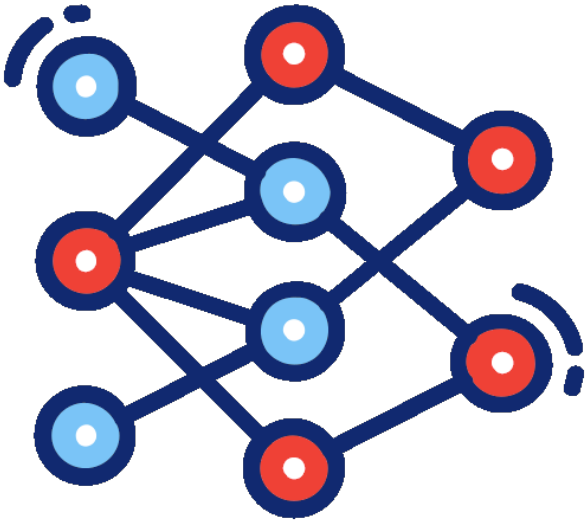
ML - Projet

Réseau de neurones : DIY

Charles Vin, Aymeric Delefosse



SORBONNE UNIVERSITÉ

DAC | 2022-2023

Table des matières

1 Introduction

Ce projet, réalisé entièrement en Python et aux concepts algébriques, avait pour objectif de développer une bibliothèque de *deep learning*. À travers des expérimentations significatives, nous avons exploré diverses tâches telles que la reconstruction d'images, la génération de données et l'entraînement de réseaux convolutionnels. Ces expérimentations nous ont permis d'analyser l'impact des choix architecturaux, des techniques d'optimisation, des hyperparamètres et des contraintes matérielles sur les performances des modèles. Ce projet a ainsi contribué à approfondir notre compréhension du *machine learning*.

Note : n'hésitez pas à zoomer sur les figures !

2 Implémentation

Le code est disponible sur ce dépôt GitHub.

— Implémentation optimisée de la convolution avec le moins de boucles possibles (utilisation de fonctions avancées de `numpy`) ;

— Une documentation en ligne ;

— Implémentation d'un `logger` qui peut afficher graphiquement la *loss* après chaque époque ;

— Early stopping qui permet de réduire les temps d'apprentissage et les risques de sur-apprentissage.

La bibliothèque est organisée de manière analogue aux bibliothèques de *deep learning* plus populaires.

3 Expérimentation

Dans le but de mettre notre bibliothèque à l'épreuve, nous avons entrepris une série d'expérimentations en utilisant diverses architectures sur deux types de données distincts. Le premier ensemble de données était constitué de jeux aléatoires de données gaussiennes, présentant des caractéristiques linéairement séparables ou non, tels que le XOR ou le jeu d'échecs. Le second ensemble de données concernait la classification d'images sur deux jeux de données : un pour la reconnaissance de chiffres (le plus populaire : `USPS`) et le deuxième, plus complexe, pour la reconnaissance de vêtements (`Fashion MNIST`).

3.1 Classification

3.1.1 Effet de l'initialisation des paramètres

Nous avons très vite constaté le rôle et l'impact de l'initialisation des paramètres sur les performances des réseaux de neurones. Cette initialisation peut paraître anodine à première vue mais joue un rôle crucial sur la performance et la significativité du modèle.

Une initialisation inadéquate peut entraîner des problèmes tels que la saturation des neurones, la divergence de l'apprentissage ou la stagnation dans des minima locaux. Une initialisation judicieuse peut quant à elle favoriser une convergence plus rapide et une meilleure généralisation des données.

Ainsi, l'optimisation des poids et des biais à l'initialisation constitue une étape cruciale dans la conception et l'entraînement des réseaux de neurones. Cette initialisation peut se faire en prenant en compte les

spécificités de l'architecture et de la fonction d'activation.

Ainsi, pour les modules utilisant des paramètres (linéaire et convolution), il est possible d'initialiser les poids et biais de huit manières différentes :

— Initialisation normale : les paramètres sont initialisés selon une loi normale centrée réduite : $W, B \sim \mathcal{N}(0, 1)$;

— Initialisation uniforme : les paramètres sont initialisés selon une loi uniforme : $W, B \sim \mathcal{U}(0, 1)$;

— Initialisation à 1 : tous les paramètres sont initialisés à 1 ;

— Initialisation à 0 : tous les paramètres sont initialisés à 0 ;

— Initialisation de Xavier (ou de Glorot) : cette méthode, développée par Xavier Glorot et Yoshua Bengio, ajuste les poids de manière à maintenir une variance constante tout au long du réseau, en fonction du nombre d'entrées et de sorties de chaque couche. Cela favorise une propagation efficace du signal. Il est possible d'initialiser selon une loi normale ou une loi uniforme, où input représente la taille de l'entrée dans le cadre d'un module linéaire, le nombre de canaux d'entrée dans le cadre d'une convolution et output représente la taille de la sortie dans le cadre d'un module linéaire, le nombre de canaux en sortie (le nombre de *feature maps*) dans le cadre d'une convolution.

