Faculté des Sciences et Ingénierie — Sorbonne Université

Master Informatique — Parcours Intelligence Artificielle et Données (MIND)

ML — Machine Learning

Rapport de Projet

Réseau de neurones : DIY

Réalisé par :

Sara ELAMOURI : 21106775

Habiba ELHUSSIENY : 21105223

Date : Mai 2025

Sommaire

Introduction

Ce projet a pour objectif la conception et l'implémentation, en `Python`, d'une bibliothèque de réseaux de neurones construits entièrement manuellement, sans recours à des frameworks haut niveau comme `PyTorch` ou `TensorFlow`.

L'approche adoptée repose sur une **architecture modulaire** : chaque couche du réseau est modélisée comme un module indépendant, qu'il s'agisse de couches linéaires, de fonctions d'activation (Tanh, Sigmoïde) ou encore de fonctions de coût (MSE, Cross-Entropy). La rétropropagation est implémentée à partir de la dérivation en chaîne, et les modules sont chaînés via une classe `Séquentiel` pour faciliter l'assemblage des réseaux.

L'optimisation des réseaux est assurée par une descente de gradient stochastique (SGD), encapsulée dans une classe `Optim`, permettant une mise à jour efficace des paramètres. L'ensemble de cette infrastructure a été validé sur plusieurs cas d'usage : *régression*, *classification binaire*, *classification multi-classe* sur MNIST, et *auto-encodage*.

Ce projet vise ainsi à **approfondir la compréhension du fonctionnement interne des réseaux de neurones**, tout en fournissant une base solide et évolutive pour expérimenter différentes architectures et techniques d'apprentissage supervisé ou non supervisé.

Ce rapport est structuré en plusieurs sections : une première sur l'implémentation pas à pas, suivie d'une phase expérimentale, et se concluant par une analyse critique et des perspectives.

2. Implémentation

2.1 Composants fondamentaux du réseau

Chaque composant du réseau (couche linéaire, fonction d'activation, etc.) est encapsulé dans une classe héritant d'une interface commune nommée `Module`. Cette approche modulaire permet une organisation claire et extensible. Elle facilite le chaînage des opérations, rend la rétropropagation cohérente et garantit la réutilisabilité du code dans différentes architectures. Chaque module dispose d'une interface unifiée avec les méthodes essentielles : `forward()`, `backward_update_gradient()`, `backward_delta()` et `update_parameters()`.

2.1.1 Couches linéaires

Les couches linéaires, représentées par la classe `Linear`, effectuent des transformations affines selon la formule :

$$sortie = XW + b$$

où W est la matrice des poids et b est le vecteur de biais. À l'initialisation, les poids sont échantillonnés depuis une distribution normale centrée, et les biais sont initialisés à zéro.

- La méthode `forward` calcule la sortie du module.
- La méthode `backward_update_gradient` accumule les gradients à partir du mini-batch.
- La méthode `backward_delta` propage le gradient vers les couches précédentes.

Chaque mise à jour est effectuée par `update_parameters()`, suivie de `zero_grad()` pour réinitialiser les gradients.

2.1.2 Fonctions d'activation : Tanh, Sigmoïde, Softmax, Log-Softmax

Tanh : Cette fonction est utilisée principalement dans les couches cachées pour centrer les données autour de zéro. Elle est définie par :

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Sa dérivée, utilisée pour la rétropropagation, est :

$$\tanh'(x) = 1 - \tanh^2(x)$$

Sigmoïde : Principalement utilisée en sortie pour la classification binaire, elle transforme toute valeur réelle en une sortie comprise entre 0 et 1 :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sa dérivée est :

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Softmax : Cette fonction d'activation est utilisée en sortie des réseaux pour la classification multi-classes. Elle convertit un vecteur de scores (logits) en une distribution de probabilités :

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

Elle est différentiable et garantit que la somme des sorties vaut 1. Dans notre projet, Softmax est utilisée uniquement dans la couche finale, juste avant le calcul de la perte.

LogSoftmax : Variante stabilisée de la Softmax, elle combine Softmax et logarithme naturel :

$$LogSoftmax(z_i) = \log\left(\frac{e^{z_i}}{\sum_k e^{z_k}}\right)$$

Elle est souvent utilisée avec la fonction de perte NLL (Negative Log Likelihood) car elle évite les instabilités numériques dues à des valeurs trop grandes dans l'exponentielle. Comme Softmax, elle est utilisée en couche de sortie.

Chaque fonction d'activation est implémentée comme un module indépendant en héritant de la classe `Module`. Les fonctions `forward()` et `backward_delta()` sont redéfinies selon les spécificités de chaque activation. Aucune de ces fonctions n'a de paramètre à apprendre ; ainsi, les méthodes `backward_update_gradient()` et `update_parameters()` sont redéfinies comme vides.

2.2 Fonctions de coût

Les fonctions de coût mesurent l'écart entre la sortie prédite par le modèle et la vérité terrain. Elles jouent un rôle essentiel dans l'apprentissage car elles guident la mise à jour des paramètres.

2.2.1 Erreur quadratique moyenne (MSE)

Utilisée dans les tâches de régression, la MSE est définie par :

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Elle est facile à implémenter, différentiable, et adaptée aux sorties continues.

2.2.2 Entropie croisée (CE) et LogSoftmax

Pour la classification multi-classe, on utilise une combinaison de `LogSoftmax` (stabilise les calculs) avec la `NLLLoss` :

$$NLL(y, \log \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Cela permet d'optimiser directement la log-vraisemblance.

