Projet - Réseau de neurones : DIY

Saliou Barry, Zhile Zhang

Sorbonne Université
MI DAC
Machine Learning
May 24, 2024

Contents

1 Introduction

Ce projet s'inscrit dans la dynamique actuelle du Machine Learning, visant à explorer les fondements des réseaux de neurones et à mettre en œuvre une approche modulaire inspirée des premières versions de PyTorch et des pratiques analogues. L'objectif principal est de créer une infrastructure flexible et générique pour la construction et l'entraînement de réseaux de neurones, couvrant un large éventail de tâches allant de la régression à la classification, en passant par la compression de données.

Nous adoptons une architecture modulaire qui considère chaque couche du réseau comme une entité autonome, permettant une composition aisée de réseaux complexes. Cette approche facilite également une mise à jour cohérente des poids à travers le réseau grâce aux principes de rétro-propagation.

Au cours de ce projet, nous explorerons les différentes étapes de construction d'un réseau de neurones modulaire, de l'implémentation de modules de base tels que les couches linéaires et les fonctions d'activation, à l'expérimentation avec des architectures plus avancées comme les auto-encodeurs.

2 Implémentation

2.1 Loss

Nous avons implémenté 4 fonctions de loss: MSE, Binary Cross Entropy, MultiClass Cross Entropy et LogSoftMax Cross Entropy. Nous n'avons rien à signaler car l'implémentation est directe.

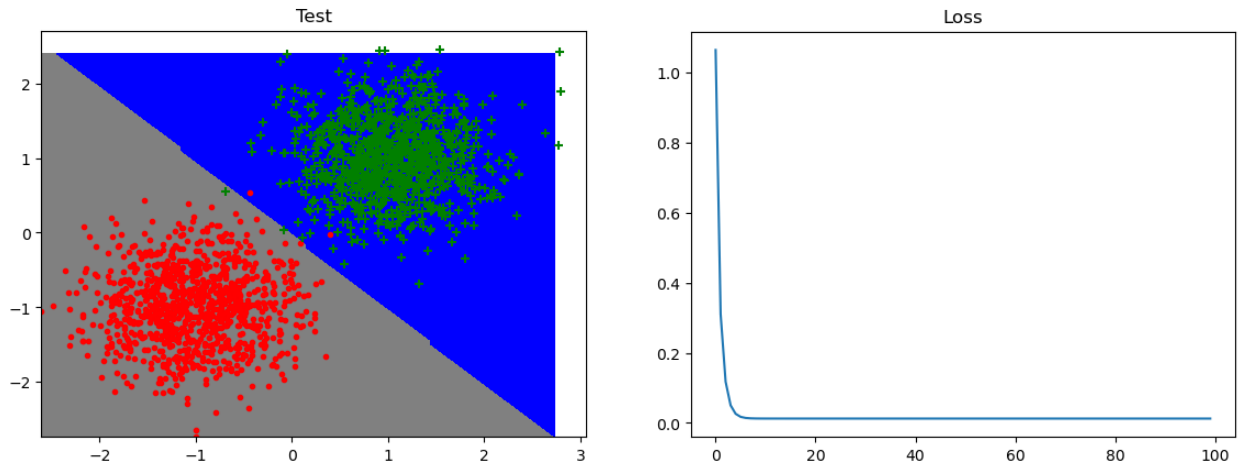
2.2 Module

Le module constitue l'unité de base dans notre architecture de réseau de neurones. Il encapsule les opérations fondamentales effectuées sur les données en entrée pour produire une sortie. Dans cette structure, nous considérons qu'une fonction d'activation est également un module.

2.2.1 Linear

La classe Linear représente un module linéaire dans notre réseau. La sortie y est calculée comme le produit de la matrice d'entrée X et de la transposée de la matrice de poids W, selon l'équation $y = XW^T$.
Un paramètre booléen, *bias*, permet l'ajout d'un biais à la sortie. Si ce paramètre est vrai, un biais est ajouté à la sortie. Ce biais peut être initialisé aléatoirement en suivant une

distribution normale si spécifié, ou à zéro. Dans le cas où le biais est activé, il est ajouté à la sortie selon l'équation $y = XW^T + b$.
Les poids de la couche sont également initialisés de la même manière que le biais.
Après avoir mis en place le modèle linéaire, nous avons procédé à son test sur des données quelconques, séparables linéairement. Pour cela, nous avons utilisé une seule couche linéaire et une fonction de perte MSELoss().



(a) frontière de décision (b) courbe loss

Figure 1: visualisation de la frontière de décision ainsi que l'évolution de la fonction de perte au fil de l'apprentissage.

2.2.2 Fonctions d'Activation

Les fonctions d'activation jouent un rôle crucial dans les réseaux de neurones, introduisant une non-linéarité dans le modèle et permettant ainsi la capture de relations complexes dans les données. Nous avons implémenté 4 fonctions d'activation: : la tangente hyperbolique (Tanh), la sigmoïde, la softmax et la ReLU (Rectified Linear Unit). Chacune de ces fonctions a des propriétés et des comportements différents, ce qui les rend adaptées à différentes tâches et architectures de réseaux neuronaux.

- **Tanh (Tangente Hyperbolique)**

La fonction d'activation TanH est définie comme $f(x) = \tanh(x)$, où $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Elle produit des valeurs dans l'intervalle [-1, 1], ce qui en fait une fonction centrée autour de zéro. La TanH est souvent utilisée dans les couches cachées des réseaux de neurones pour introduire une non-linéarité tout en maintenant la sortie centrée.
Nous avons calculé la rétropropagation du gradient pour la softmax est calculée comme suit:

$$\frac{\partial L}{\partial x} = \delta(1 - \tanh^2(x))$$

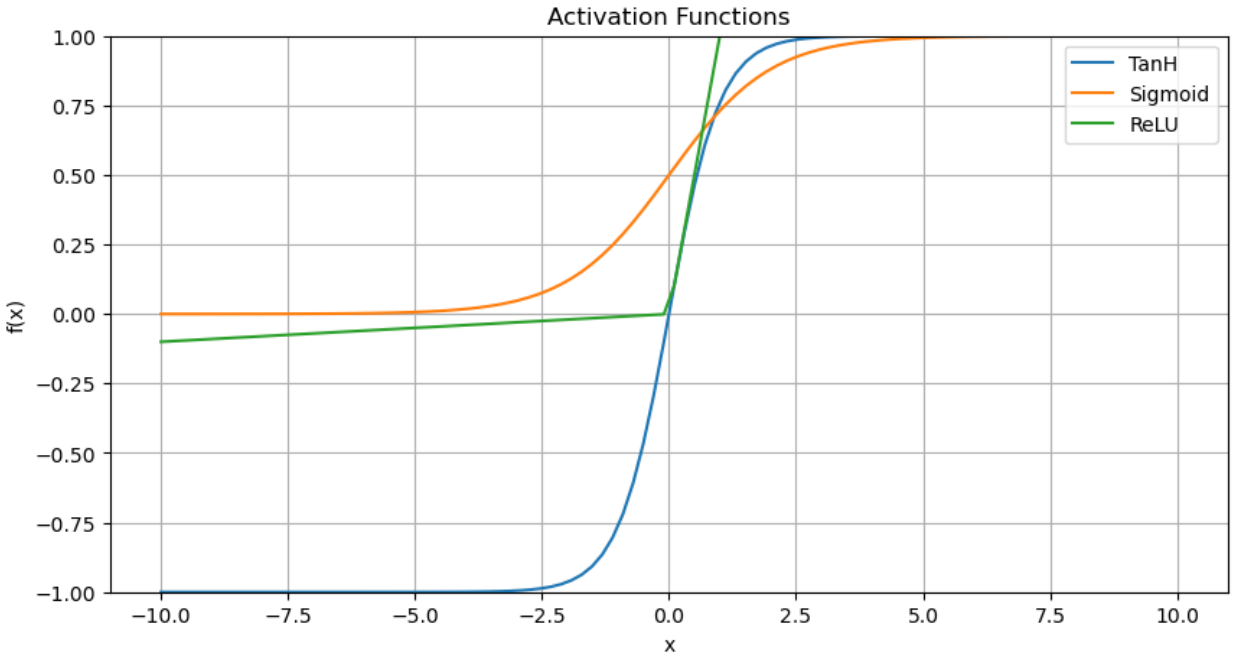


Figure 2: Courbes des differentes fonctions d'activation

- **Sigmoïde**

La fonction sigmoïde est définie comme $f(x) = \frac{1}{1+e^{-x}}$. Elle produit des valeurs dans l'intervalle [0, 1], ce qui la rend appropriée pour modéliser des probabilités ou des sorties binaires. Cependant, la sigmoïde souffre du problème de disparition du gradient pour des entrées très positives ou très négatives.
Nous avons calculé la rétropropagation du gradient pour la softmax est calculée comme suit:

$$\frac{\partial L}{\partial x} = \delta(1 - \sigma(x)) \cdot \sigma(x)$$

- **Softmax**

La fonction softmax est couramment utilisée dans la couche de sortie des réseaux de neurones pour classifier plusieurs classes. Elle transforme les scores bruts en une distribution de probabilité sur les différentes classes. La softmax est définie comme $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$ pour chaque élément x_i du vecteur d'entrée x.
Nous avons calculé la rétropropagation du gradient pour la softmax est calculée comme suit:

$$\frac{\partial L}{\partial x_i} = \delta_i \left(1 - \text{softmax}(x_i)\right) \cdot \text{softmax}(x_i)$$