  — Loi normale : $W, B \sim \mathcal{N}(0, \sqrt{\frac{2}{\text{input}+\text{output}}})$;

  — Loi uniforme : $W, B \sim \mathcal{U}(-\sqrt{\frac{6}{\text{input}+\text{output}}}, \sqrt{\frac{6}{\text{input}+\text{output}}})$.

— Initialisation de He (ou de Kaiming) : Cette méthode, développée par Kaiming He et al., est similaire à l'initialisation de Xavier, mais elle prend en compte la variance spécifique des fonctions d'activation asymétriques, telles que la fonction ReLU. Elle permet une initialisation adaptée aux architectures utilisant ces fonctions d'activation.

  — Loi normale : $W, B \sim \mathcal{N}(0, \sqrt{\frac{2}{\text{input}}})$;

  — Loi uniforme : $W, B \sim \mathcal{U}(-\sqrt{\frac{6}{\text{input}}}, \sqrt{\frac{6}{\text{input}}})$.

Note : pour le module linéaire, les biais sont initialisés par défaut tandis que pour la convolution, les biais sont désactivés par défaut (sauf indication contraire).

Pour évaluer l'effet de l'initialisation des paramètres, nous avons mis en place un réseau simple pour classer le jeu de données `fashion-mnist`. Ce réseau se compose d'une couche linéaire qui prend en entrée l'image complète et effectue la classification avec dix neurones en sortie. Nous représentons ce réseau sous la forme suivante : `Linear(784, 10)` $\rightarrow$ `Sigmoid()`.

La figure 1 présente les résultats de cette expérimentation. Nous observons des difficultés d'apprentissage avec certaines initialisations. L'initialisation normale conduit à un minimum local (après 200 époques). Les initialisations uniforme et à 1 provoquent une explosion du gradient, empêchant tout apprentissage.

Cette expérience met en évidence l'importance de l'initialisation des paramètres. Il serait peut-être possible de résoudre les problèmes liés à l'initialisation normale en utilisant un autre algorithme d'optimisation, tel que Adam.

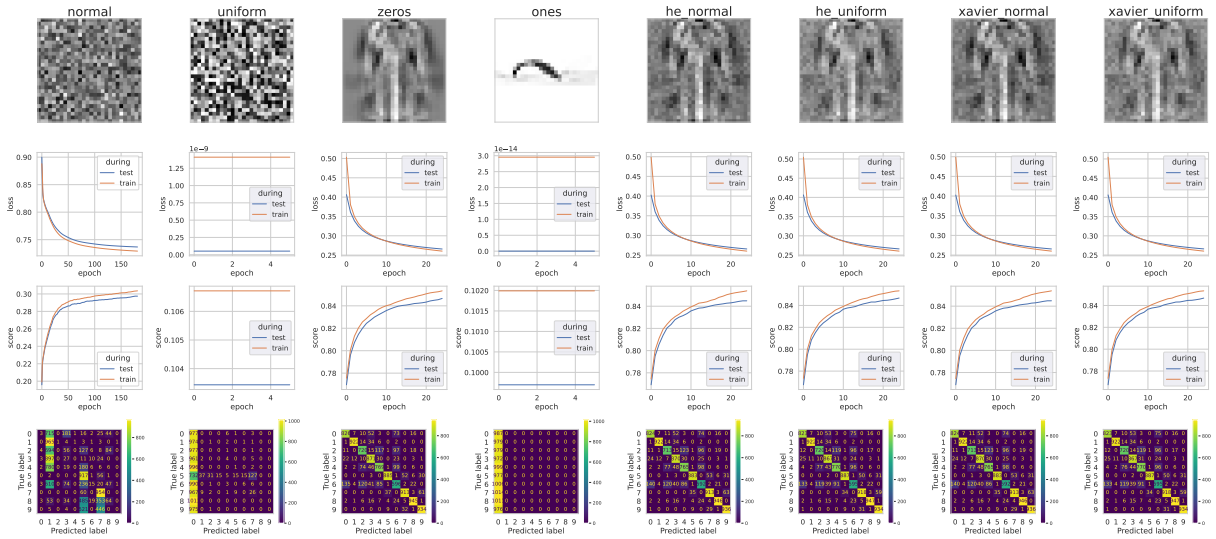


FIGURE 1 – Représentation des paramètres, d'une matrice de confusion, de l'évolution du coût et du score par époque en fonction de l'initialisation des paramètres

3.1.2 Effet du taux d'apprentissage (*learning rate*)

Le *learning rate* est un hyperparamètre crucial dans l'entraînement des réseaux de neurones. Il contrôle la taille des pas que l'algorithme d'optimisation effectue lors de la mise à jour des poids du réseau pendant l'apprentissage. Un taux d'apprentissage adapté permet aux mises à jour de poids d'être suffisamment grandes pour converger rapidement vers un minimum, tout en évitant les oscillations et les divergences. Le but est donc de trouver un bon compromis entre la rapidité de convergence et la stabilité des résultats.

