Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique - parcours DAC

ML – Machine Learning

Rapport de Projet

Réseau de neurones : DIY

Réalisé par :

Raphaël Renard
Luc Salvon

Supervisé par :

Nicolas Baskiotis
Nicolas Thome

Mai 2024

CONTENTS

CHAPTER 1

INTRODUCTION

Ce projet a pour objectif d'implémenter des réseaux de neurones, inspirés des anciennes versions de PyTorch en Lua. Notre approche vise à créer une architecture modulaire où chaque couche du réseau est considérée comme un module distinct. Cette modularité permet de construire des réseaux neuronaux flexibles et réutilisables, facilitant ainsi l'expérimentation et l'optimisation.

Le projet consiste à développer une série de modules, y compris des couches linéaires et convolutives, des fonctions d'activation, et des fonctions de perte. L'implémentation repose sur le calcul des gradients par rétropropagation, en utilisant la dérivation en chaîne, pour ajuster les poids et améliorer la performance du réseau.

En plus des modules de base, nous avons également implémenté des encapsulages pour simplifier l'architecture du réseau et des modules pour des tâches spécifiques telles que la classification multi-classe et l'auto-encodage. Chaque étape du projet est testée sur des données réelles, permettant ainsi d'évaluer et d'améliorer les performances de notre réseau neuronal.

CHAPTER 2

IMPLÉMENTATION

Cette partie renseigne l'architecture de notre implémentation, en détaillant l'utilité de chacune de ses composantes.

2.1 Modules

Un module est une composante atomique du réseau de neurones. Il est défini par la classe abstraite `Module`, qui demande l'implémentation des méthodes suivantes :

- `zero_grad` – réinitialise le gradient des paramètres du module
- `forward(X)` – applique la transformation effectuée par le module aux données X
- `update_parameters(gradient_step)` – Met à jour les paramètres du module selon le gradient, et le `gradient_step` (qui permet de réaliser des corrections plus ou moins importantes. Par défaut, `gradient_step` $= 10^{-3}$)
- `backward_update_gradient(input, delta)` – Met à jour la valeur du gradient du coût par rapport aux paramètres du module. `input` représente les entrées du module, `delta` le gradient du coût par rapport aux entrées du module suivant.
- `backward_delta(input, delta)` – Calcule le gradient du coût par rapport aux entrées du module. `input` et `delta` sont les mêmes que la méthode précédente.

Ces modules peuvent être séparés en deux catégories principales, les couches et les fonctions d'activation.

2.1.1 Couches

Nos couches sont des modules qui permettent une transformation des données en fonction de paramètres. Nous avons implémenté 4 couches différentes :

- `Linear` – Une couche linéaire, effectue le produit scalaire des entrées avec les paramètres du module, ainsi qu'un biais
- `Flatten` – Une couche qui transforme des images avec plusieurs canaux, donc en plusieurs dimensions, en vecteurs 1D
- `MaxPool1D` – Une couche de pooling qui permet de réduire la taille des données d'entrée en effectuant un sous-échantillonage et calculant le maximum à chaque fois
- `Conv1D` – Une couche convolutionnelle conçue pour extraire des caractéristiques locales des données d'entrée, particulièrement utile pour les données structurées en grille comme les images

2.1.2 Fonctions d'activation

Les fonctions d'activation non linéaires sont essentielles pour introduire des non-linéarités dans le réseau. Nous avons implémenté plusieurs fonctions d'activation :

- `TanH` – Applique la fonction tangente hyperbolique aux entrées. les sorties sont comprises entre -1 et 1.
- `Sigmoide` – Une fonction sigmoïde qui transforme les entrées en une plage comprise entre 0 et 1.
- `ReLU` – Une fonction qui renvoie la valeur d'entrée si elle est positive, sinon zéro.
- `Softmax` – Une fonction qui amplifie les différences au sein des entrées. Une entrée proche du maximum aura une valeur proche de 1, une valeur proche du minimum aura une valeur proche de 0.

2.2 Fonctions de perte

Les fonctions de perte (loss) quantifient l'écart entre les prédictions du modèle et les véritables étiquettes. Elles sont cruciales pour l'entraînement supervisé du modèle.

Les fonctions de loss implémentées héritent de la classe abstraite `Loss`, qui nécessite l'implémentation de deux méthodes :

- `forward(y, yhat)` – Evalue l'erreur entre la valeur attendue y et la prédiction du modèle $\hat{y}$.
- `backward(y, yhat)` – Calcule le gradient du coût par rapport à $\hat{y}$.

Nous avons implémenté les fonctions de perte suivantes :

- `MSELoss` (Mean Squared Error) – Utilisée pour les tâches de régression.
- `BCELoss` (Binary Cross Entropy) – Utilisée pour les tâches de classification binaire.
- `CrossEntropyLoss` – Utilisée pour les tâches de classification multi-classe.
- `LogSoftmaxCrossEntropy` – Une combinaison de la fonction log-softmax et de la perte par entropie croisée, utilisée pour la classification multi-classe. Cette fonction est particulièrement efficace car elle stabilise les calculs en appliquant log-softmax avant de calculer l'entropie croisée, ce qui peut être bénéfique pour les modèles avec des sorties à valeurs élevées.

2.3 Modèles

Pour simplifier la construction et l'entraînement des réseaux de neurones, nous avons créé des encapsulages qui facilitent l'assemblage des différents modules de base.

2.3.1 Sequentiel

Un modèle séquentiel est un enchaînement de modules. Il est défini dans la classe `Sequentiel`, qui contient un paramètre `modules`, la liste de modules du modèle. Cette classe comprend deux méthodes :

- `forward(X)` – Inférence du modèle, applique successivement les méthodes `forward` de chaque module à partir de l'entrée X jusqu'à obtenir la sortie du modèle.
- `backward(delta)` – Calcule le gradient de chaque paramètre du modèle, `delta` représente ici le gradient de la loss par rapport aux sorties du modèle.

2.3.2 AutoEncoder

L'auto-encodeur est un type de réseau de neurones utilisé pour l'apprentissage non supervisé des représentations efficaces des données. Nous l'avons défini dans la classe `AutoEncoder`.