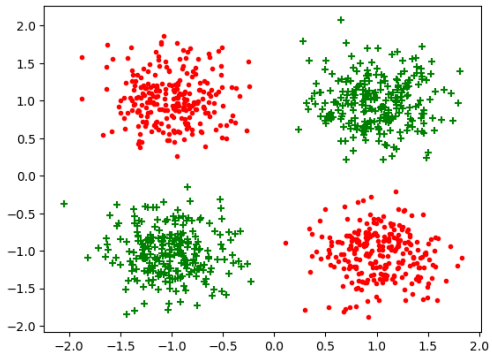
- **ReLU (Rectified Linear Unit)**

La fonction ReLU est définie comme $f(x) = \max(0, x)$. Elle introduit une non-linéarité en remplaçant les valeurs négatives par zéro. La ReLU est largement utilisée dans les couches

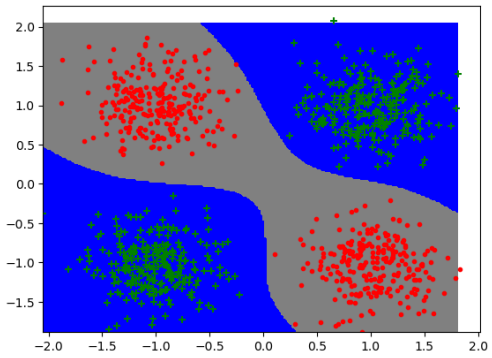
cachées pour son efficacité de calcul et sa capacité à atténuer le problème de disparition du gradient.
Nous avons calculé la rétropropagation du gradient pour la softmax est calculée comme suit:

$$\frac{\partial L}{\partial x} = \delta \cdot \begin{cases} 1 & \text{si } x > 0 \\ \alpha & \text{sinon} \end{cases}$$

Après avoir exploré le modèle linéaire de base, nous avons étendu notre expérimentation en implémentant un réseau à deux couches linéaires. Dans cette nouvelle configuration, nous avons introduit une fonction d'activation tangente hyperbolique entre les deux couches et une fonction d'activation sigmoïde à la sortie.
$Linear(input\text{-}size, 50, bias = True) \rightarrow TanH() \rightarrow Linear(50, dim, bias = True) \rightarrow Sigmoid$



(a) Données originale



(b) Frontière de décision

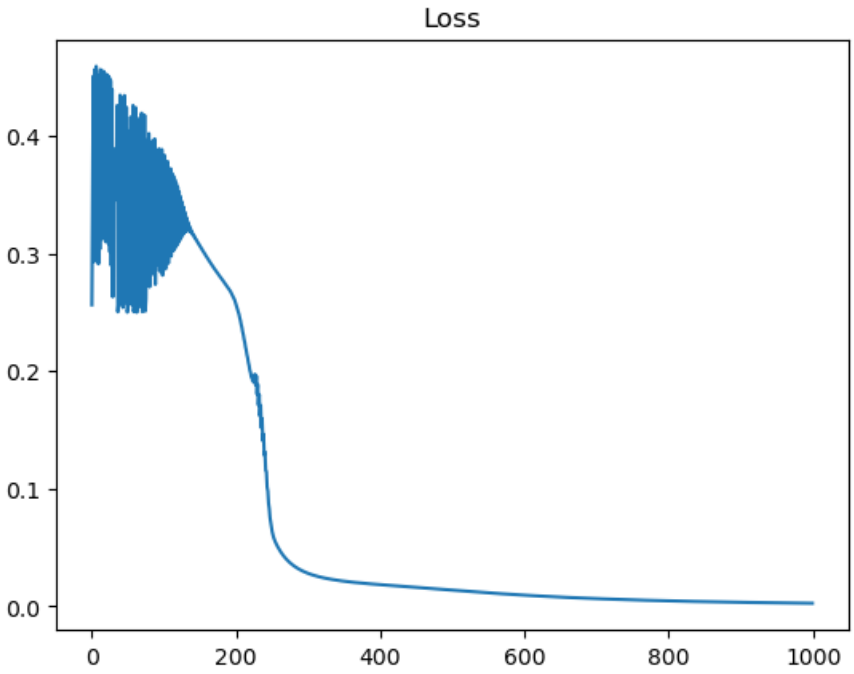


Figure 4: Courbe Loss

2.2.3 Modèle Séquentiel et Optimiseur

La classe `Sequential` permet de composer plusieurs modules en séquence, facilitant ainsi la construction de réseaux de neurones multicouches. Elle est constituée par quatre fonctions:

- $forward$: La méthode $forward$ effectue une passe en avant à travers tous les modules du réseau. Elle prend en entrée X, les données d'entrée du réseau, puis itère sur chaque module dans la liste $self.modules$. Pour chaque module, elle appelle sa méthode $forward$ avec les données d'entrée X. Elle stocke également les résultats intermédiaires dans $self.inputs$ pour une utilisation ultérieure lors de la rétro-propagation. Enfin, elle renvoie la sortie du dernier module.

- *backward*: La méthode *backward* est utilisée pour effectuer la rétro-propagation à travers tous les modules du réseau. Elle prend *input*, qui est l'entrée initiale du réseau, et *delta*, qui est le gradient de la perte par rapport à la sortie du réseau. Elle itère ensuite sur les modules du réseau dans l'ordre inverse et appelle leurs méthodes *backward_update_gradient* et *backward_delta* avec les entrées correspondantes. À chaque itération, elle met à jour le delta en fonction du module actuel.

- *backward_delta*: Cette méthode calcule le delta pour la rétro-propagation du dernier module au premier. Elle prend *input*, l'entrée initiale du réseau, et *delta*, le gradient de la perte par rapport à la sortie du réseau. Elle itère sur les modules dans l'ordre

inverse, passant le delta à travers chaque module en appelant leurs méthodes $forward$ et $backward\_delta$.

- $update\_parameters$: Cette méthode met à jour les paramètres de tous les modules du réseau en utilisant la méthode $update\_parameters$ de chaque module, avec un taux d'apprentissage donné.

- $zero\_grad$: Cette méthode réinitialise les gradients de tous les modules du réseau en utilisant la méthode $zero\_grad$ de chaque module. Cela est généralement fait avant chaque passe en avant pour garantir que les gradients sont recalculés correctement.

Ensuite, nous avons implémenté la classe Optimizer, qui prend en compte un réseau de type Sequential, une fonction de perte et un pas de descente de gradient. Cette classe implémente deux méthodes :

- step : Cette méthode effectue un pas de descente de gradient.

- sgd: Cette méthode effectue une descente de gradient mini-batch jusqu'à convergence. Le paramètre taille_batch peut être ajusté pour transformer la méthode en une descente de gradient batch ou stochastique.

Après avoir mis en place la classe Sequential et l'optimiseur, nous avons testé notre modèle sur des données multiclasse, notamment en utilisant le jeu de données USPS. Pour cela, nous avons utilisé le réseau suivant :

```
network =Sequential(
            Linear(input, 128),
            TanH(),
            Linear(128, 64),
            TanH(),
            Linear(64, 32),
            TanH(),
            Linear(32, 10))
```

Nous avons également utilisé la fonction de perte LogSoftMaxCrossEntropy pour évaluer les performances du modèle.
Nous avons obtenu une accuracy de 97.87% sur l'ensemble d'entraînement et de 91.72% sur l'ensemble de test. Pour une meilleure visualisation de ces résultats, des graphiques montrant l'évolution de la précision au cours de l'apprentissage sont présentés ci-dessous :

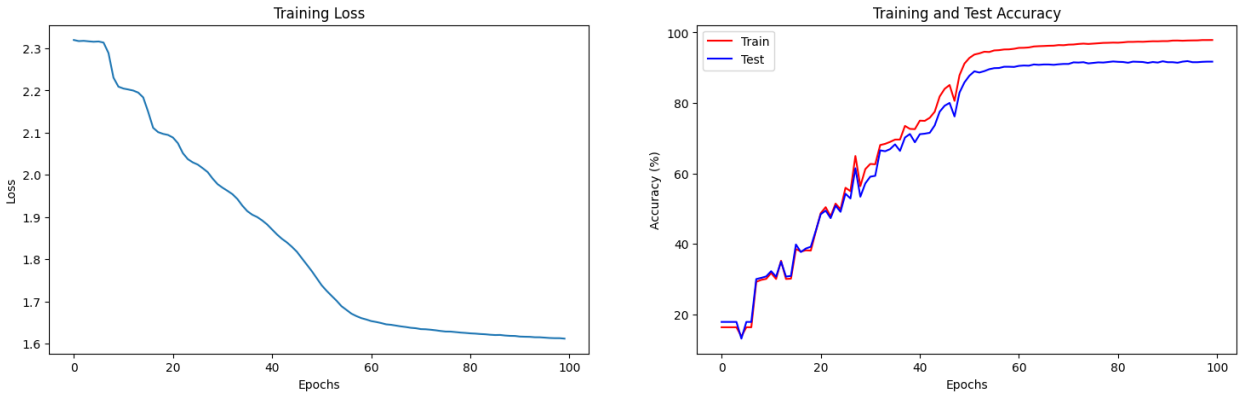


Figure 5: Evolution des différentes courbes pendant l'apprentissage

3 Auto-encodeur

Dans cette partie, notre objectif principal est de construire des réseaux encodeur-décodeur capables de prendre une image originale en entrée de l'encodeur, puis d'utiliser le décodeur pour reconstruire l'image, en s'efforçant de faire en sorte que l'image reconstruite ressemble autant que possible à l'image originale. La comparaison entre les réseaux repose sur la performance des images reconstruites.

