mis en œuvre un réseau de neurones convolutif pour classifier les images de la base de données USPS. Plus précisément, nous avons utilisé une architecture de réseau de type :

`Conv1D(3,1,32) → MaxPool1D(2,2) → Flatten() → Linear(4064,100) → ReLU() → Linear(100,10)`.

Cette configuration comprend une couche convolutionnelle suivie d'une couche de max-pooling, d'une couche de flattening pour aplatir les données, et de deux couches linéaires avec une activation ReLU entre elles.

7.1 Expérimentations et Résultats

Pour des raisons de temps de calcul et de ressources, nous avons limité l'ensemble d'entraînement à 200 images. Ces images ont été divisées en mini-lots (batches) de 20 pour optimiser l'entraînement. Nous avons mené des expérimentations avec deux paramètres d'epochs différents : 100 et 250.

Copier le code

7.1.1 Avec 100 epochs

— **Accuracy** : Nous avons obtenu une précision de 0.97 pour l'ensemble d'entraînement et de 0.66 pour l'ensemble de test.

— **Courbe de Perte** : La courbe de perte montre une diminution progressive, indiquant que le modèle apprend efficacement.

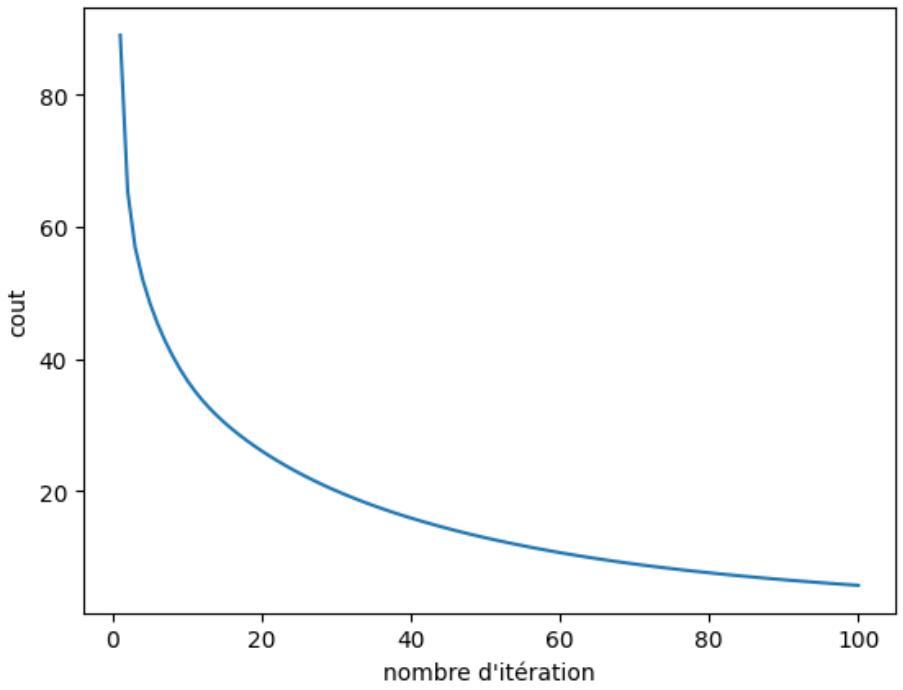


FIGURE 31 – La courbe de la loss avec 100 epochs

7.1.2 Avec 250 epochs

— **Accuracy** : Nous avons obtenu une précision de 1.00 pour l'ensemble d'entraînement et de 0.77 pour l'ensemble de test.

— **Courbe de Perte** : La courbe de perte continue de diminuer, mais avec une pente plus douce, suggérant que le gain en performance devient marginal après un certain nombre d'epochs.