Il comprend un encodeur qui calcule une représentation dans un espace latent des données et un décodeur qui reconstruit les données d'entrée originales à partir de l'espace latent.

L'encodeur et le décodeur sont tous les deux des réseaux représentés avec la classe `Sequentiel`. Cette classe contient trois méthodes :

- `forward` – Applique le `forward` du décodeur au `forward` de l'encodeur
- `backward` – Calcule le `backward` de l'encodeur à partir du `backward` du décodeur
- `latent` – Calcule la représentation dans l'espace latent des données, c'est-à-dire applique le `forward` de l'encodeur uniquement

2.4 Autres fonctions

2.4.1 Optim

`Optim` est une classe implémentant un optimiseur, qui a pour rôle de modifier les paramètres d'un modèle pour chaque batch donné, afin d'optimiser son erreur.

Cette classe contient trois paramètres :

- `net` – Le modèle à optimiser (par exemple un modèle séquentiel)
- `loss` – La fonction de loss à utiliser pour optimiser le modèle.
- `eps` – Le pas du gradient, un pas plus élevé effectuera des modifications des poids plus importantes à chaque étape.

L'optimiseur contient une seule méthode `step(batch_X, batch_Y)`, qui calcule la prédiction du modèle puis en déduit son erreur. Il calcule le gradient de chaque paramètre d'après cette erreur, et met à jour ces paramètres en fonction du gradient.

2.4.2 sgd

La fonction `sgd` permet de réaliser une descente de gradient afin d'entraîner un modèle. Elle comporte les paramètres suivants :

- `net` – Le modèle à entraîner
- `data` – Un tuple (X, Y) comportant les données d'entraînement du modèle
- `loss` – La fonction de loss à utiliser
- `batch_size` – Le nombre d'exemples à utiliser à chaque étape de la descente de gradient
- `nb_epochs` – Le nombre d'epochs (= utilisation de tous les exemples pour l'entraînement) maximum pour entraîner le modèle
- `eps` – La différence d'erreur entre deux étapes minimale, en dessous de laquelle on considère le modèle comme entraîné
- `step` – Le pas du gradient

2.4.3 grid_search

La fonction `grid_search` permet d'explorer différentes configurations d'hyperparamètres pour un modèle et des données, et identifier ceux donnant les meilleurs résultats via une cross-validation. Elle comporte les paramètres suivants :

- `net` – Le réseau dont on veut identifier les meilleurs hyperparamètres
- `param_grid` – Un dictionnaire de la forme `{<paramètre> : [<valeur1>, <valeur2>, ...]}` contenant les hyperparamètres à tester
- `X` et `y` – Les données sur lesquelles entraîner les modèles
- `metric` – La métrique à utiliser lors de la cross-validation
- `cv` (optionnel) – Le nombre de folds de la cross-validation, par défaut 3
- `save_loss_graphs` (optionnel) – Booléen déterminant si l'on veut enregistrer les courbes de loss pour chaque combinaison d'hyperparamètres. Faux par défaut

CHAPTER 3

EXPÉRIMENTATIONS

Grâce à des expérimentations approfondies, nous explorons diverses tâches, telles que la reconstruction d'images et l'entraînement de réseaux convolutionnels. Ces expérimentations nous permettent d'analyser l'impact des choix architecturaux et des hyperparamètres sur les performances des modèles.

Pour faire ces expérimentations, nous avons utilisé des données pour la reconnaissance de chiffres (MNIST).

3.1 Architecture

3.1.1 Couches linéaires vs convolution

On compare tout d'abord un réseau avec uniquement des couches linéaires et un réseau avec une convolution sur la classification de chiffres.

Onn utilise le réseau linéaire suivant :

$$Linear(64, 992) \to ReLU() \to Linear(992, 100) \to ReLU() \to Linear(100, 10)$$

qui nous donne la figure 3.1a.

Pour le réseau convolutionnel on prend l'architecture suivante :

$$Conv1D(3, 1, 32) \to MaxPool1D(2, 2) \to Flatten() \to Linear(992, 100) \to ReLU() \to Linear(100, 10)$$

qui donne la figure 3.1b. La loss utilisée dans les deux cas est `LogSoftmaxCrossEntropy()`.

Au bout de 200 epochs, on a dans les deux cas une accuracy de 0.99 sur le jeu de test. En fin de compte, les deux réseaux ont donné des performances similiaires, mais on remarque que la loss de la convolution a convergé plus vite.

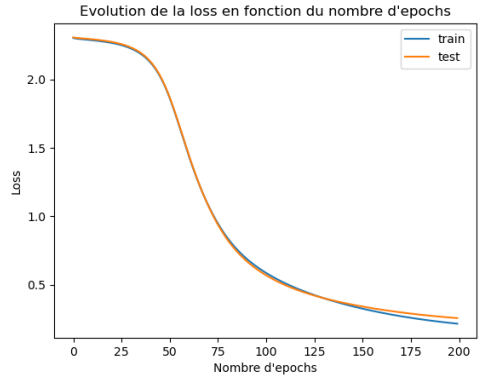
3.1.2 Fonctions d'activation

Pour tester les différentes fonctions d'activation, nous utilisons un réseau simple : $Linear(64, 100) \to ReLU() \to Linear(100, 10)$ avec une `LogSoftmaxCrossEntropy()` pour faire de la reconnaissance de chiffres manuscrits.

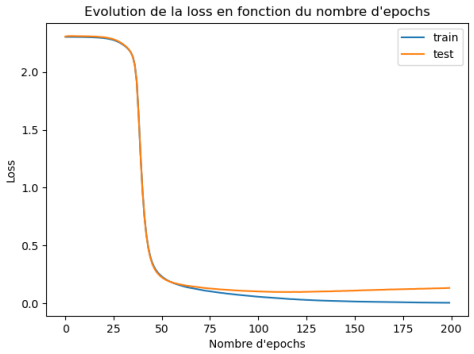
Pour l'activation `Sigmoide` nous avons utilisé un learning rate de 2×10^{-5}, pour `ReLU` 10^{-2} et pour `Softmax` 10^{-4}. La figure 3.2 montre les courbes d'apprentissage. On voit que la fonction softmax est la moins efficace, suivie de la sigmoide.