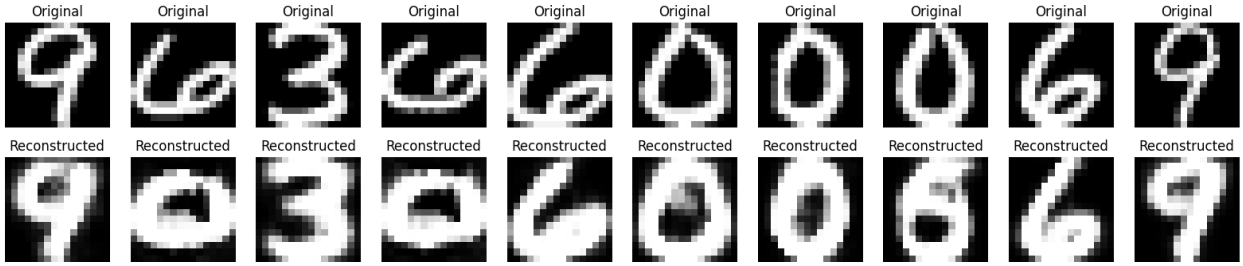
3.1 Reconstruiction

3.1.1 Données USPS

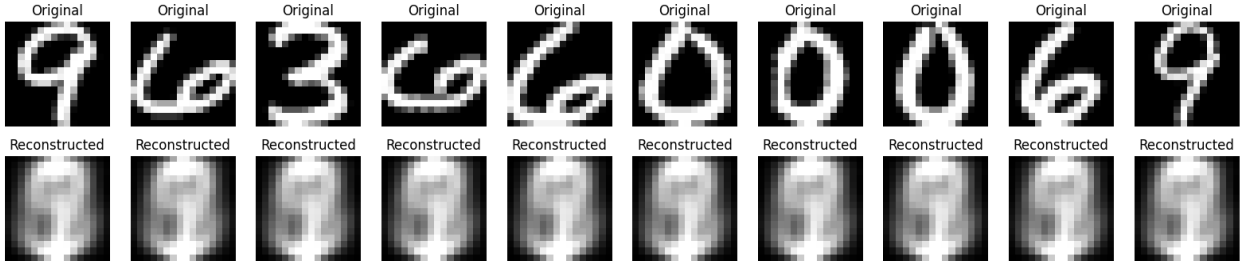
Nous avons d'abord testé avec le jeu de données USPS, en créant deux réseaux d'auto-encodeurs, l'un simple et l'autre plus complexe.

- Réseau simple: `Linear(256, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()` `ET Linear(64, 128)` → `TanH()` → `Linear(128,256)` → `Sigmoid()`
- Réseau complexe: `Linear(256, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()` → `Linear(64, 32)` → `TanH()` `ET Linear(32, 64)` → `TanH()` → `Linear(64,128)` → `TanH()` → `Linear(128,256)` → `Sigmoid()`

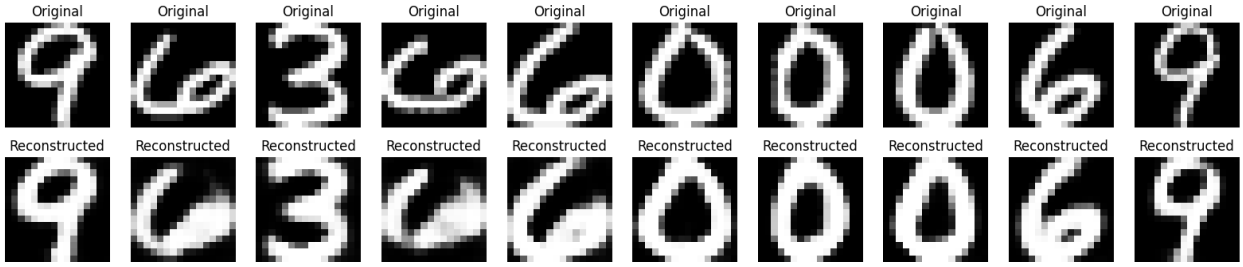
Pour ces deux réseaux, nous avons fixé la taille de batch à 64 et le taux d'apprentissage à $1e-3$ pour l'entraînement. Nous avons utilisé deux types de fonctions de perte : une fonction MSE(Mean Squared Error) et une fonction BCE(Binary Cross Entropy). Par conséquent, nous disposons de quatre résultats de reconstruction des images à présenter :



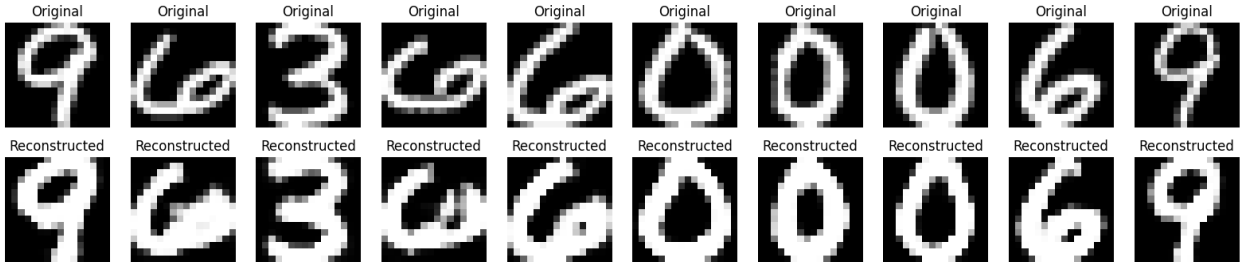
(a) Réseau simple avec la fonction de perte MSE()



(b) Réseau complexe avec la fonction de perte MSE()



(c) Réseau simple avec la fonction de perte BCE()



(d) Réseau complexe avec la fonction de perte BCE()

Figure 6: Reconstruction des images de USPS

Selon ces résultats, nous pouvons clairement voir que la meilleure combinaison est celle du réseau simple avec la fonction de perte BCE, suivie par le réseau simple avec la fonction de perte MSE, puis par le réseau complexe avec la fonction de perte BCE. La combinaison du réseau complex avec la fonction de perte MSE donne les résultats les moins satisfaisants, avec pratiquement aucune restauration des images originales.

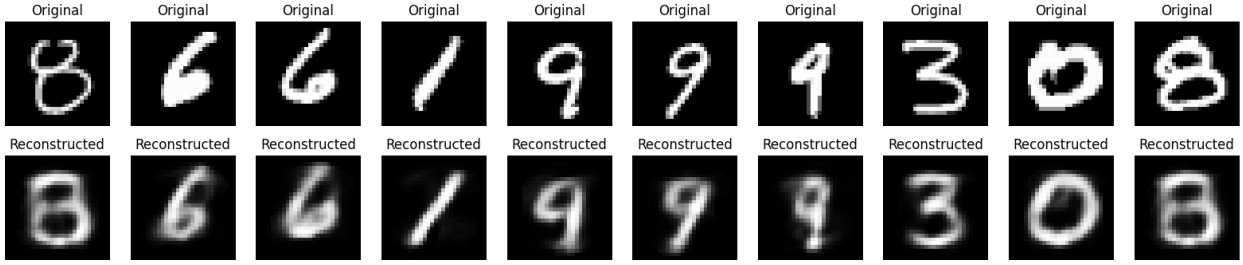
3.1.2 Données MNIST

Nous avons également testé le jeu de données MNIST, qui est aussi un jeu de données de chiffres manuscrits, mais il diffère de USPS, ce qui nous permet de faire des comparaisons et des contrastes entre les deux jeux de données.

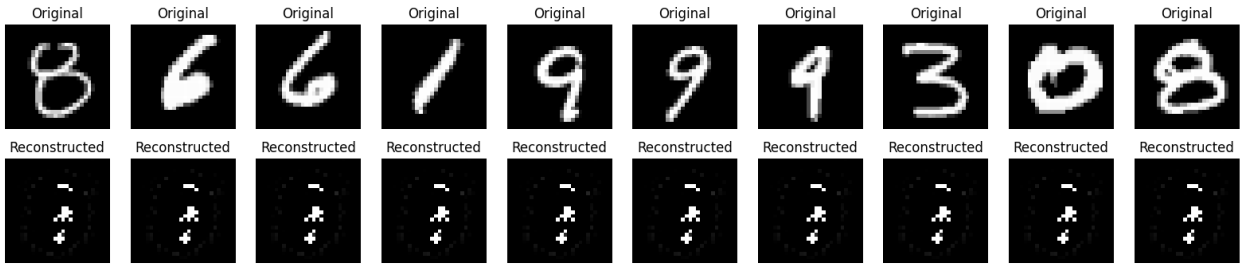
De même, nous avons utilisé deux réseaux : un simple et un complexe. Le réseau simple a seulement modifié la taille de l'entrée, tandis que le réseau complexe, en plus de changer la taille de l'entrée, a intégré plus de composants au modèle.

- Réseau simple: `Linear(784, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()`
  ET `Linear(64, 128)` → `TanH()` → `Linear(128,784)` → `Sigmoid()`
- Réseau complexe: `Linear(784, 512)` → `TanH()` → `Linear(512, 256)` → `TanH()`
  → `Linear(256, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()`
  ET `Linear(64, 128)` → `TanH()` → `Linear(128,256)` → `TanH()` → `Linear(256,512)`
  → `TanH()` → `Linear(512,784)` → `Sigmoid()`

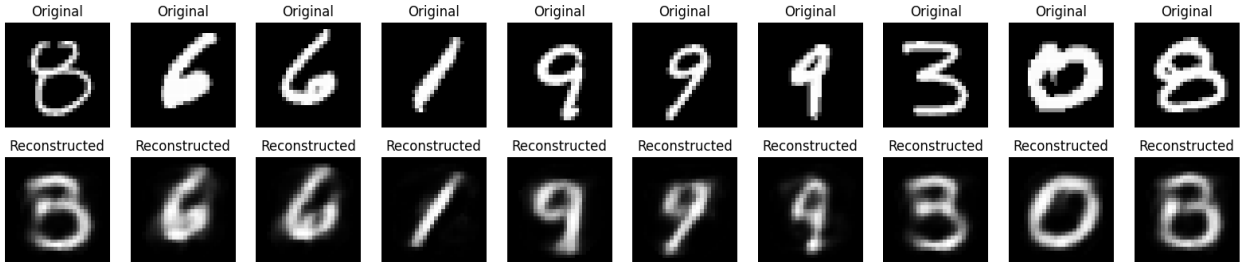
Nous avons suivi les mêmes étapes expérimentales que pour le jeu de données USPS. Les images reconstruites du jeu de données MNIST obtenues sont présentées ci-dessous.