2.2.3 Cross Entropy Binaire (BCE)

Utilisée pour la classification binaire avec une sortie sigmoïde :

$$BCE(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Cette fonction est sensible à la précision des probabilités extrêmes, et une version stabilisée (avec clipping) est utilisée dans le projet.

2.3 Construction modulaire des architectures

2.3.1 Classe Séquentiel

La classe `Sequentiel` permet d'empiler dynamiquement les modules du réseau. Chaque appel à `add_module` ajoute une couche au modèle. Elle gère le passage avant complet, la rétropropagation, la mise à jour des paramètres et le reset des gradients. La gestion de l'historique des entrées intermédiaires facilite le backward.

2.3.2 Implémentation d'un auto-encodeur

Un auto-encodeur a été implémenté avec une architecture symétrique : encodeur (réduction de dimension) + décodeur (reconstruction). Chaque bloc est un `Sequentiel` distinct. Il est entraîné avec la BCE sur le dataset MNIST prétraité. Ce modèle est utile pour la compression ou le débruitage d'images.

2.4 Optimisation et apprentissage

2.4.1 Descente de gradient stochastique (SGD)

L'algorithme d'optimisation utilise la descente de gradient stochastique avec mini-batchs. À chaque itération, un batch est échantillonné, l'erreur est mesurée, les gradients sont propagés, et les paramètres sont mis à jour. L'apprentissage se fait sur plusieurs époques avec un mélange aléatoire des données.

2.4.2 Optimiseur : classe Optim

Cette classe encapsule toutes les étapes d'un pas d'optimisation : remise à zéro des gradients, passe forward, calcul de la perte, rétropropagation, et mise à jour des poids. Elle centralise la logique d'apprentissage pour une réutilisabilité maximale.

2.4.3 Recherche d'hyperparamètres

Le projet inclut une étude des effets de plusieurs hyperparamètres (learning rate, batch size, nombre d'époques). Ces paramètres ont été évalués sur différentes tâches pour observer leur influence sur la stabilité, la convergence et la performance finale.

3. Expérimentations

3.1 Classification

Dans cette partie, nous nous intéressons à la classification supervisée des images de la base de données MNIST, qui contient 60 000 images d'entraînement et 10 000 images de test représentant des chiffres manuscrits (de 0 à 9). L'objectif est d'entraîner un réseau de neurones capable d'attribuer à chaque image le chiffre correspondant. Pour commencer, nous avons conçu un réseau simple : Linear(784,10) → Softmax() Le modèle est entraîné avec la fonction de perte CrossEntropy, qui est bien adaptée aux tâches de classification multi-classe.

3.1.1 Effet du learning rate

Le taux d'apprentissage est un hyperparamètre essentiel qui détermine la vitesse à laquelle un modèle ajuste ses poids lors de l'entraînement. Un choix judicieux de cette valeur est crucial pour assurer une convergence efficace et stable du modèle.

Un taux trop élevé (par exemple, 0.1) peut provoquer des oscillations dans la fonction de perte (loss) et une instabilité dans la accuracy, notamment sur les données de test d'après la figure 1. Cela s'explique par le fait que le modèle effectue des mises à jour de poids trop importantes, ce qui peut le faire dépasser le minimum de la fonction de perte, l'empêchant ainsi de converger correctement .

À l'inverse, un taux trop faible (par exemple, 0.0001 ou 0.001) ralentit considérablement l'apprentissage comme on peut le voir sur les figures 3 et 4. Le modèle met beaucoup de temps à réduire la perte, ce qui peut entraîner une convergence lente et inefficace. De plus, il risque de rester bloqué dans des minima locaux, n'atteignant pas la performance optimale. Donc, c'est le taux d'apprentissage = 0.01 qui maximise la accuracy tout en convergeant.

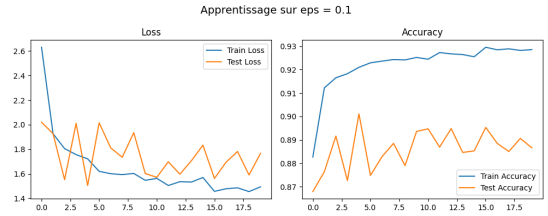

FIGURE 1 – Taux d'apprentissage = 0.1

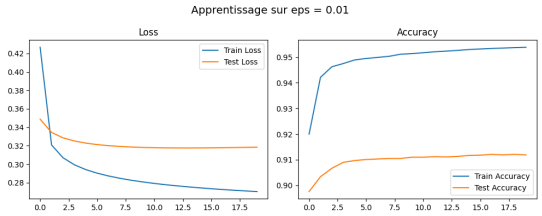

FIGURE 2 – Taux d'apprentissage = 0.01

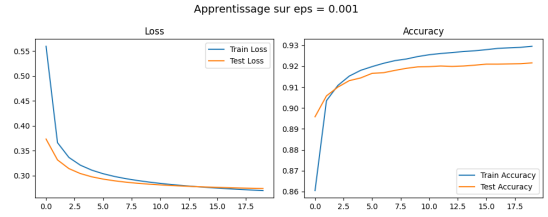

FIGURE 3 – Taux d'apprentissage = 0.001

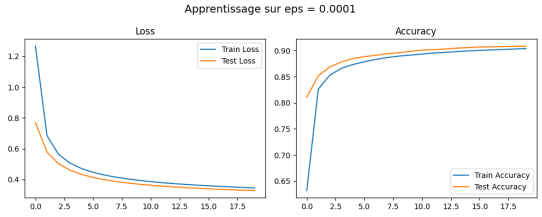

FIGURE 4 – Taux d'apprentissage = 0.0001

3.1.2 Taille des batchs

La taille des batchs est un hyperparamètre crucial qui influence la vitesse de convergence, la stabilité de l'entraînement et la capacité de généralisation du modèle.