3.1.3 Nombre de couches

Le nombre de couches d'un réseau neuronal est très important. Il faut bien le choisir en fonction de la tâche que l'on veut effectuer. La capacité de représentation du réseau en dépend. Mais augmenter le nombre de couches augmente aussi le temps d'entraînement.

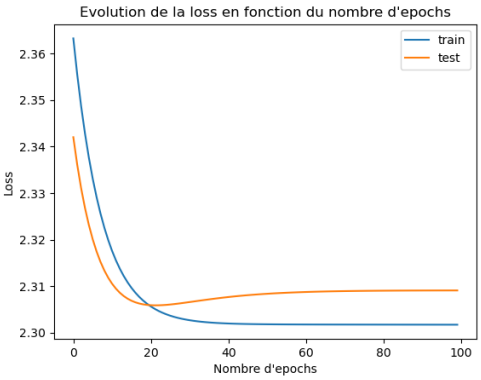


(a) Réseau avec seulement des couches linéaires

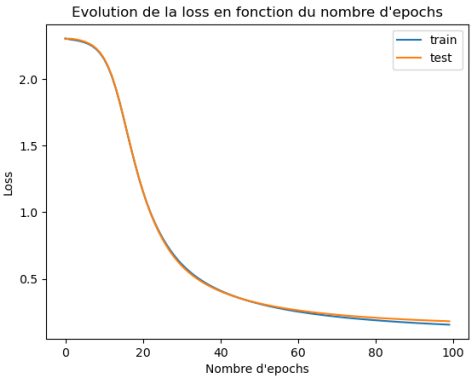


(b) Réseau avec une convolution

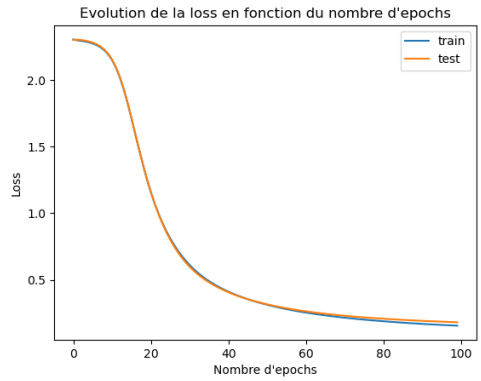
Figure 3.1: Courbes de loss pendant l'entraînement de réseaux pour de la reconnaissance de chiffres



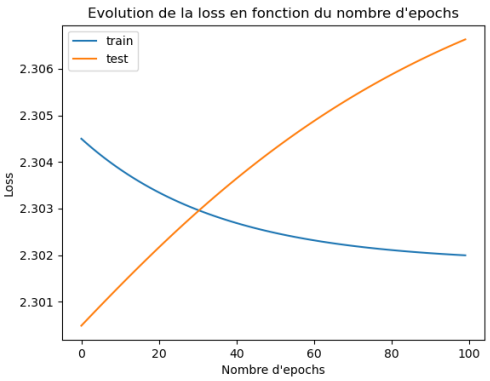
(a) Sigmoide



(b) ReLU

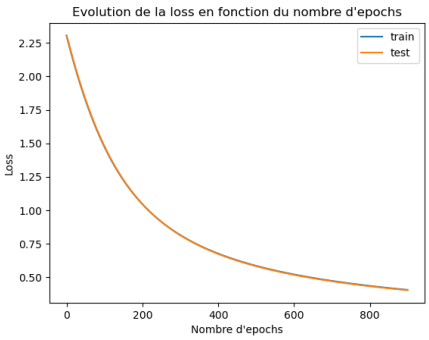


(c) TanH

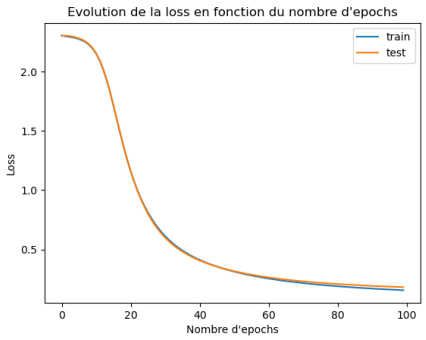


(d) Softmax

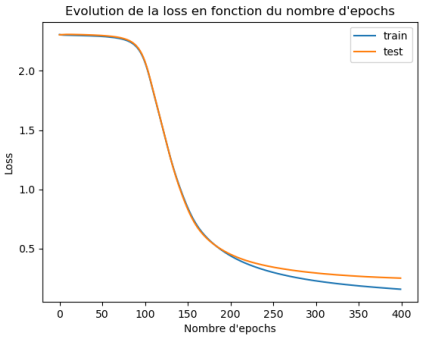
Figure 3.2: Courbes de loss pendant l'entraînement pour différentes fonctions d'activation



(a) Avec une couche



(b) Avec 2 couches



(c) Avec 3 couches

Figure 3.3: Courbes de loss pendant l'entraînement pour différents nombres de couches

Pour tester cela nous avons utiliser trois réseaux, toujours sur la reconnaissance de chiffres :

$$Linear(64, 10)$$

$$Linear(64, 100) \rightarrow ReLU() \rightarrow Linear(100, 10)$$

$$Linear(64, 100) \rightarrow ReLU() \rightarrow Linear(100, 10) \rightarrow ReLU() \rightarrow Linear(100, 200) \rightarrow ReLU() \rightarrow Linear(200, 10)$$

On peut voir sur la figure 3.3 que les réseaux avec plus de couches ont une meilleure loss.