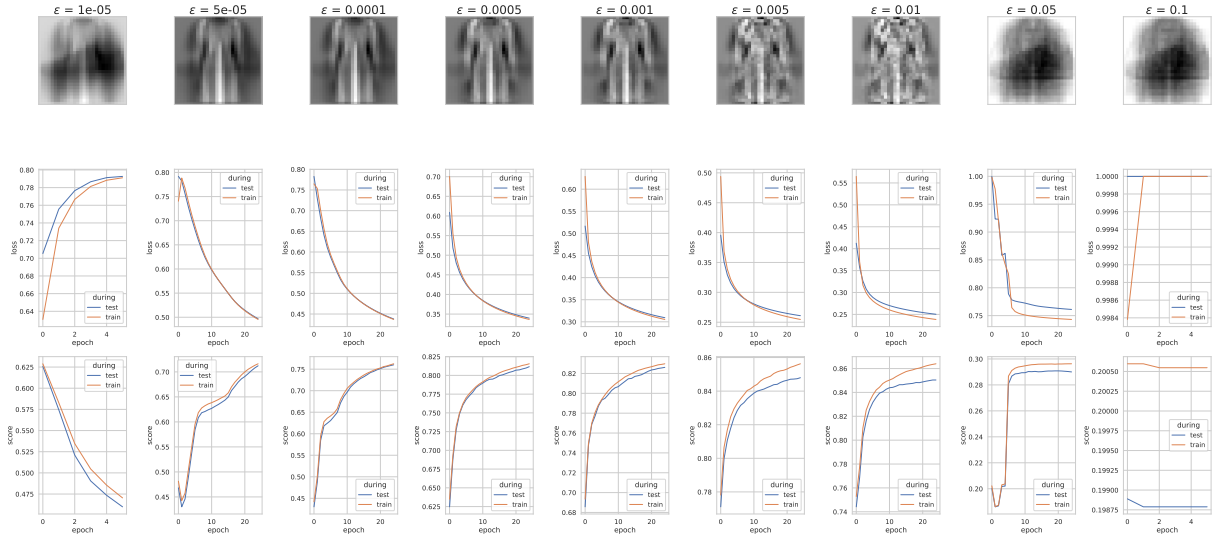


FIGURE 2 – Représentation des paramètres, de l'évolution du coût et du score par époque en fonction du taux d'apprentissage ϵ

Toujours avec le même réseau très simple, nous obtenons les résultats dans la figure 2. Un *learning rate* trop faible ralentit donc la convergence, et un nombre élevé d'itérations sera nécessaire pour atteindre une performance obtenue avec un taux d'apprentissage plus élevé. Les ajustements apportés aux poids sont

également plus lents et progressifs, au risque de rester coincer dans des minima locaux peu optimaux. Un *learning rate* trop élevé entraîne une divergence dans notre cas, les poids se mettent à jour trop brutalement entraînant une convergence vers un modèle très rapidement sous-optimal.

3.1.3 Effet des fonctions d'activation

L'ajout d'une fonction d'activation non linéaire après un module linéaire, tel qu'une couche dense ou une convolution, permet d'introduire des interactions non linéaires entre les neurones et d'augmenter la capacité de représentation du modèle. Sans fonction d'activation, le réseau de neurones se réduirait simplement à une combinaison linéaire des entrées, limitant ainsi sa capacité à modéliser des relations complexes.

Différentes fonctions d'activation peuvent être utilisées en fonction du problème et des caractéristiques des données. Par exemple :

- La fonction tangente hyperbolique qui comprime les sorties entre -1 et 1 ;
- La fonction sigmoïde qui comprime les sorties entre 0 et 1 ;
- La fonction sigmoïde "stable" qui a été implémentée dans l'optique d'éviter des instabilités numériques ;
- La fonction softmax pour transformer les données en une distribution de probabilités ;
- La fonction log-softmax afin d'éviter des instabilités numériques ;
- La fonction ReLU ;
- La fonction LeakyReLU afin d'éviter du *gradient vanishing* ;
- La fonction SoftPlus qui est une autre approximation de la fonction ReLU.

On pourrait, rajouter encore d'autres fonctions d'activation, telles que la fonction identité, une identité courbée, une sinusoïde, un sinus cardinal, une fonction gaussienne...