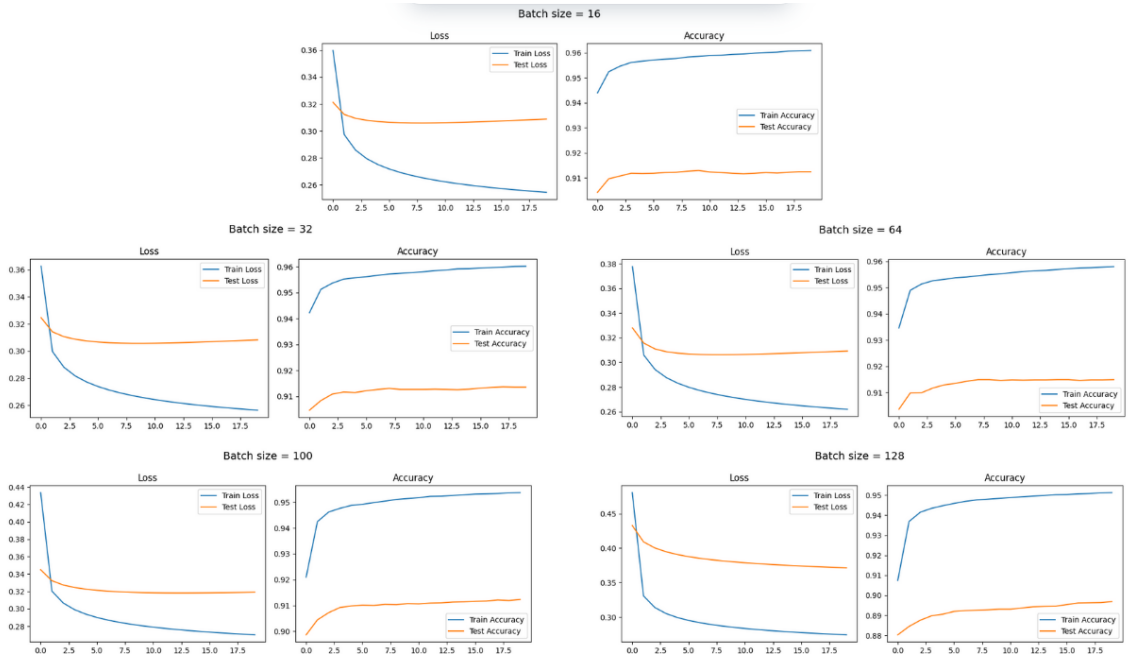

FIGURE 5 – Impact de la taille du batch sur la loss et le score

Nous observons que de meilleures performances sont obtenues avec des batchs de petite taille : comme le montre la Figure 5, une taille de batch de 16 ou 32 permet d'atteindre une précision d'environ 91,5% sur l'ensemble de test.

À l'inverse, en augmentant la taille du batch (par exemple à 100 ou 128), la précision test diminue légèrement. Bien que la loss sur l'entraînement continue de décroître, la courbe de précision sur les données de test indique une généralisation moins efficace. Cela pourrait s'expliquer par une moindre diversité des gradients mis à jour à chaque itération, limitant ainsi la capacité du modèle à explorer des minima plus généralisables.Ce résultat confirme que, dans certains cas, l'utilisation de plus petits batchs favorise une meilleure robustesse du modèle.

3.1.3 Impact du nombre de couches

Pour étudier l'influence de la profondeur du réseau, nous avons tester trois architectures differentes.

`Réseau simple : Linear(784,10)` $\rightarrow$ `Softmax()`. Cette architecture correspond à un classifieur linéaire, équivalent à une régression logistique multinomiale. D'après la figure 6, elle offre des performances de base raisonnables, mais elle est limitée dans sa capacité à capturer des structures complexes dans les données.

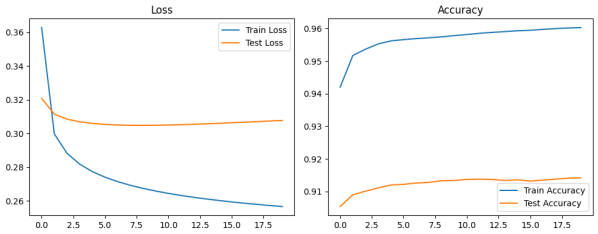

FIGURE 6 – Réseau simple

`Réseau complexe : Linear(784,128)` $\rightarrow$ `TanH()` $\rightarrow$ `Linear(128,10)` $\rightarrow$ `Softmax()`.

Ce changement a permis au réseau d'apprendre des représentations intermédiaires non linéaires, ce qui a rapidement amélioré la précision sur l'ensemble de test, notamment dans les premières époques et a également accéléré l'apprentissage global en facilitant la convergence du modèle.

`Réseau très complexe : Linear(784,128)` → `TanH()` → `Linear(128,64)` → `TanH()` → `Linear(64,10)` → `Softmax()`. Dans la figure 8, l'ajout d'une deuxième couche cachée a continué à améliorer légèrement les performances, tout en ralentissant légèrement la convergence. On remarque également une légère instabilité dans les courbes de perte et de précision, comparée au réseau à une seule couche cachée (figure 7). En effet, un début de surapprentissage semble apparaître.