3.1.4 Taille des couches

Similairement, la taille des couches est aussi importante. Nous avons testé trois tailles de couches cachées pour en comprendre l'effet :

$$Linear(64, 10) \rightarrow ReLU() \rightarrow Linear(10, 10)$$

$$Linear(64, 100) \rightarrow ReLU() \rightarrow Linear(100, 10)$$

$$Linear(64, 1000) \rightarrow ReLU() \rightarrow Linear(1000, 10).$$

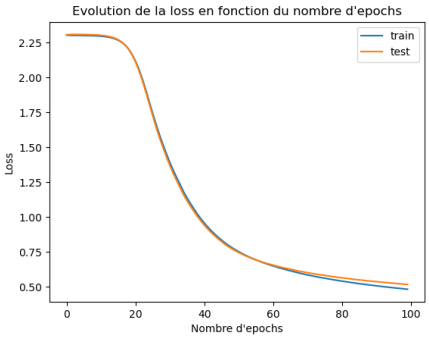
Nous pouvons voir sur la figure 3.4 qu'avec une couche cachée de taille 100, on obtient une meilleure loss car le réseau a de meilleures capacités de représentation. Cependant avec une taille de 1000 le réseau fait du sur-apprentissage.

3.2 Hyperparamètres

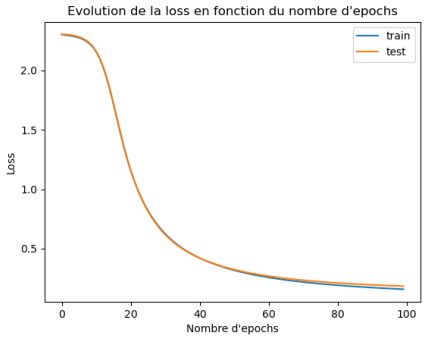
3.2.1 Effet de la loss

Il est important de choisir une loss adaptée à la tâche d'apprentissage que l'on veut effectuer.

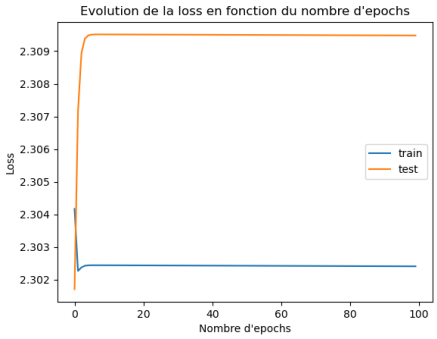
En reprenant un réseau $Linear(64, 100) \rightarrow ReLU() \rightarrow Linear(100, 10)$, on teste différentes fonctions de perte (figure 3.5). On peut voir par exemple que la Binary Cross Entropy n'est pas du tout adaptée au problème.



(a) Avec une couche cachée de taille 10

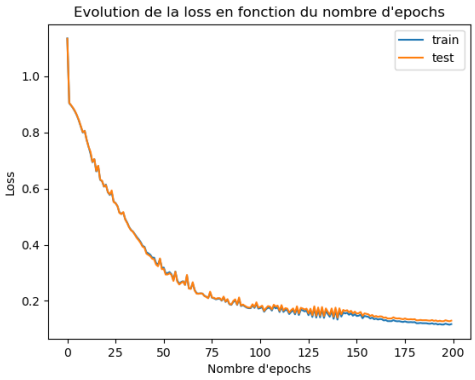


(b) Avec une couche cachée de taille 100

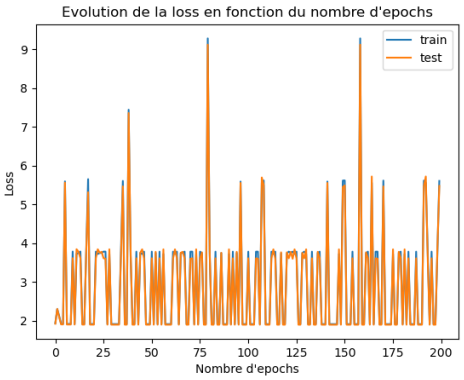


(c) Avec une couche cachée de taille 1000

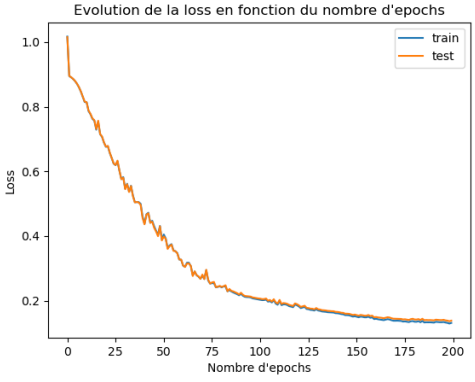
Figure 3.4: Courbes de loss pendant l'entraînement pour différentes taille de couche cachée



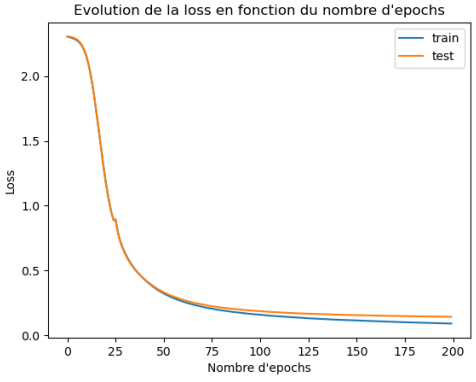
(a) MSE



(b) BCE



(c) Cross Entropy



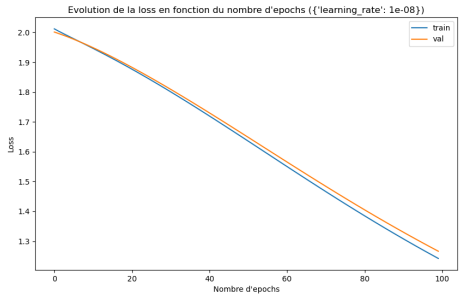
(d) LogSoftmax Cross Entropy

Figure 3.5: Courbes de loss pendant l'entraînement pour différentes fonctions de perte

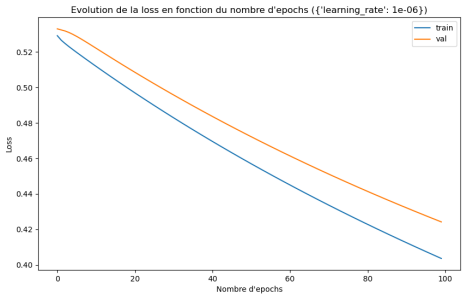
3.2.2 Taux d'apprentissage (learning rate)

Le taux d'apprentissage est un hyperparamètre très important lorsqu'on entraîne des réseaux de neurones. Il définit la taille des pas à chaque mise à jour des poids du réseau durant l'apprentissage. Un taux d'apprentissage bien ajusté permet des mises à jour de poids suffisamment importantes pour converger rapidement vers un minimum, tout en prévenant les oscillations et les divergences. L'objectif est de trouver un équilibre optimal entre la vitesse de convergence et la stabilité des résultats.