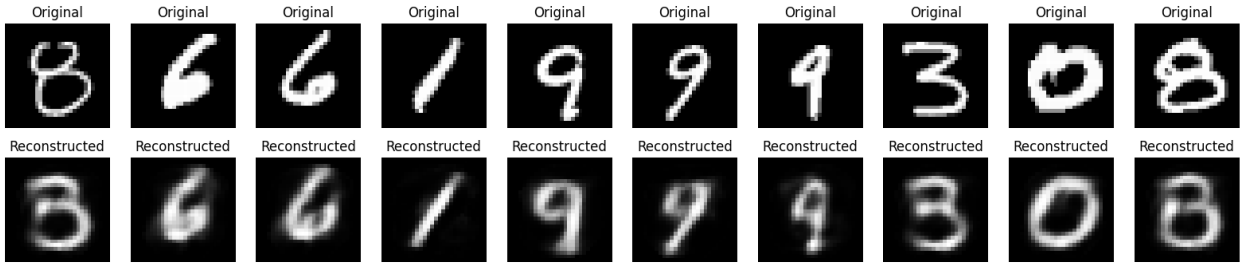
(a) Réseau simple avec la fonction de perte MSE()



(b) Réseau complexe avec la fonction de perte MSE()



(a) Réseau simple avec la fonction de perte BCE()



(b) Réseau complexe avec la fonction de perte BCE()

Figure 8: Reconstruiction de MNIST

En analysant les résultats de la reconstruction des données MNIST, il est évident que l'utilisation de la fonction de perte BCE() produit des reconstructions de meilleure qualité par rapport à l'utilisation de la fonction de perte MSE(). Cette observation reste cohérente, que ce soit pour notre réseau neuronal simple ou complexe. La qualité des reconstructions, mesurée en termes de fidélité visuelle aux images d'origine, est nettement supérieure lorsque la fonction de perte BCE() est employée. Ce constat suggère que la BCE(), mieux adaptée à la nature binaire des données MNIST, permet une meilleure capture des détails et des caractéristiques des images reconstruites.

3.2 Visualisation

Pour visualiser les représentations latentes afin de comprendre plus intuitivement les caractéristiques apprises par l'encodeur, nous avons utilisé la technique de réduction de dimensionnalité t-SNE, qui mappe les données de haute dimension en deux dimensions pour la visualisation. Puisque MNIST et USPS sont tous deux des jeux de données de chiffres manuscrits, il suffit de visualiser l'un d'entre eux. Les données que nous avons utilisées sont les résultats après l'apprentissage par l'encodeur sur les ensembles de données d'entraînement et de test de USPS.

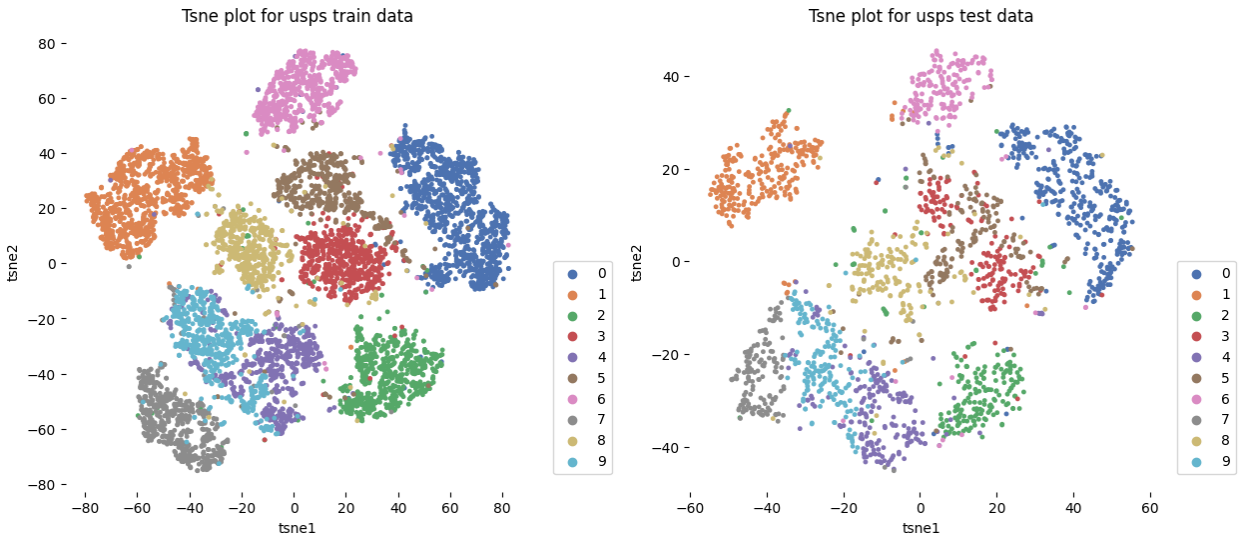


Figure 9: Tsne for usps (Réseau complexe BCE)

D'après ces deux graphiques de visualisation t-SNE, il est évident que le réseau d'encodage capture et distingue efficacement les caractéristiques des différents chiffres, aussi bien dans les données d'entraînement que dans les données de test, formant des clusters clairs où les points à l'intérieur de chaque cluster sont regroupés de manière dense, indiquant une grande similarité intra-classe. Dans ces clusters, les chiffres ayant une structure unique (comme 0, 1, 6) montrent une bonne séparation, tandis que les chiffres visuellement similaires (comme 4 et 9) se chevauchent légèrement dans certaines zones, ce qui est tout à fait normal. De plus, les graphiques t-SNE des ensembles de données d'entraînement et de test montrent une cohérence, reflétant la forte capacité de généralisation de notre encodeur.

3.3 Débruitage

Pour étudier la capacité de débruitage de l'auto-encodeur, nous avons décidé d'ajouter un bruit gaussien à l'ensemble de test. Nous avons également tenté d'introduire du bruit durant la phase d'entraînement, mais les expériences ont montré que cela n'était pas efficace, nous avons donc conservé uniquement les résultats des expériences où le bruit était ajouté au jeu de test. De plus, nous avons examiné les performances de l'auto-encodeur avec différentes valeurs de σ, c'est-à-dire avec différents niveaux d'intensité du bruit, pour observer comment l'auto-encodeur se comportait sous divers degrés de perturbation.

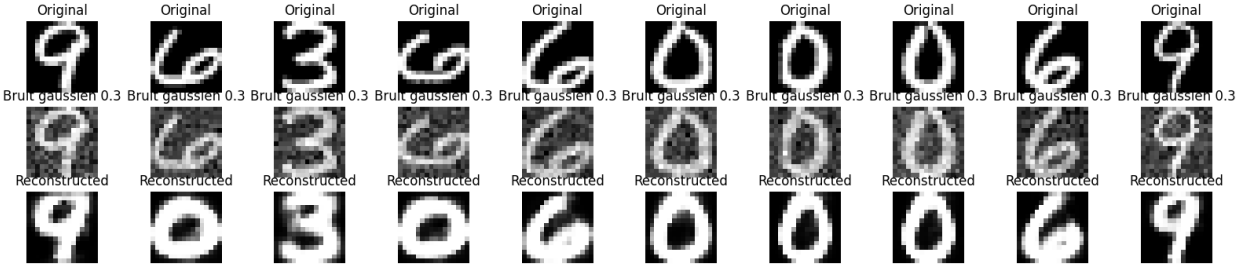
Étant donné que la combinaison du réseau complexe avec la fonction de perte MSE n'a pas donné de bons résultats, nous avons décidé de ne plus tester cette combinaison.

3.3.1 Données USPS

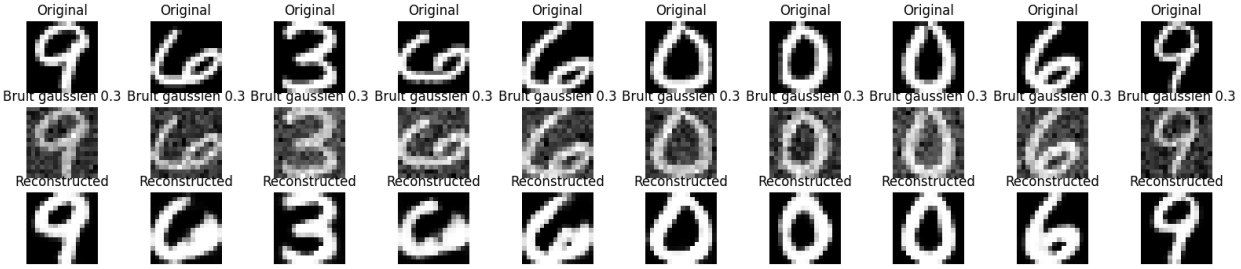
Pour le jeu de données USPS, nous avons testé deux niveaux différents de bruit gaussien, avec des valeurs de σ à 0.3 et 0.4. Sous ces deux intensités de bruit, nous avons observé que les trois combinaisons étaient toujours capables de reconstruire les images originales, même si des erreurs étaient présentes. Par exemple, la combinaison du réseau simple et de la fonction de perte MSE a complètement mal reconstruit une image d'un 6 irrégulier en un 0. Dans les cas précédents sans bruit, l'image ressemblait légèrement à un 0, mais l'ajout de bruit a conduit à une reconstruction incorrecte en 0. Toutefois, la plupart des chiffres étaient correctement reconstruits. La combinaison qui a montré les meilleures performances était celle du réseau complexe avec la fonction BCE, qui était pratiquement non affectée par le bruit.