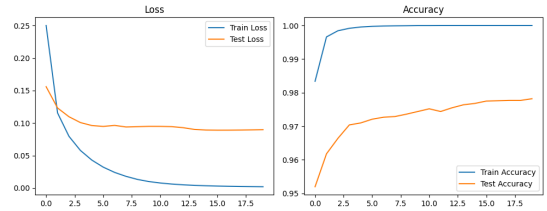

FIGURE 7 – Réseau complexe avec une couche cachée

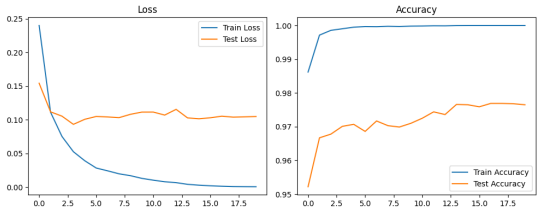

FIGURE 8 – Réseau très complexe avec deux couches cachées

Ces resulats confirme qu'il est essentiel de trouver un compromis entre profondeur, capacité du réseau et généralisation, adapté à la complexité réelle des données.

3.1.4 Comparaison des fonctions d'activation

Dans le but d'évaluer l'impact des fonctions d'activation sur les performances du classifieur, nous avons expérimenté différentes activations non linéaires dans les couches cachées du réseau, par exemple : Linear(784, 128) → activation → Linear(128, 10) → Softmax(). Les fonctions évaluées sont TanH, Sigmoid, Softmax (interne), et LogSoftmax. Remarque : on teste différentes fonctions d'activation dans les couches cachées, mais on garde toujours un Softmax en sortie pour que le réseau puisse produire des probabilités interprétables et utiliser la CrossEntropyLoss de manière correcte.

Les résultats montrent que TanH est la fonction la plus performante. Elle permet une convergence rapide, une perte qui diminue régulièrement pour l'entraînement et le test, et une accuracy test d'environ 98% dès les premières époques d'après la figure 9. Cela s'explique par sa capacité à produire des activations centrées autour de zéro, ce qui facilite la descente de gradient et la stabilité des poids.

La fonction Sigmoid donne aussi de très bons résultats, avec une précision test légèrement inférieure mais stable autour de 97.8% (figure 10), et des courbes de loss très régulières. Toutefois, comme attendu, on observe une saturation plus rapide des neurones, autrement-dit, une convergence un peu plus lente que TanH.

En revanche, l'utilisation de Softmax comme activation dans la couche cachée s'est révélée inappropriée. Les courbes de loss augmentent continuellement et la précision chute jusqu'à 45% comme on peut le voir sur la figure 11, ce qui montre que le modèle n'apprend rien de significatif. Cela confirme que Softmax est mal adapté comme activation intermédiaire, car il contraint les sorties à une distribution de probabilité dès les couches internes, ce qui écrase l'information utile à la classification.

Enfin, avec LogSoftmax, les sorties étant des log-probabilités (souvent négatives), l'incompatibilité avec la fonction de perte CrossEntropy a conduit à une explosion immédiate des valeurs : le réseau ne peut pas apprendre.

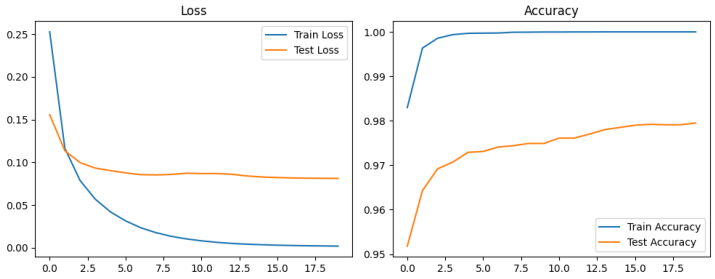

FIGURE 9 – Fonction de loss et score en utilisant TanH

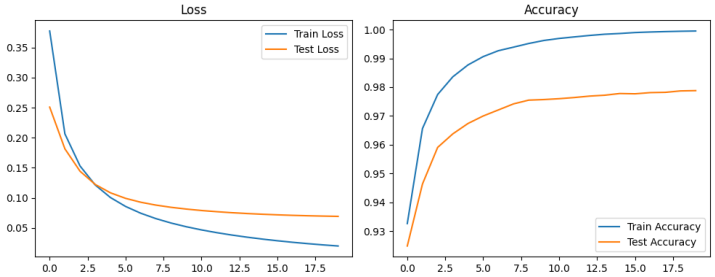

FIGURE 10 – Fonction de loss et score en utilisant Sigmoid

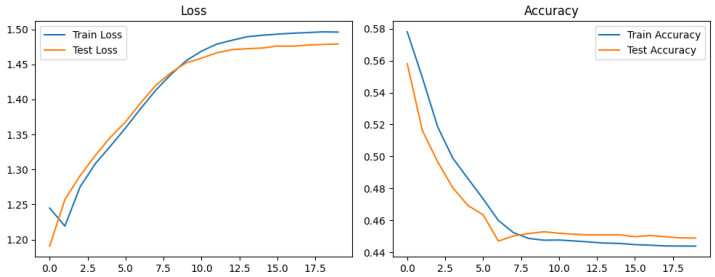

FIGURE 11 – Fonction de loss et score en utilisant Softmax

Apres qu'on a fait la comparaison des figures 9, 10 et 11, on conclut que la fonction TanH offre un bon compromis entre rapidité de convergence, stabilité de la loss et précision finale.

3.1.5 Effet de la loss

Nous avons comparé deux fonctions de perte couramment utilisées : Mean Squared Error (MSE) et Cross-Entropy Loss, afin d'évaluer leur effet sur la qualité d'apprentissage dans cete tâche de classification multi-classe.