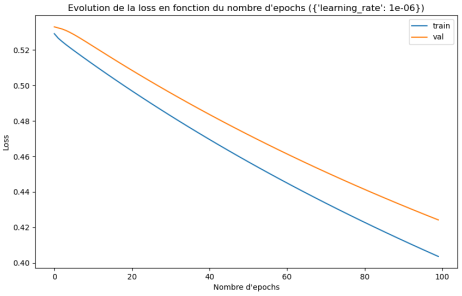
Pour tester cet hyperparamètre, nous utilisons les données MNIST et essayons de faire de la reconnaissance de chiffre avec un réseau simple $Linear(64, 100) \rightarrow ReLU() \rightarrow Linear(100, 10)$. On peut voir sur la figure 3.6 qu'un learning rate trop faible ralentit la convergence et devra donc être compensé par un nombre plus élevé d'itérations. D'un autre côté, un learning rate trop élevé donne une moins bonne loss de test dans notre cas car les poids se mettent à jour trop brutalement entraînant une convergence vers un modèle très rapidement sous-optimal.



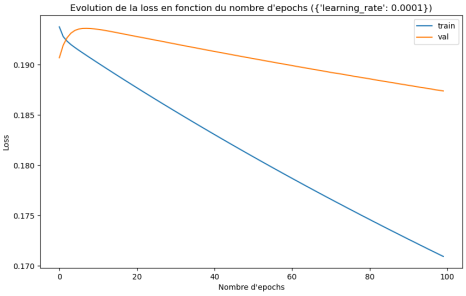
(a) learning rate = 1e-08



(b) learning rate = 1e-06



(c) learning rate = 1e-05



(d) learning rate = 1e-04

Figure 3.6: Courbes de loss pendant l'entraînement pour différents learning rates

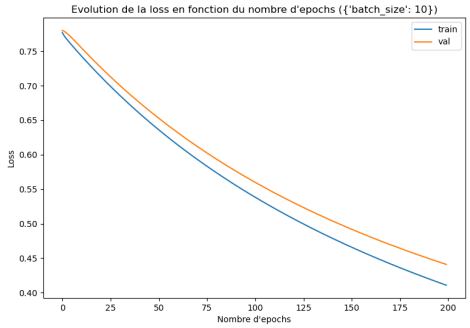
3.2.3 Taille de batch

La taille du batch fait référence au nombre d'exemples de données utilisés dans une itération d'apprentissage pour mettre à jour les poids du modèle.

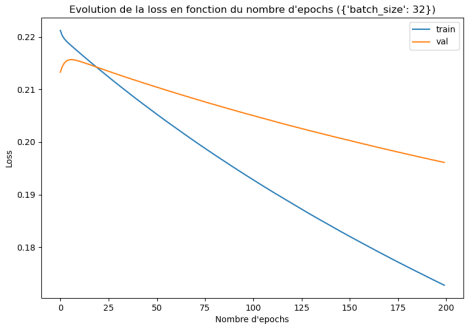
Des batchs plus grands peuvent accélérer le processus d'entraînement car le modèle met à jour ses poids moins fréquemment. Ils peuvent aussi fournir des estimations de gradient plus précises car ils sont calculés sur un plus grand nombre d'exemples de données. Cela peut contribuer à une convergence plus stable et plus rapide du modèle.

Cependant, des batchs plus grands peuvent augmenter le risque de surapprentissage, où le modèle mémorise les données d'entraînement au lieu de généraliser des modèles. Réduire la taille du batch peut aider à introduire plus de variabilité dans le processus d'entraînement et à réduire ce risque.

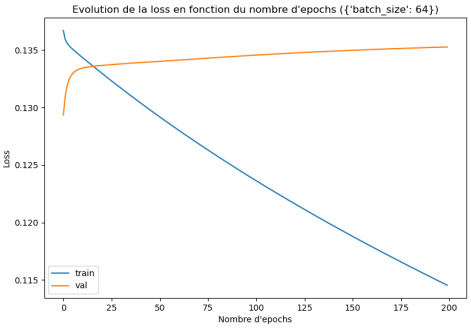
Pour tester cela, on reprend le même réseau qu'avant. On peut observer sur la figure 3.7 que effectivement, plus les batchs sont grands, plus la courbe de loss de test est mauvaise.



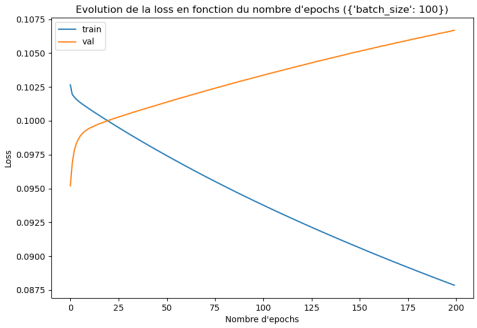
(a) bacth size = 10



(b) bacth size = 32



(c) bacth size = 64



(d) bacth size = 100

Figure 3.7: Courbes de loss pendant l'entraînement pour différentes tailles de batch

3.2.4 Convolution - taille du kernel

Pour les convolutions, il y a des hyperparamètres en plus à prendre en compte. La taille du kernel est l'un d'eux. Cette taille détermine la taille de la fenêtre sur laquelle le filtre est appliqué sur l'image d'entrée.

Nous testerons 3 tailles de noyaux différentes, 1, 5 et 10. Cela nous donne alors les architectures suivantes :

$Conv1D(1, 1, 10) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(320, 10)$,
$Conv1D(5, 1, 10) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(310, 10)$, et
$Conv1D(10, 1, 10) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(310, 10)$.