3.3.2 Données MNIST

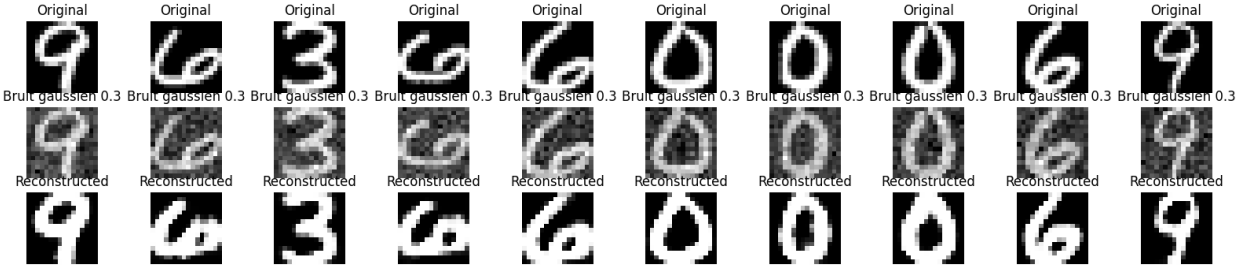
Pour le jeu de données MNIST, nous avons seulement testé avec un niveau de bruit de $\sigma = 0.3$, car dans ces conditions, le réseau simple n'était plus capable de reconstruire correctement les images. Même en ajustant le niveau de bruit à 0.1 ou 0.2, le résultat était le même. Cependant, le réseau complexe a continué à bien performer, réussissant à reconstruire avec succès les chiffres des images originales.



(a) Réseau simple avec la fonction de perte MSE()

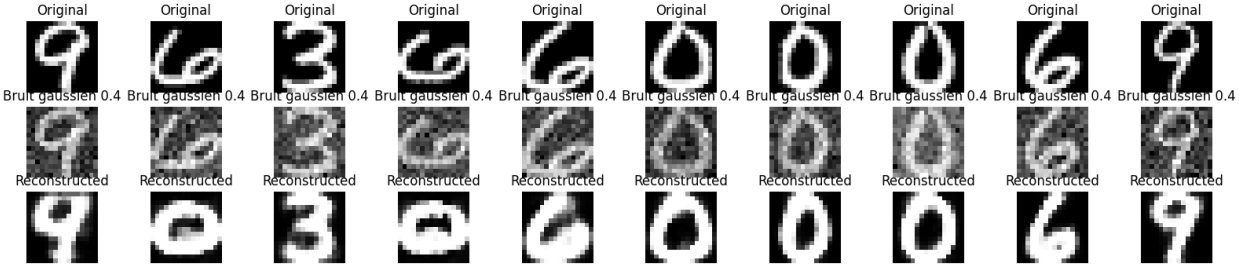


(b) Réseau simple avec la fonction de perte BCE()

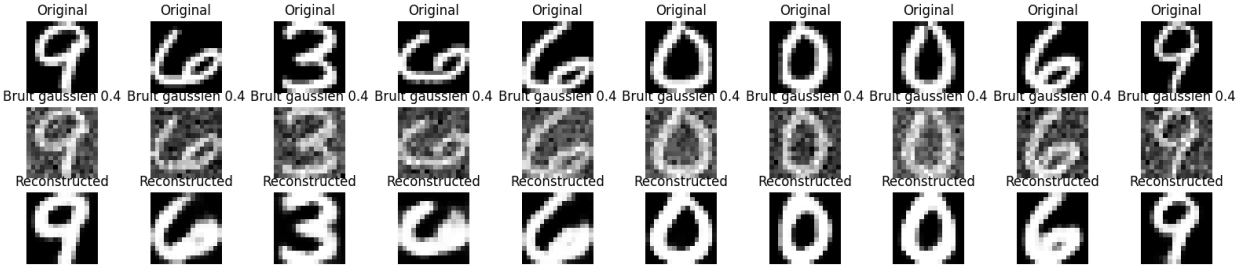


(c) Réseau complexe avec la fonction de perte BCE()

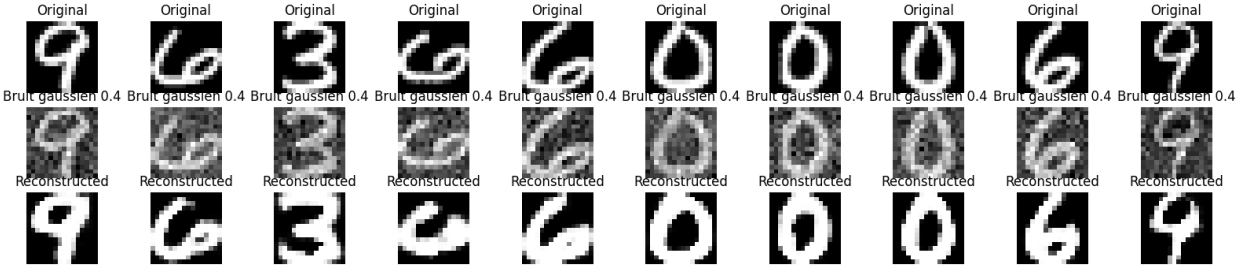
Figure 10: USPS débruitées avec $\sigma = 0.3$



(a) Réseau simple avec la fonction de perte MSE()

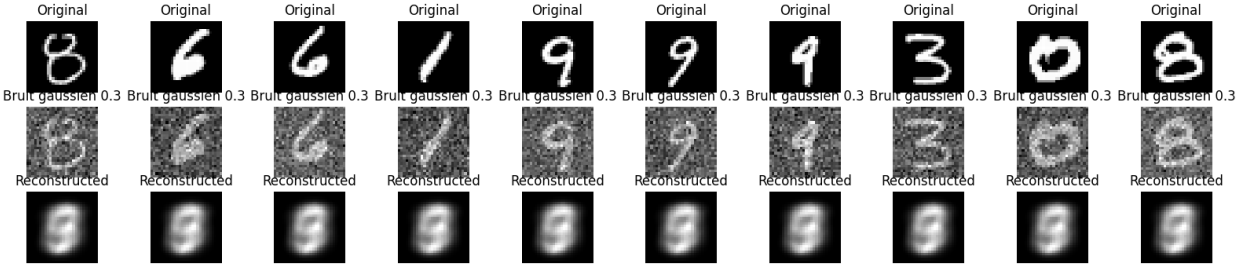


(b) Réseau simple avec la fonction de perte BCE()

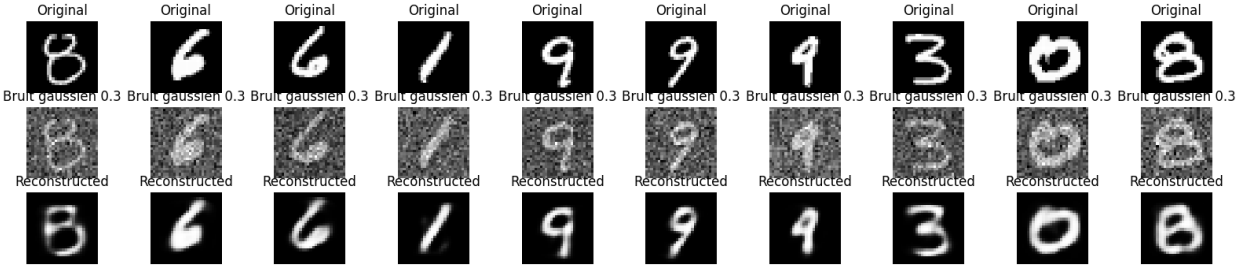


(c) Réseau complexe avec la fonction de perte BCE()

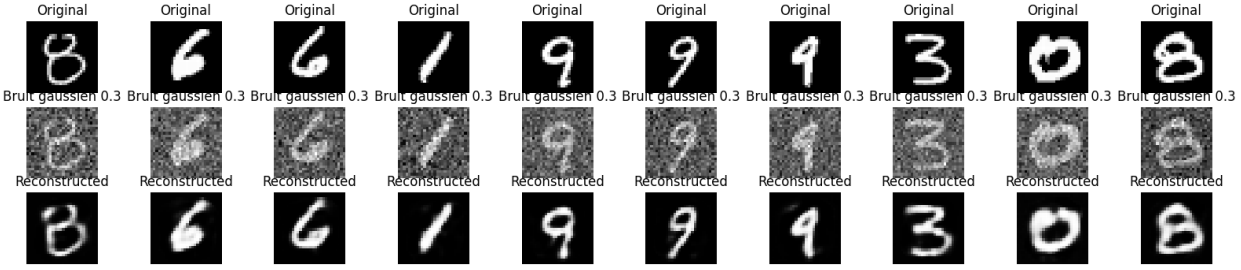
Figure 11: USPS débruitées avec $\sigma = 0.4$



(a) Réseau simple avec la fonction de perte MSE()



(b) Réseau simple avec la fonction de perte BCE()



(c) Réseau complexe avec la fonction de perte BCE()

Figure 12: MNIST débruitées avec $\sigma = 0.3$

4 Convolution

4.1 Convolution 1D

La classe Convolution1D implémente une couche de convolution en une dimension, souvent utilisée dans les réseaux de neurones pour traiter des données séquentielles telles que des signaux ou des séries temporelles. Elle réalise des opérations de convolution sur les entrées pour extraire des caractéristiques importantes tout en réduisant la dimensionnalité de ces entrées.