Dans la figure 12, avec `MSE`, la perte décroît rapidement vers 0, et les précisions atteignent 99.9% en entraînement et 97.75% en test. Cependant, les valeurs de loss sont extrêmement basses car la MSE mesure un écart quadratique moyen, qui tend naturellement vers 0 sur des sorties continues.

Dans la figure 13, avec `Cross Entropy`, la perte reste dans une échelle plus élevée. Pourtant, les performances finales sont équivalentes à celles obtenues avec la MSE, voire légèrement meilleures en généralisation : la test accuracy est identique voire supérieure, avec une convergence plus fluide.

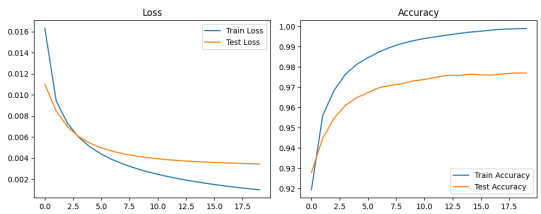

FIGURE 12 – MSE

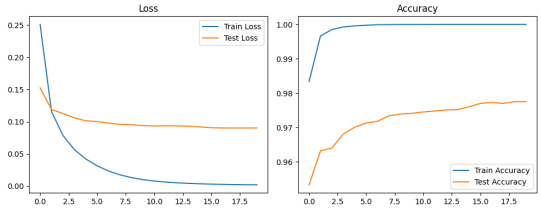

FIGURE 13 – Cross Entropy

Même si la MSE donne l'illusion d'une loss très faible, elle n'est pas naturellement adaptée à la classification, car elle ne prend pas en compte la structure probabiliste des classes et ne pénalise pas efficacement les erreurs de classement. En effet, la MSE a tendance à "moyenner" les erreurs, ce qui peut conduire à des sorties intermédiaires (entre 0 et 1) plutôt qu'à des décisions nettes. Contrairement à la Cross-Entropy, elle ne pousse pas fortement les sorties vers la classe correcte (valeur proche de 1) et les autres vers 0.

La Cross-Entropy, en revanche, est mathématiquement bien adaptée à la classification, surtout lorsqu'elle est combinée à une Softmax en sortie. Elle maximise directement la vraisemblance de la bonne classe et produit des gradients plus significatifs pour guider l'apprentissage.

D'autre part, pour explorer un peu, nous avons testé l'utilisation de la `BCE loss`. Comme on peut le voir dans la figure 14, elle s'est révélée mal adaptée pour la classification multi-classe. En effet, la BCE est conçue pour des problèmes de classification binaire ou multi-label, où chaque sortie est traitée comme une probabilité indépendante via une fonction Sigmoid.

Dans notre cas, la sortie du réseau est une distribution de probabilité sur 10 classes exclusives, produite par une Softmax. Or, la BCE applique une pénalisation individuelle sur chaque sortie, ce qui entre en conflit avec le comportement interdépendant des sorties d'une Softmax, où augmenter la probabilité d'une classe diminue nécessairement celle des autres, cela engendre des gradients incohérents et perturbe l'apprentissage.

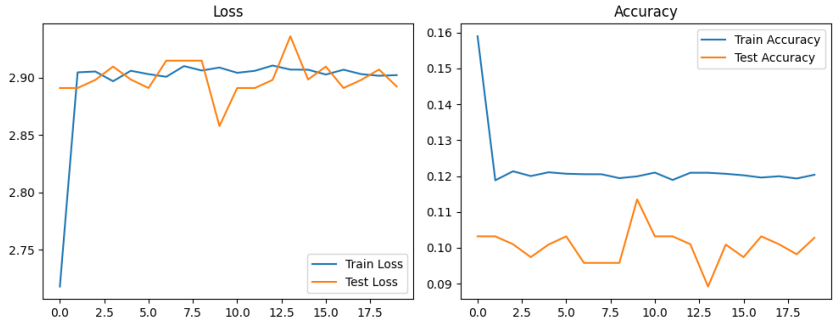

FIGURE 14 – BCE

Enfin, nous avons également testé la combinaison `LogSoftmax + NLLLoss`, qui est théoriquement équivalente à Softmax + CrossEntropyLoss (figure 13). Les résultats obtenus sont similaires (figure 15), en termes de convergence et de précision. Cette approche a aussi l'avantage de stabiliser les calculs log dès la sortie, ce qui peut être bénéfique pour éviter les erreurs numériques lorsque les probabilités deviennent très petites. De plus, nous avons observé que la précision progresse légèrement plus rapidement au début de l'apprentissage, ce qui témoigne d'une meilleure efficacité du gradient.

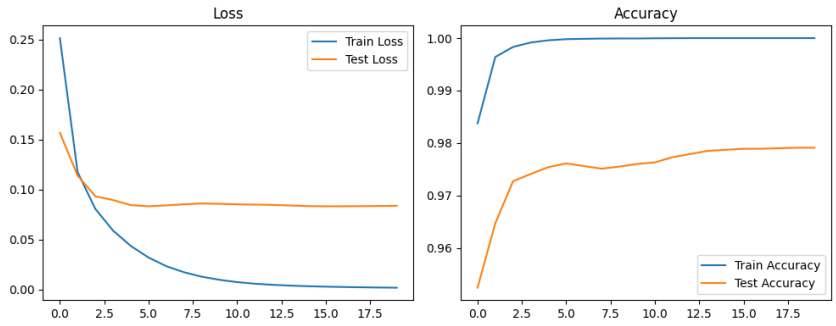

FIGURE 15 – LogSoftmax + NLLLoss

3.2 Auto-encodeur

Notre objectif dans cette partie sera d'entraîner un *auto-encodeur* capable de reconstruire les images de la même base de données MNIST à partir de leur représentation latente. L'encodeur prend les images en entrée et les transforme en une représentation latente de dimension réduite. Ensuite, le décodeur prend cette représentation latente et tente de générer des images similaires aux images d'origine. L'idée est de forcer le modèle à compresser l'information présente dans chaque image dans un espace de plus petite dimension, puis d'utiliser cette représentation comprimée pour reconstruire l'image originale. Cette capacité de reconstruction sera un bon indicateur de la qualité de l'encodage appris.