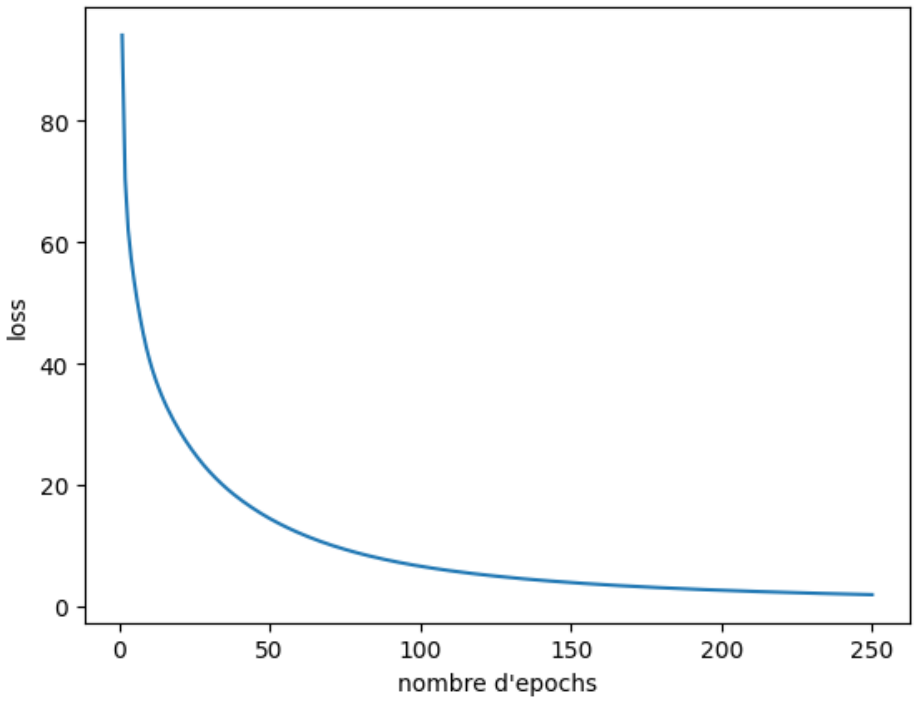


FIGURE 32 – La courbe de la loss avec 100 epochs

Les courbes de perte pour les deux configurations (100 et 250 epochs) montrent une tendance de convergence. Avec 100 epochs, la perte diminue de manière significative initialement, puis se stabilise. En augmentant à 250 epochs, la perte continue de diminuer mais à un rythme plus lent, confirmant que le modèle a appris les caractéristiques principales des données USPS et que des epochs supplémentaires n'apportent que peu d'amélioration.

Les résultats de nos expérimentations suggèrent que le modèle proposé est capable d'apprendre efficacement à partir d'un nombre limité d'images d'entraînement, atteignant une précision raisonnable de 0.77. Cependant, la performance semble plafonner au-delà de 100 epochs, indiquant qu'un ajustement supplémentaire des hyperparamètres ou une augmentation de la taille de l'ensemble d'entraînement pourrait être nécessaire pour améliorer la précision du modèle. Ces observations ouvrent des pistes pour des travaux futurs, notamment l'optimisation du modèle et l'exploration de techniques avancées pour mieux capturer les variations présentes dans les images USPS.

8 Conclusion

En conclusion, ce projet de développement et d'évaluation de divers modèles de réseaux de neurones a démontré l'efficacité et la robustesse de nos approches personnalisées en comparaison avec des implémentations standards comme celles fournies par scikit-learn. Nos expériences se sont concentrées sur plusieurs aspects clés du machine learning :

— **Régression Linéaire** : Notre modèle de régression linéaire personnalisé a montré des performances similaires à celles du modèle de scikit-learn, confirmant ainsi la validité de notre implémentation. La trajectoire des points de données a été bien capturée, attestant de la précision de notre approche.

— **Classification Linéaire** : Dans le cadre de la classification, notre classificateur linéaire a réussi à séparer deux catégories de points de manière efficace, validée par une comparaison directe avec le modèle de scikit-learn. Cette comparaison a démontré que notre modèle est capable de fournir une séparation claire entre les clusters de données.

— **Modèles Non Linéaires** : Les tests sur des données linéairement et non linéairement séparables ont montré que nos modèles non linéaires, intégrant des activations telles que TanH et Sigmoïde, fonctionnent bien pour capturer des relations complexes entre les données. Les résultats indiquent une bonne performance de ces modèles dans des contextes de séparation non linéaire, avec une convergence observable de la perte (MSELoss).

— **Module Séquentiel et Multi-classes** : Nos expérimentations avec des architectures séquentielles et des problèmes multi-classes ont mis en évidence la flexibilité et l'adaptabilité de notre implémentation modulaire. Les résultats obtenus montrent que notre modèle peut apprendre efficacement à partir d'un ensemble limité de données tout en maintenant une performance stable.

— **Autoencodeur** : L'implémentation et l'expérimentation de notre autoencodeur ont montré son efficacité pour la reconstruction et le débruitage des images. Nous avons observé que le modèle entraîné sur des images bruitées était plus performant pour le débruitage par rapport à celui entraîné sur des images non bruitées, notamment à des niveaux de bruit élevés. Les variations de la taille des couches latentes et du batch ont également montré l'importance de ces hyperparamètres pour la qualité de la reconstruction des images.

— **Réseaux Convolutionnels** : Notre réseau de neurones convolutionnel a été efficace pour la classification d'images de la base de données USPS. Les résultats expérimentaux ont montré une précision élevée pour l'ensemble d'entraînement, bien que la précision de l'ensemble de test soit restée inférieure, suggérant la nécessité d'un ajustement des hyperparamètres ou d'une augmentation de la taille de l'ensemble d'entraînement. Les courbes de perte ont indiqué une convergence satisfaisante, validant ainsi notre approche.

En résumé, ce projet a permis de valider nos approches et implémentations dans divers contextes de machine learning, démontrant ainsi leur efficacité et leur potentiel d'application dans des tâches complexes.