4.1.1 Explication des fonctions de conv1D et Maxpool1D

- ***Conv1d forward*** : La méthode forward effectue l'opération de convolution sur les données d'entrée X. Elle utilise la fonction np.lib.stride_tricks.as_strided pour créer une vue sur la matrice d'entrée en utilisant une nouvelle forme et un nouveau pas, ce qui permet de générer des fenêtres de convolution à partir de l'entrée X. Ensuite, elle utilise np.einsum pour effectuer la somme de produits tensoriels entre les données d'entrée et les poids du filtre. Cela produit la sortie de convolution de taille (batch, $(\frac{d-k_{size}}{stride}+1)$, canaux de sortie).

- ***Conv1d backward_delta*** : La méthode backward_delta effectue la rétropropagation du gradient à travers le module de convolution en une dimension. Elle prend en entrée les données d'entrée X et le gradient de la sortie (delta). Elle calcule le gradient par rapport aux données d'entrée en utilisant np.einsum pour effectuer la somme de produits tensoriels entre le gradient de sortie et les poids du filtre. Enfin, elle utilise np.add.at pour accumuler les gradients dans le tableau de résultats afin de propager le gradient aux positions appropriées du tenseur d'entrée X.

- ***Conv1d backward_update_gradient*** : La méthode backward_update_gradient met à jour les gradients des poids du filtre en utilisant le gradient de sortie et les données d'entrée. Elle utilise np.lib.stride_tricks.as_strided pour créer une vue sur les données d'entrée en utilisant une nouvelle forme et un nouveau pas, ce qui permet de générer des fenêtres correspondant à la taille du noyau. Ensuite, elle utilise np.einsum pour calculer le gradient par rapport aux poids du filtre. Le gradient calculé est ajouté au gradient existant des poids du filtre.

- ***MaxPool1D forward***: La méthode forward effectue l'opération de max pooling sur les données d'entrée X. Elle utilise np.lib.stride_tricks.as_strided pour créer une vue sur la matrice d'entrée en utilisant une nouvelle forme et un nouveau pas, ce qui permet de générer des fenêtres de pooling à partir de l'entrée X. Ensuite, elle utilise np.max

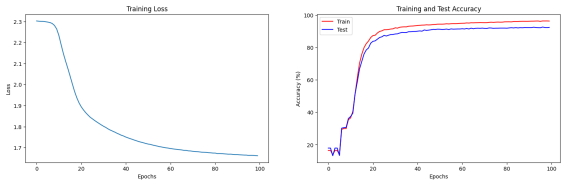
pour calculer la valeur maximale dans chaque fenêtre de pooling le long de l'axe des dimensions de la fenêtre de pooling, produisant ainsi la sortie de max pooling.

- ***MaxPool1D backward_delta***: La méthode backward_delta effectue la rétropropagation du gradient à travers le module de max pooling en une dimension. Elle prend en entrée les données d'entrée X et le gradient de la sortie (delta). Elle récupère l'indice de la valeur maximale de chaque fenêtre de pooling à partir de la sortie de max pooling. Ensuite, elle calcule les coordonnées de départ de chaque fenêtre de pooling dans l'entrée originale. Enfin, elle ajoute les gradients aux positions des valeurs maximales dans la matrice de résultats res en utilisant np.add.at, assurant ainsi la propagation correcte du gradient.

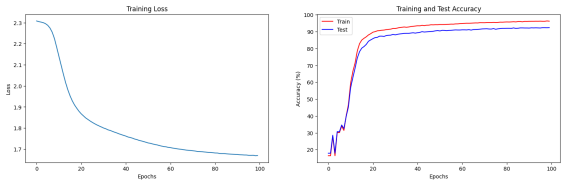
On a utilisé le réseau 1: `Conv1D(3,1,32,1)` → `MaxPool1D(2,2)` → `Flatten()` → `Linear(4024,100)` → `ReLU()` → `Lineaire(100,10)`

Pour nos entraînements et tests à venir, nous utiliserons la fonction de perte LogSoftMaxCrossEntropy et les entraînons d'abord 100 epochs, principalement pour identifier les meilleurs paramètres d'entraînement et la meilleure architecture réseau. Ensuite, nous entraînerons le modèle optimal sur 500 époques pour observer la précision finale des tests.

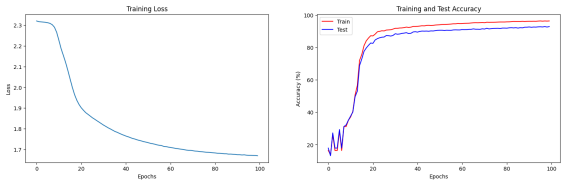
4.1.2 Évaluation avec des tailles de batch différentes



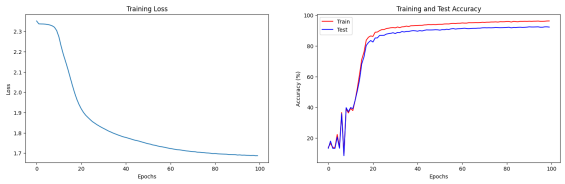
(a) Batch size 16, Training acc est 96.28%, Test acc est 92.37%



(b) Batch size 32, Training acc est 96.22%, Test acc est 92.42%



(c) Batch size 64, Training acc est 96.42%, Test acc est 92.87%



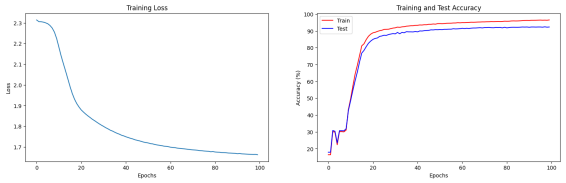
(d) Batch size 128, Training acc est 96.26%, Test acc est 92.27%

Figure 13: Impact de la taille des batch

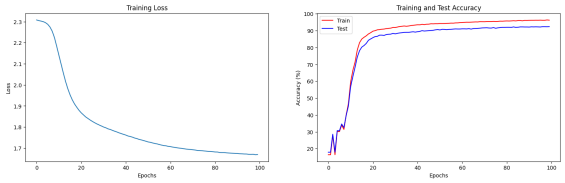
Bien que les différentes tailles de batch aient montré des performances d'entraînement et de test similaires, la taille de batch de 64 a légèrement surpassé les autres en termes de

précision de test. Ainsi, pour ce modèle spécifique, une taille de batch de 64 semble offrir une meilleure généralisation, bien que les variations de performance entre les différentes tailles de batch soient relativement minimes.

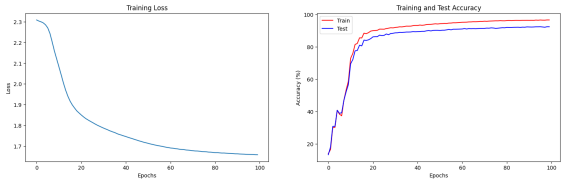
4.1.3 Évaluation des canaux de sortie



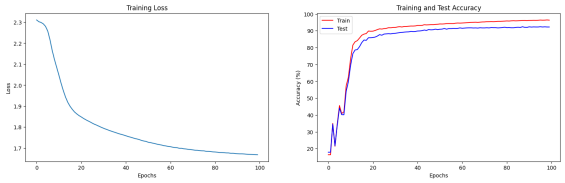
(a) channel out 16, Training acc est 96.28%, Test acc est 92.32%



(b) channel out 32, Training acc est 96.37%, Test acc est 92.87%



(c) channel out 64, Training acc est 96.62%, Test acc est 92.42%



(d) channel out 128, Training acc est 96.29%, Test acc est 92.22%

Figure 14: Impact des canaux de sortie

En conclusion, les expérimentations sur la taille des batchs et les canaux de sortie ont révélé que, bien que les variations de performance soient généralement faibles, une taille de batch de 64 et 32 canaux de sortie ont permis d'obtenir les meilleures précisions de test, respectivement 92.67% et 92.87. Ces configurations semblent favoriser une meilleure généralisation du modèle, indiquant que des ajustements optimaux dans ces paramètres peuvent améliorer les performances globales..

4.1.4 Différents réseaux

Réseau 2: `Conv1D(3,1,128,1)` → `MaxPool1D(2,2)` → `Conv1D(3,128,32,1)`→ `MaxPool1D(2,2)` →`Flatten()` → `Linear(1984,100)` → `ReLU()` → `Lineaire(100,10)`

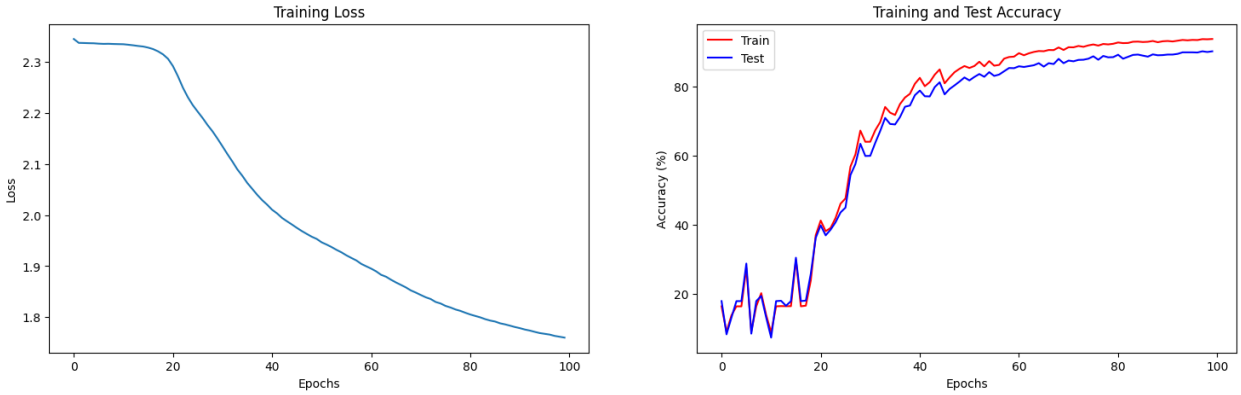


Figure 15: Réseau 2: Training acc est 93.74%, Test acc est 90.18%

Réseau 3: `Conv1D(3,1,128,1)` → `MaxPool1D(2,2)` → `Conv1D(3,128,64,1)`→`MaxPool1D(2,2)` →`Conv1D(3,64,32,1)`→ `MaxPool1D(2,2)`→`Flatten()` → `Linear(960,100)` → `ReLU()` → `Lineaire(100,10)`