Chaque image est une matrice 28×28 représentant un chiffre manuscrit. Nous avons appliqué les traitements suivants :

— Normalisation des valeurs de pixels entre 0 et 1.
— Aplatissement des images en vecteurs de dimension 784 (28×28) pour les rendre compatibles avec des couches linéaires.

3.2.1 Analyse de différents architectures d'auto-encodeurs

Nous avons commencé par expérimenter deux architectures différentes pour notre auto-encodeur, toutes deux utilisant la fonction de perte BCE :

Architecture simple :

— **Encodeur** : Linear(784,64) → TanH()
— **Décodeur** : Linear(64,784) → Sigmoid()

Cette architecture compacte permet une réduction directe de la dimension d'entrée à un espace latent de taille 64, avec une reconstruction immédiate.

Architecture plus complexe :

— **Encodeur** : Linear(784,256) → TanH() → Linear(256,64) → TanH()
— **Décodeur** : Linear(64,256) → TanH() → Linear(256,784) → Sigmoid()

Cette version introduit des couches supplémentaires pour améliorer la capacité d'apprentissage du modèle en capturant des représentations plus abstraites.

Dans la figure 16, correspondant à l'architecture complexe, nous observons une convergence rapide de la perte dès les premières époques, atteignant environ 0,099 en fin d'entraînement.

Dans la figure 17, représentant l'architecture simple, la perte initiale est plus élevée (environ 0,25) mais diminue de manière régulière et continue, sans sursaut notable, pour atteindre environ 0,14. Cette stabilité peut être attribuée à la simplicité du modèle, qui limite les risques d'oscillations ou de sur-apprentissage.

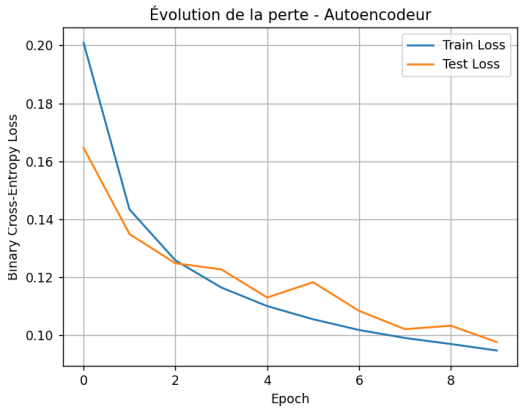

FIGURE 16 – Coût BCE, pour auto-encodeur complexe, sur 10 époques et lr =0.01

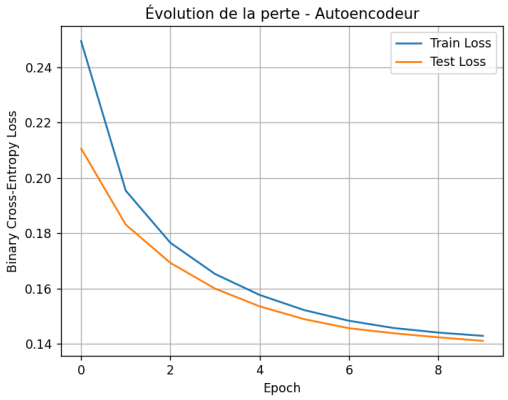

FIGURE 17 – Coût BCE, pour auto-encodeur simple, sur 10 époques et lr =0.01

En effet, l'auto-encodeur complexe fournit une reconstruction plus fidèle et plus nette que l'architecture simple, comme nous pouvons le voir dans la figure 18. Il capture des représentations plus riches : structure des traits fins est mieux conservée et le niveau de flou est réduit. Cela confirme que l'augmentation de la profondeur permet au modèle d'apprendre des représentations latentes plus informatives, au prix d'un entraînement potentiellement plus long et d'un risque de sur-apprentissage plus élevé.

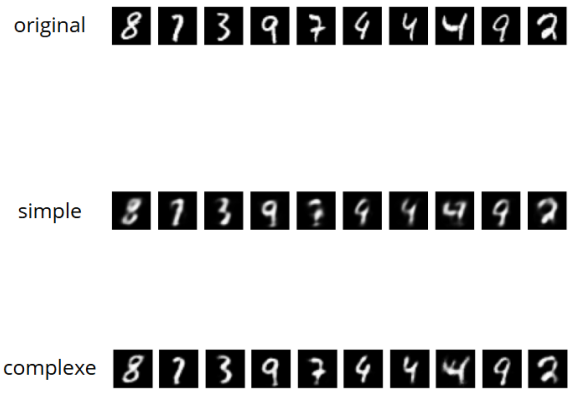

FIGURE 18 – Exemple de reconstruction en fonction de la complexité du réseau

3.2.2 Étude de l'impact de la taille de l'espace latent

Après avoir testé une architecture d'auto-encodeur avec un espace latent de dimension 64, nous avons fait varier cet hyperparamètre clé afin d'évaluer son impact sur la qualité de reconstruction et la capacité de compression du modèle. En effet, la taille du code latent détermine directement le niveau de compression appliqué à l'image initiale.