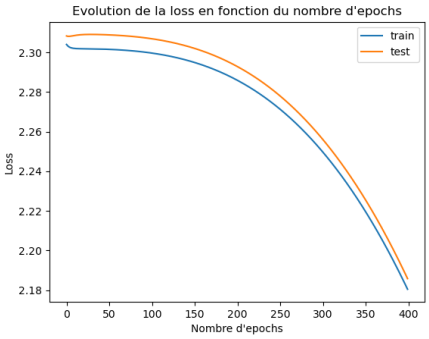
Les résultats sont présentés en figure 3.8.

3.2.5 Convolution - nombre de feature maps

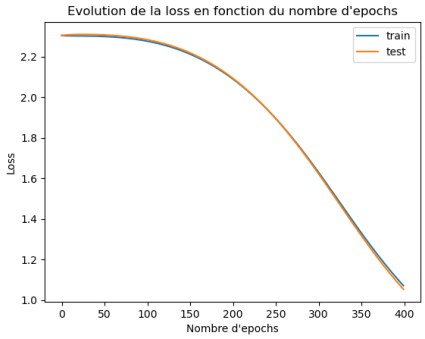
Nous testons aussi 3 valeurs de feature maps différentes, 3, 10 et 32. Cela nous donne alors les architectures suivantes :

$Conv1D(3, 1, 3) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(93, 10)$,
$Conv1D(3, 1, 10) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(310, 10)$, et
$Conv1D(3, 1, 32) \rightarrow MaxPool1D(2, 2) \rightarrow Flatten \rightarrow Linear(992, 10)$

On peut observer sur la figure 3.9 que 3 feature maps nous donne une loss très élevée en test, ce n'est donc pas une option viable. Entre 10 et 32 feature maps, on observe qu'au bout de 400 epochs d'entraînement, 32 nous donne la loss de test la plus faible. Parmi ces trois propositions, nous nous orientons donc vers 32 feature maps.



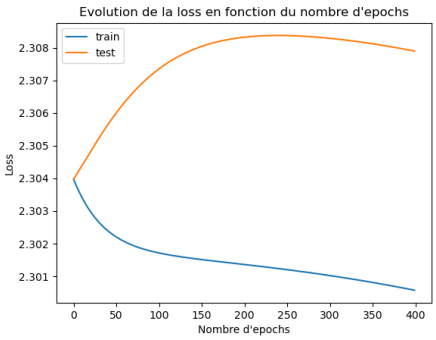
(a) Kernel de taille 1



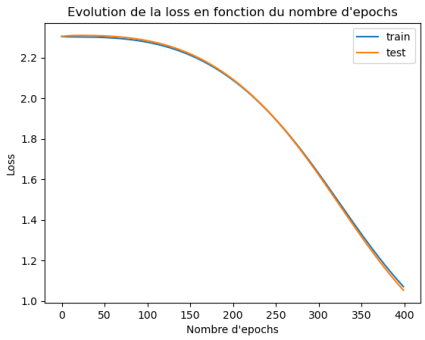
(b) Kernel de taille 5

(c) Kernel de taille 10

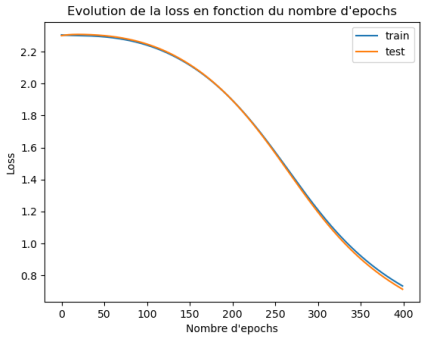
Figure 3.8: Courbes de loss pendant l'entraînement pour différentes tailles du kernel



(a) Avec 3 feature maps



(b) Avec 10 feature maps



(c) Avec 32 feature maps

Figure 3.9: Courbes de loss pendant l'entraînement pour différents nombres de feature maps

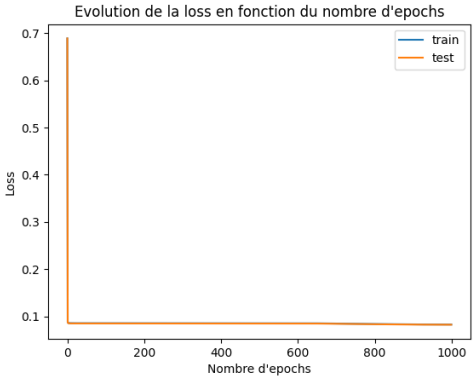
3.3 Auto-encodeur

Notre objectif dans cette partie sera d'entraîner un autoencoder pouvant représenter les images de la base de données MNIST par un vecteur latent. Nous explorerons une approche linéaire et une approche convolutionnelle.

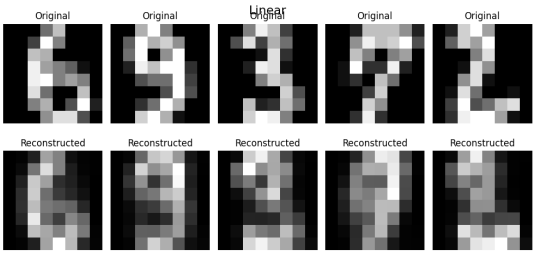
3.3.1 Approche linéaire

Pour l'approche linéaire, nous utiliserons un réseau de base sous la forme :
Encodage : $Linear(64, 200) \rightarrow TanH \rightarrow Linear(200, 50) \rightarrow TanH$
Décodage : $Linear(50, 200) \rightarrow TanH \rightarrow Linear(200, 64) \rightarrow Sigmoide$

Ce réseau nous donne les résultats de la figure 3.10. On peut noter le temps d'apprentissage nécessaire très long (1000 epochs), nous allons allons alors essayer de faire varier le nombre max d'epochs pour voir les résultats.