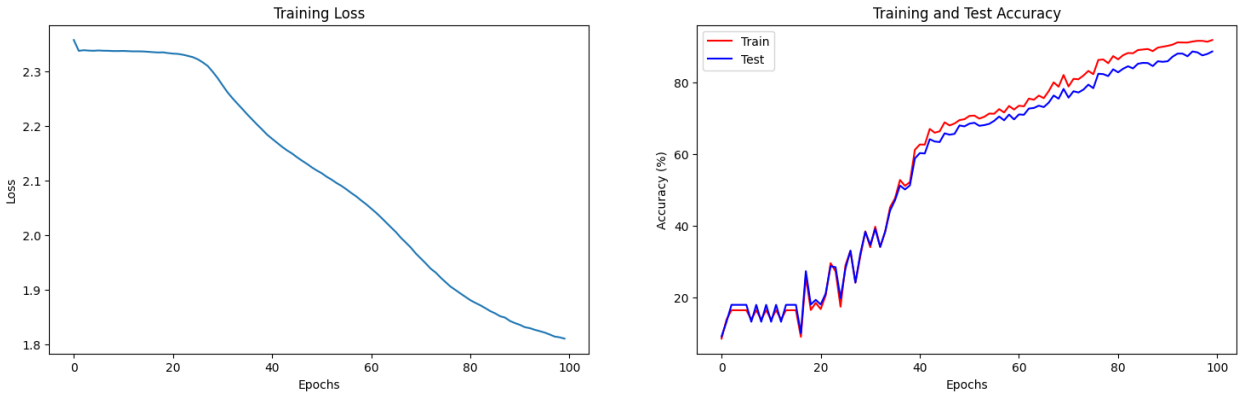
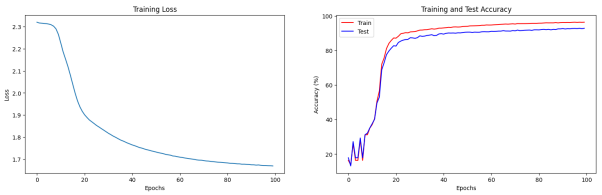


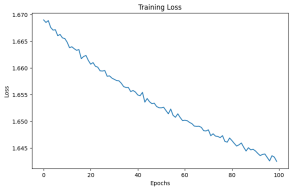
Figure 16: Réseau 3: Training acc est 94.87%, Test acc est 91.22%

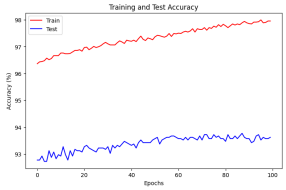
Nous avons observé qu'au bout de 100 epochs, les précisions des réseaux 2 et 3 n'étaient pas aussi élevées que celle du réseau 1, bien que la différence ne soit pas très grande. Le réseau 3 n'avait pas encore convergé après 100 epochs, mais il était proche de son niveau de convergence. Comme chaque réseau avait une couche supplémentaire de conv1D et de maxpool1D par rapport à sa structure précédente, cela a résulté en un temps d'entraînement de 4 heures pour le réseau 3 et de 2 heures pour le réseau 2. En comparant le temps et l'efficacité, le réseau 1 s'est avéré être un peu plus optimal.

4.1.5 500 epochs pour le réseau 1 avec batch size 64 et 32 canaux de sorties

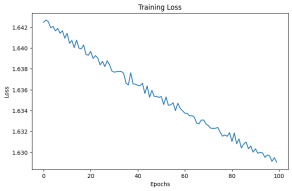


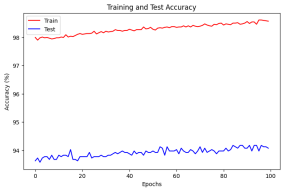
(a) [0 - 100] epochs



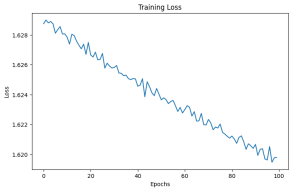


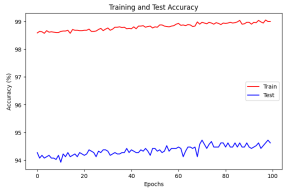
(b) [100 - 200] epochs



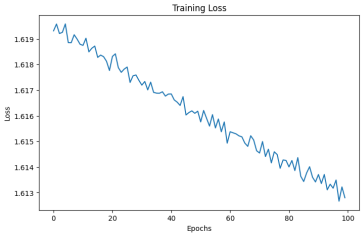


(c) [200 - 300] epochs





(d) [300 - 400] epochs



(e) [400 - 500] epochs

Figure 17: 500 epochs: Training acc est 99.27%, Test acc est 94.71%

En conclusion, en augmentant le nombre d'epochs à 500 pour le réseau 1, avec une taille de batch de 64 et 32 canaux de sortie, nous avons observé une amélioration progressive des performances. Les résultats montrent une précision d'entraînement de 99.27% et une précision de test de 94.71% après 500 epochs, indiquant une meilleure généralisation et une performance globale accrue du modèle par rapport aux phases d'entraînement précédentes.

4.2 Convolution 2D

La convolution 2D est principalement utilisée pour traiter des données d'image. Elle fonctionne en appliquant un filtre dans l'espace bidimensionnel pour capter les motifs spatiaux dans les images, tels que les bords, les coins, etc. Chaque filtre travaille sur une région locale de l'image d'entrée, extrayant des caractéristiques utiles qui seront ensuite utilisées dans des tâches telles que la classification d'images, la détection d'objets, et plus encore.

Pour Class Conv2D, la méthode est similaire à celle de Conv1D, sauf que nous remplaçons le paramètre k_size par v_size et h_size, et le $stride$ par v_stride et h_stride. La logique globale est identique. Concernant Class AvgPool2D, de la même manière, nous remplaçons les paramètres de taille de fenêtre et de pas (stride). La principale différence avec Maxpool1D est que nous ne sélectionnons pas la valeur maximale des éléments dans la fenêtre de pooling pour l'output, mais plutôt la moyenne des éléments. Ainsi, nous devonsnremplacer toutes les opérations qui prennent la valeur maximale par des opérations qui calculent la valeur moyenne.

4.2.1 Résultats et comparaisons de différents réseaux

De la même façon, nous utiliserons la fonction de perte LogSoftMaxCrossEntropy et les entraînons d'abord 100 epochs.

Réseau 1: `Conv2D(3,3,1,32,1,1)` → `AvgPool2D(2,2,2,2)` → `Flatten2D()`→ `Linear(1568,100)` → `ReLU()`→`Linear(100,10)`

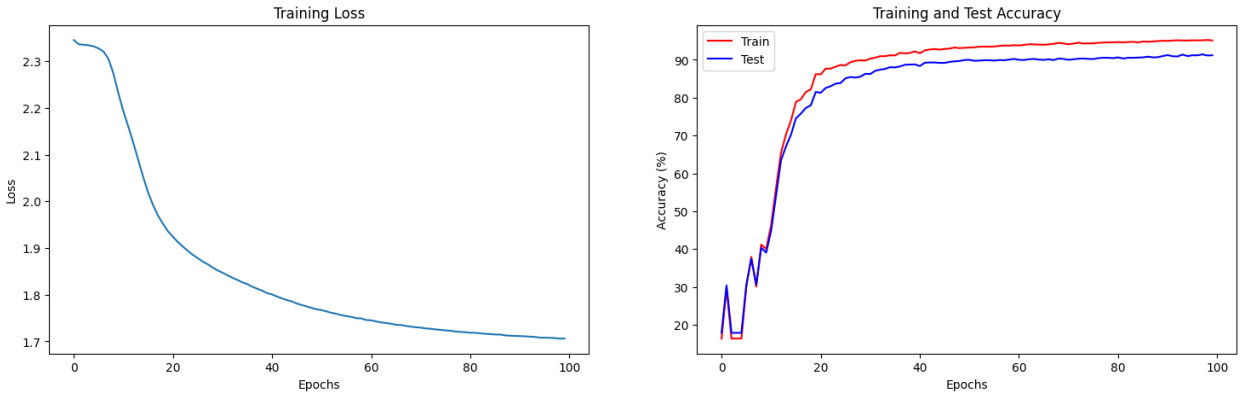


Figure 18: Réseau 1: Training acc est 95.25%, Test acc est 91.62%

Réseau 2: `Conv2D(3,3,1,64,1,1)` → `AvgPool2D(2,2,2,2)` → `Flatten2D()`→ `Linear(3136,100)` → `ReLU()`→`Linear(100,10)`