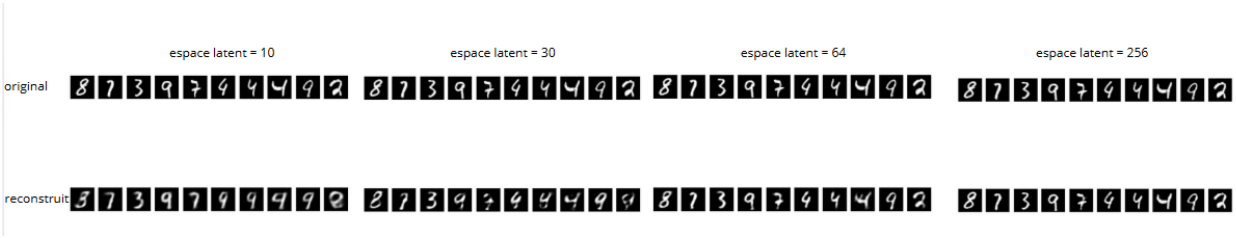

FIGURE 19 – Reconstruction avec différent taille d'espace latent

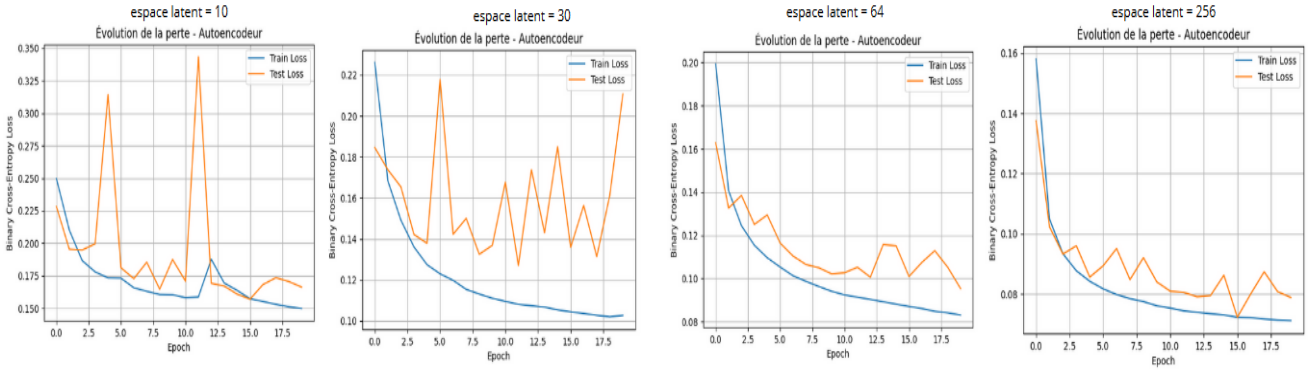

FIGURE 20 – Coût BCE en fonction de la taille de l'espace latent

Nous avons exploré plusieurs tailles d'espace latent (10, 30, 64 et 256) afin d'étudier leur influence sur la qualité de reconstruction (figure 19) et la capacité de généralisation de l'auto-encodeur (figure 20). Ainsi, d'après la figure 19 et la figure 20, on remarque qu'avec une taille réduite à 10, le modèle ne parvient pas à compresser efficacement l'information et présente des oscillations marquées de la perte de test entre les époques, ce qui suggère que l'information transmise est instable et trop contrainte. Une légère amélioration est observée en passant à 30 dimensions, avec une perte d'entraînement plus faible (0.10). Néanmoins, des fluctuations persistent. Donc, un espace latent très petit force le modèle à compresser l'information de manière très agressive. Le réseau peut alors choisir des représentations très sensibles à des variations mineures dans les données d'entrée. De plus, le réseau, faute d'espace suffisant, ne peut pas encoder de manière robuste tous les détails utiles. Il apprend alors des représentations non généralisables, ce qui cause une bonne reconstruction pour certains exemples du train, et une mauvaise sur d'autres ou sur le test et donc la loss test peut monter et descendre selon le batch vu. Ces oscillations traduisent l'incapacité du modèle à converger vers une solution stable qui reconstruit bien l'ensemble des exemples.

La configuration à 64 dimensions offre un équilibre optimal. La reconstruction est bien meilleure et presque similaire à l'original, les courbes deviennent nettement plus régulières, et la perte de test suit bien la perte d'entraînement. Cela indique un apprentissage stable, avec une bonne capacité de compression sans perte significative d'information. Enfin, pour un espace latent égal à 256, la reconstruction est très fidèle (presque parfaite), avec une perte d'entraînement très basse, mais la courbe test commence à diverger légèrement. Cela suggère un début de surapprentissage car le modèle dispose alors d'une capacité si élevée qu'il mémorise les exemples au lieu d'apprendre des représentations compressées

généralisables. Ainsi, bien que 256 offre la meilleure reconstruction visuelle, cela se fait au prix d'une compression faible qui est l'un des objectifs fondamentaux d'un auto-encodeur et d'un risque de suradaptation.

Donc, l'ensemble des observations – numériques et visuelles – confirme que 64 peut constituer le meilleur compromis entre fidélité de reconstruction, compression efficace et généralisation, particulièrement adaptée aux données de la base MNIST et ce réseau complexe.