(a) Courbe de loss



(b) Comparaison des images reconstruites et originelles

Figure 3.10: Résultats de l'entraînement de l'autoencoder linéaire de base

Variation du nombre max d'epochs

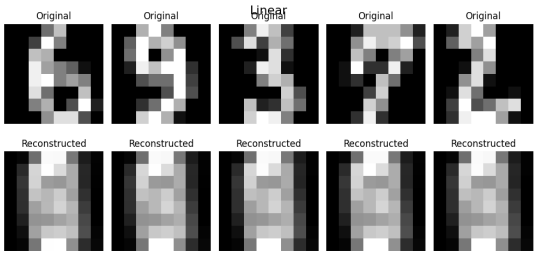
La figure 3.11 compare les images reconstruites en fonction du nombre d'epochs d'entraînement. On observe qu'à 500 epochs les images sont identiques, elles se différencient à partir de 750 epochs et continuent de s'améliorer plus on a d'epochs. 1000 epochs semble être un bon compromis entre qualité et temps de train.

Variation de la taille du vecteur latent

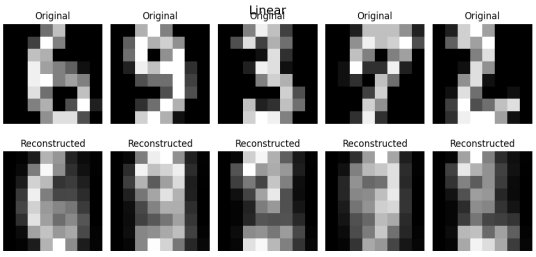
La figure 3.12 montre les différentes images reconstruites obtenues en fonction de la taille du vecteur latent. On observe que pour une taille de 10 et 30 paramètres, tous les chiffres ne sont pas discriminés. Pour 50 et 70 paramètres on obtient des résultats semblables, 50 paraît alors être la meilleure option. Cela revient cependant à réduire une image de 64 valeurs à un vecteur de 50, ce qui n'est pas une importante réduction.

Variation de la couche cachée de l'encodeur

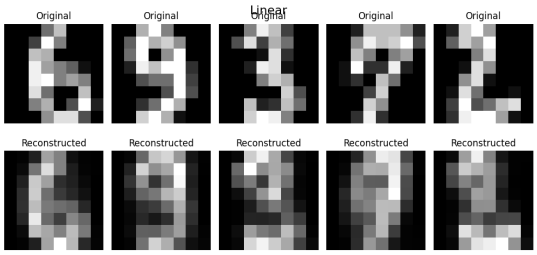
La figure 3.13 montre la différence des images reconstruites selon le nombre de paramètres de la couche cachée (et donc l'expressivité) de l'encodeur. On observe que 50 paramètres n'est pas satisfaisant, mais l'encodeur semble produire un encodage correct pour une couche cachée de 100 paramètres.



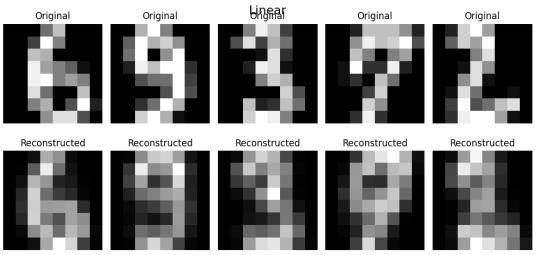
(a) 500 epochs



(b) 750 epochs

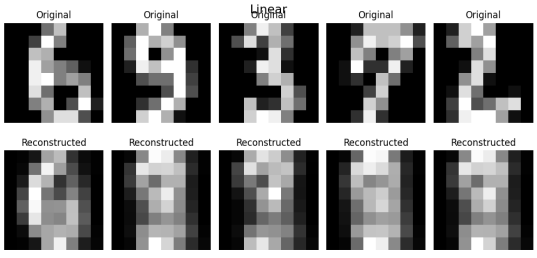


(c) 1000 epochs

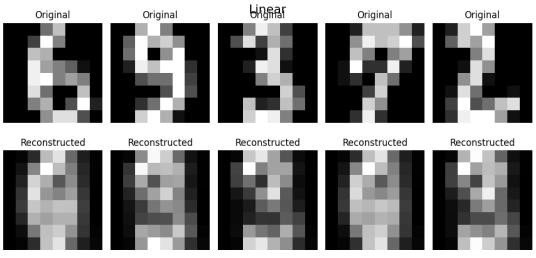


(d) 1500 epochs

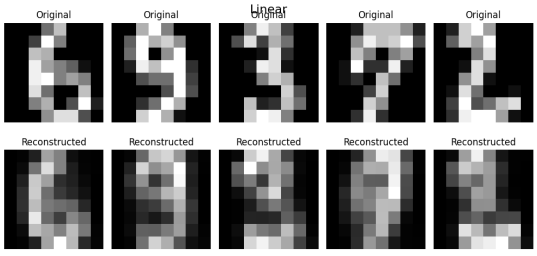
Figure 3.11: Comparaison des images reconstruites pour différents nombres d'epochs de train