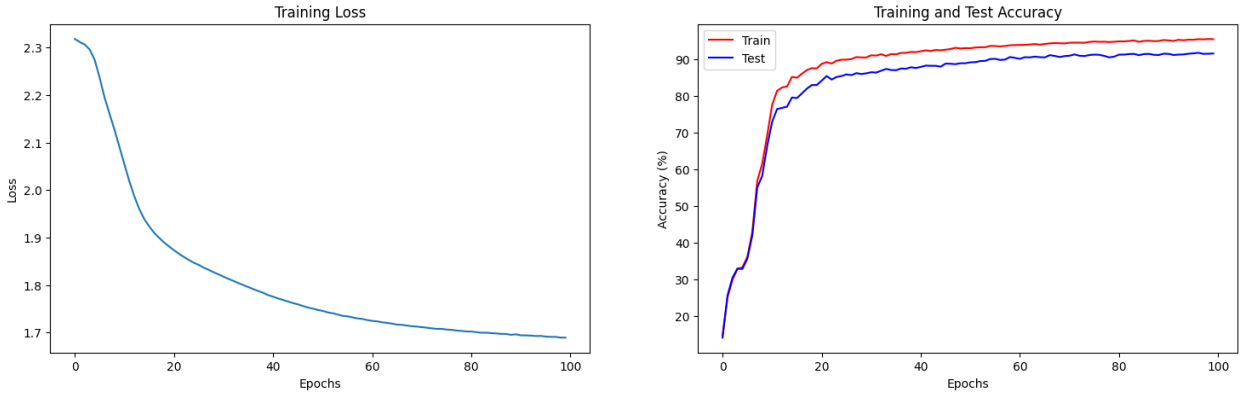


Figure 19: Réseau 2: Training acc est 95.64%, Test acc est 91.48%

Réseau 3: `Conv2D(3,3,1,32,1,1)` → `AvgPool2D(2,2,1,1)` → `Conv2D(3,3,32,64,1,1)` →`AvgPool2D(2,2,2,2)`→`Flatten2D()`→ `Linear(1600,100)`→ `ReLU()`→`Linear(100,10)`

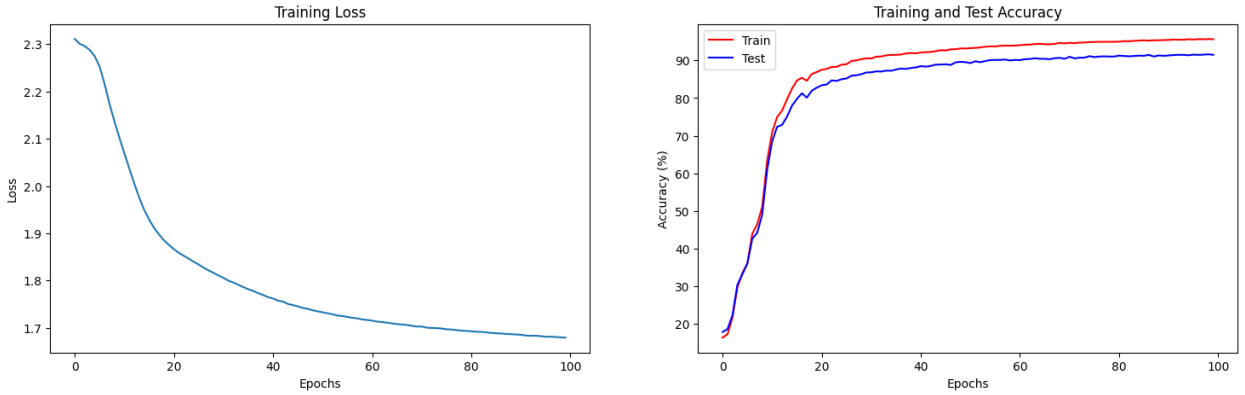
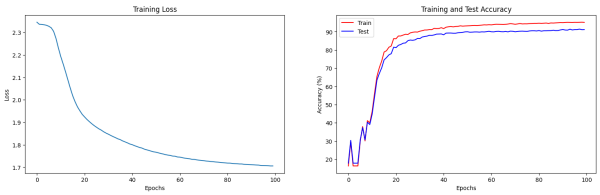


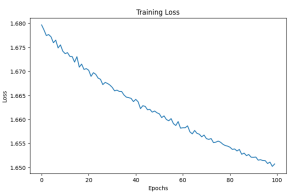
Figure 20: Réseau 3: Training acc est 95.39%, Test acc est 91.47%

La comparaison des performances des trois réseaux montre que bien que le Réseau 2 ait une précision d'entraînement légèrement supérieure (95.64% contre 95.39%) par rapport au Réseau 3, le Réseau 3 présente une précision de test légèrement inférieure (91.47% contre 91.48%). Cela suggère que malgré une architecture plus complexe avec deux couches de convolution, le Réseau 3 ne parvient pas à améliorer significativement les performances par rapport au Réseau 2. En revanche, le Réseau 1, malgré sa simplicité avec une seule couche de convolution, présente une meilleure précision de test (91.62%), indiquant une meilleure capacité de généralisation. Ainsi, ces résultats suggèrent que la complexité accrue du modèle n'est pas toujours synonyme d'amélioration des performances de généralisation.

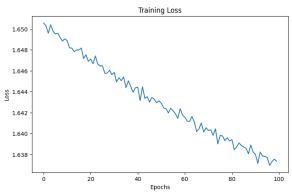
4.2.2 500 epochs pour le réseau 2 avec 64 canaux de sorties

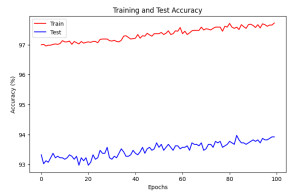


(a) [0 - 100] epochs

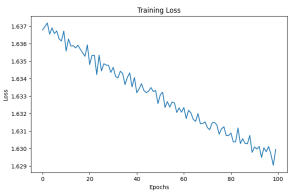


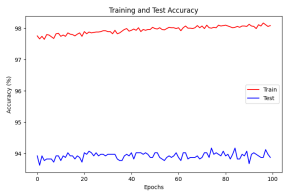
(b) [100 - 200] epochs



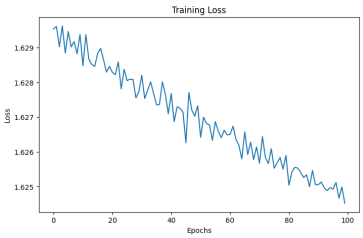


(c) [200 - 300] epochs





(d) [300 - 400] epochs



(e) [400 - 500] epochs

Figure 21: 500 epochs: Training acc est 98.39%, Test acc est 94.12%

L'extension de l'entraînement du réseau 2 sur 500 epochs a entraîné une nette amélioration des performances. Alors que les premières 100 epochs ont produit des résultats similaires à ceux initiaux, avec une précision d'entraînement de 95.25% et une précision de test de 91.62%, les epochs suivantes ont montré une tendance significative à l'amélioration. À la fin de l'entraînement, le réseau a atteint une précision d'entraînement remarquable de 98.39% et une précision de test de 94.12%. Cela indique que l'entraînement prolongé a permis au modèle de mieux capturer les motifs dans les données et de généraliser plus efficacement, comme en témoigne l'augmentation de la précision de test. Ces résultats soulignent l'importance de l'entraînement sur une période prolongée pour améliorer les performances des réseaux neuronaux.

4.2.3 Comparaions Conv1D et Conv2D

Les résultats des tests de Conv1 et Conv2 montrent des différences notables. Conv1, utilisant une convolution 1D, a démontré une meilleure performance avec une précision de test de 94.71% après 500 epochs et une configuration optimale de 32 canaux de sortie et une taille de batch de 64. En revanche, Conv2, basé sur la convolution 2D, a atteint une précision de test légèrement inférieure de 94.12% dans une configuration similaire. Bien que les architectures plus complexes n'aient pas nécessairement amélioré les performances, l'entraînement prolongé a permis à Conv2 d'améliorer sa capacité de généralisation, ce qui souligne l'importance d'un entraînement prolongé pour les deux types de convolution. Il est possible que les meilleures performances de Conv1 par rapport à Conv2 soient dues à un nombre insuffisant de tests pour trouver une configuration optimale pour Conv2, indiquant que des tests supplémentaires pourraient améliorer les résultats de Conv2.

5 Conclusion

Ce projet s'inscrit dans la dynamique actuelle du Machine Learning, visant à explorer les fondements des réseaux de neurones et à mettre en œuvre une approche modulaire inspirée des premières versions de PyTorch et des pratiques analogues. L'objectif principal était de créer une infrastructure flexible et générique pour la construction et l'entraînement de réseaux de neurones, couvrant un large éventail de tâches allant de la régression à la classification, en passant par la compression de données.

Nous avons implémenté divers modules de base, y compris les couches linéaires et les fonctions d'activation, ainsi que des architectures plus avancées comme les auto-encodeurs. Les tests sur les jeux de données USPS et MNIST ont montré que l'utilisation de la fonction de perte Binary Cross Entropy (BCE) produisait des reconstructions de meilleure qualité par rapport à Mean Squared Error (MSE). De plus, les auto-encodeurs ont démontré une capacité efficace de débruitage, particulièrement avec des réseaux plus complexes.

Dans les tests de convolution, Conv1 utilisant une convolution 1D a montré une précision de test supérieure après une optimisation adéquate de ses paramètres. Conv2, basé sur la convolution 2D, a également montré de bonnes performances, mais a légèrement moins bien performé, ce qui peut être attribué à un nombre insuffisant de tests pour optimiser ses configurations.

En termes d'axes d'amélioration, nous aurions pu explorer des configurations plus variées et des architectures plus complexes pour Conv2 afin d'optimiser davantage ses performances.

De plus, l'utilisation de techniques de régularisation avancées comme le dropout ou le batch normalization aurait pu améliorer la capacité de généralisation des modèles. Une évaluation sur des jeux de données plus diversifiés et la mise en œuvre de méthodes d'entraînement plus sophistiquées, comme le learning rate scheduling ou l'optimisation adaptative, auraient également été bénéfiques.

Ce projet nous a apporté une compréhension approfondie de la construction et de l'optimisation de réseaux de neurones modulaires. Il a mis en évidence l'importance de l'expérimentation et de l'ajustement des hyperparamètres dans le développement de modèles performants. En outre, il nous a permis d'acquérir des compétences pratiques en implémentation de diverses architectures et fonctions de perte, ainsi qu'une appréciation des défis et des subtilités associés à l'entraînement de modèles de Machine Learning.