3.2.3 Comparaison des fonctions d'activations

Nous avons évalué l'impact de différentes fonctions d'activation dans l'auto-encodeurs, en remplaçant les `TanH()` internes par : `Sigmoid()`, `Softmax()` et `LogSoftmax()`.

Les résultats, affichés sur les Figures 21 et 22, confirment que `TanH` donne les meilleures performances globales, avec une reconstruction nette et une bonne convergence, avec une perte minimale. `Sigmoid` fournit également des résultats acceptables, avec une reconstruction légèrement plus floue et une perte plus grande.

En revanche, les versions utilisant `Softmax` dans les couches cachées échouent à apprendre une représentation utile : la perte ne diminue que très lentement, et les reconstructions sont quasi aléatoires ou très dégradées.

Dans un objectif d'exploration, nous avons testé l'usage de la fonction `LogSoftmax` comme activation finale du décodeur. Cette fonction, conçue pour la classification multiclasse, produit des log-probabilités négatives ou nulles, ce qui n'est pas adapté à la reconstruction de données d'entrée normalisées entre 0 et 1. Autrement dit, les valeurs de sortie sont incompatibles avec la fonction de perte `BCE`, et donc le calcul donne des valeurs incorrectes, souvent supérieures à 1, voire incohérentes, comme on le voit sur la Figure 22.

L'entraînement avec cette configuration a conduit à une perte plus élevée, des reconstructions visuellement dégradées, et une instabilité dans l'apprentissage. Cela s'explique par le fait que `LogSoftmax` contraint les sorties à une forme logarithmique peu expressive pour une image. Ce test confirme qu'il est important d'adapter la fonction d'activation de sortie au type de données reconstruites.

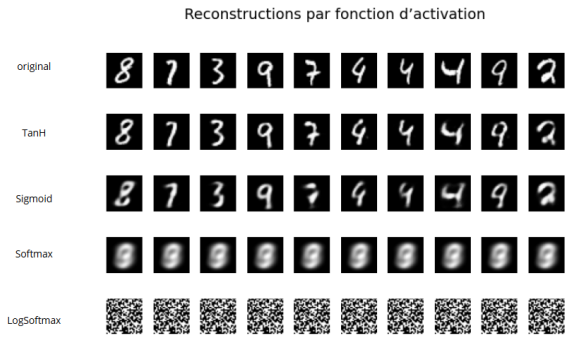

FIGURE 21 – Reconstructions par fonction d'activation

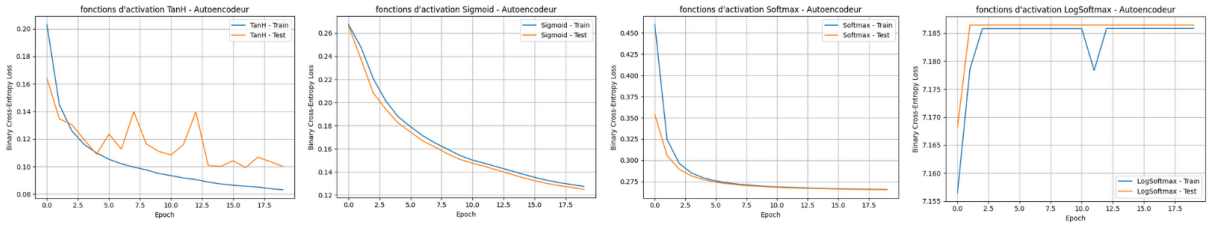

FIGURE 22 – Loss pour chaque fonction d'activation

Conclusion

L'implémentation manuelle de réseaux de neurones dans ce projet a été une expérience particulièrement formatrice. Elle nous a permis de comprendre en profondeur les mécanismes fondamentaux des réseaux de neurones, notamment la construction modulaire de réseaux, la propagation avant, la rétropropagation du gradient, et l'optimisation par descente de gradient.

Dans un premier temps, nous avons abordé la classification supervisée d'images de chiffres manuscrits à partir du jeu de données MNIST. À travers l'expérimentation de différentes architectures (réseaux simples ou profonds), nous avons pu évaluer concrètement l'impact de paramètres clés tels que le taux d'apprentissage, la taille des batchs et la profondeur du réseau sur la performance en entraînement et en test.

Dans un second temps, la mise en œuvre d'un autoencodeur a mis en lumière la richesse du concept de compression non supervisée et les subtilités liées au choix d'architecture, à la fonction de perte, et au comportement du modèle pendant l'entraînement. Nous avons pu explorer différentes configurations architecturales pour l'autoencodeur, comparer leurs performances sur le jeu de données MNIST, et visualiser l'impact des choix de dimensions latentes et du nombre d'époques sur la qualité de reconstruction. Ce travail a renforcé notre compréhension des compromis entre capacité du modèle et généralisation.

D'autres pistes de recherche restent ouvertes et mériteraient d'être explorées, notamment :

— le *clustering* dans l'espace latent pour évaluer la structuration des représentations internes ;
— l'utilisation de l'espace latent comme vecteur d'entrée pour des tâches de classification supervisée ;
— ou encore l'extension vers des architectures convolutionnelles.

Par ailleurs, l'absence d'accès à des ressources matérielles avancées comme les GPU a limité nos expérimentations à des modèles de taille modérée, ralentissant l'entraînement et restreignant notre capacité à tester des configurations plus complexes.

En conclusion, bien que notre exploration ait été limitée par des contraintes de temps, ce projet nous a non seulement permis d'acquérir des compétences pratiques en implémentation de réseaux de neurones, mais aussi de mieux appréhender les enjeux liés à la conception, l'entraînement, et l'analyse de modèles d'apprentissage profond.