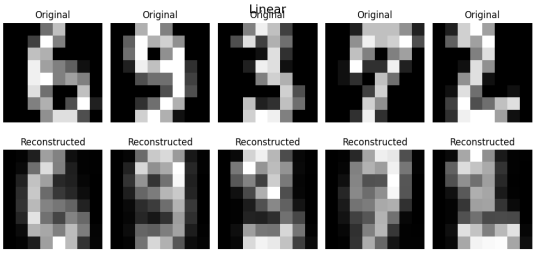
(a) 10 paramètres



(b) 30 paramètres

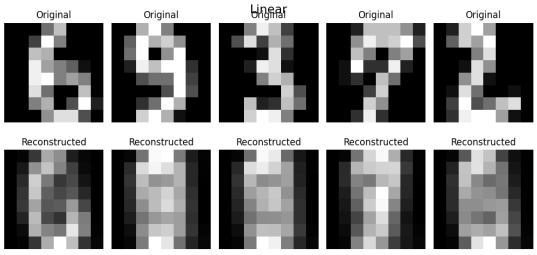


(c) 50 paramètres

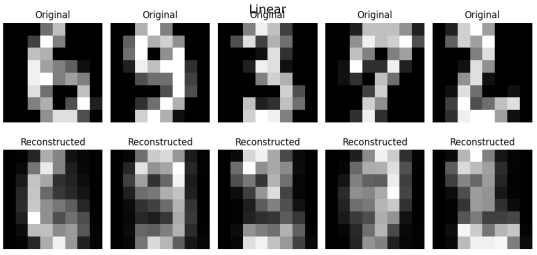


(d) 70 paramètres

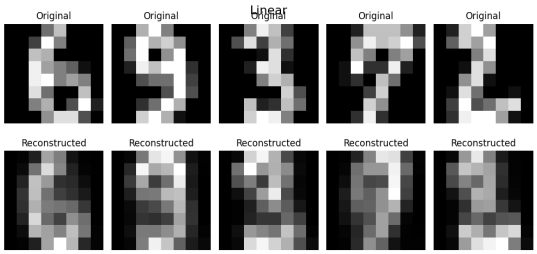
Figure 3.12: Comparaison des images reconstruites pour différentes tailles de vecteur latent



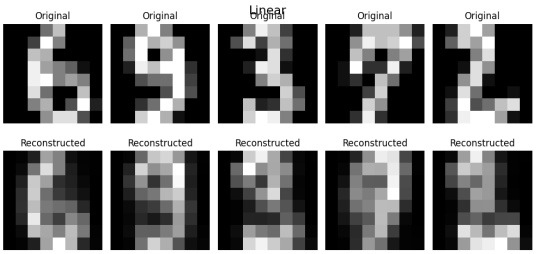
(a) 50 paramètres



(b) 100 paramètres



(c) 150 paramètres

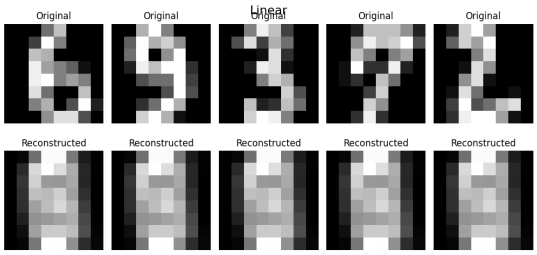


(d) 200 paramètres

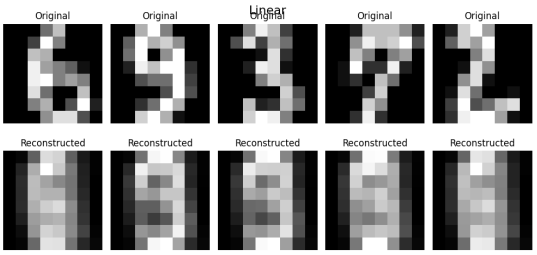
Figure 3.13: Comparaison des images reconstruites pour différentes tailles de la couche cachée de l'encodeur

Variation de la couche cachée du décodeur

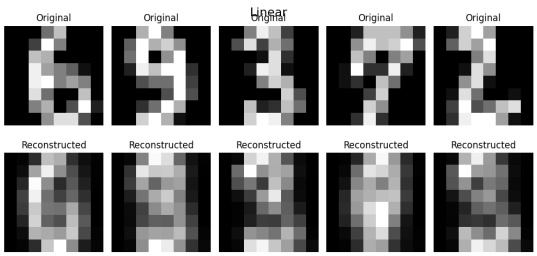
La figure 3.14 montre que l'expressivité du décodeur doit être plus importante que l'encodeur. En effet, là où l'encodeur peut obtenir de bons résultats avec une couche cachée de 100 paramètres (figure 3.13), le décodeur ne produit pas de résultat satisfaisant avant une couche cachée de 200 paramètres.



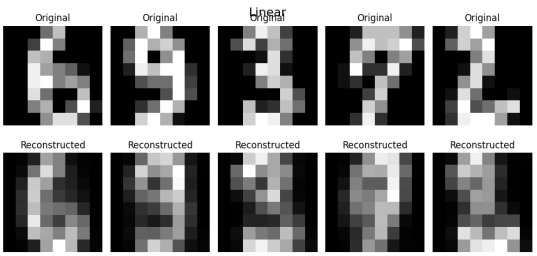
(a) 50 paramètres



(b) 100 paramètres



(c) 150 paramètres



(d) 200 paramètres

Figure 3.14: Comparaison des images reconstruites pour différentes tailles de la couche cachée du décodeur

Combinaison des résultats

La figure 3.11 nous indique un temps de train optimal de 1000 epochs. La figure 3.12 nous indique qu'il faut utiliser un vecteur latent de 50 paramètres. La figure 3.13 nous indique une couche cachée de l'encodeur de 100 paramètres. Enfin, la figure 3.14 nous indique que le

décodeur a besoin d'une couche cachée de 200 paramètres. Le seul changement par rapport à notre auto-encodeur de base est donc la taille de la couche cachée de l'encodeur.

3.3.2 Approche convolutive

Bien que nous ayions essayé d'intégrer une approche convolutive à l'autoencodeur, nous n'avons pas réussi à obtenir des résultats concluants.

CHAPTER 4

CONCLUSION

L'expérience d'implémentation de ces réseaux neuronaux a été extrêmement enrichissante, nous offrant une compréhension approfondie des concepts algébriques sous-jacents et de la subtilité de la rétropropagation. Bien que nous ayons exploré diverses tâches, nous n'avons malheureusement pas eu le temps d'implémenter des fonctionnalités plus avancées comme le clustering dans l'espace latent d'un auto-encodeur. L'utilisation de GPU pour améliorer les performances de nos modèles et accélérer nos calculs aurait pu nous aider et nous permettre d'explorer des architectures plus complexes ainsi que des ensembles de données plus vastes.

En conclusion, bien que notre exploration ait été limitée par des contraintes de temps, cette expérience nous a permis de mieux appréhender les fondements du deep learning. Elle nous a également fait prendre conscience de l'importance cruciale des choix architecturaux, de l'optimisation matérielle et de l'implémentation de fonctionnalités avancées pour obtenir de meilleures performances et pour explorer des domaines encore plus étendus dans le domaine du deep learning.