Le choix de la fonction d'activation dépend du problème à résoudre, des caractéristiques des données et de l'architecture du réseau. Chaque fonction d'activation a des propriétés différentes et peut être plus adaptée à certains types de problèmes ou à certaines architectures. Les résultats sont visibles sur la figure 3. Certains résultats nous étonnent, notamment pour la tangente hyperbolique.

3.2 Classification et convolution

L'utilisation de réseaux de neurones convolutifs offre de nombreux avantages pour le traitement des images. La convolution permet de capturer des motifs locaux, de réduire le nombre de paramètres à apprendre, d'obtenir une certaine invariance aux translations et de réduire la dimensionnalité des données. Cela en fait un module particulièrement adapté aux images, aux séquences temporelles ou aux signaux.

Nous avons implémenté avec succès la convolution en une dimension, ainsi que les couches de pooling (Max et Average). L'implémentation a été optimisée en utilisant le moins de boucles possible, en se basant sur les fonctionnalités avancées de la bibliothèque `numpy` (seulement une boucle présente dans la convolution, sur la taille du kernel). Notre raisonnement est détaillé au sein du code.

Un réseau utilisant une seule couche convolutionnelle de ce type `Conv1D(3,1,32)` → `MaxPool1D(2,2)` → `Flatten()` → `Linear(4064,100)` → `ReLU()` → `Linear(100,10)` fonctionne extrêmement bien dans

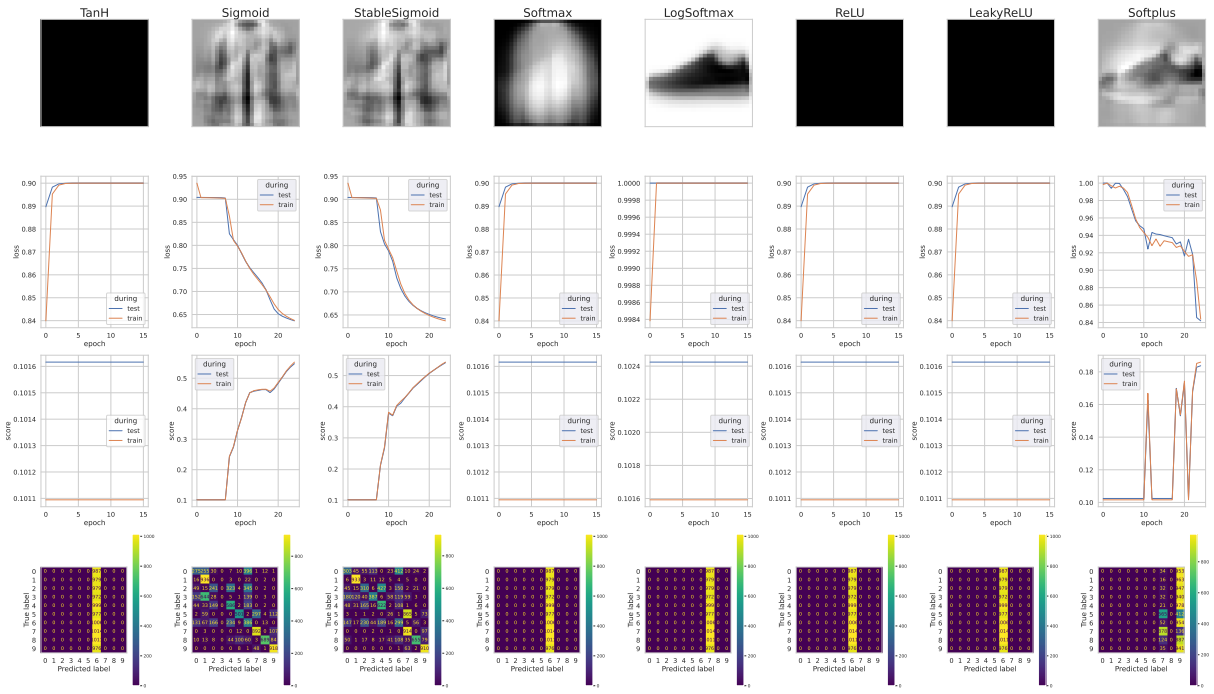


FIGURE 3 – Représentation des paramètres, de l'évolution du coût et du score par époque en fonction de la fonction d'activation

le cadre de la reconnaissance de chiffres.

En utilisant un batch de taille fixe de 32, 50 époques et un early-stopping à 5, répétées 10 fois, nous obtenons un score d'apprentissage de 99.36 ± 0.0017 % et un **score d'évaluation** de $\mathbf{97.08 \pm 0.0016}$ %, en seulement 31 ± 6 époques.

Par ailleurs, l'initialisation des poids est d'une grande importance dans les couches convolutionnelles. Par exemple, une initialisation à zéro ne fonctionne pas (voir figure 4). Dans notre cas, nous avons utilisé une initialisation de Xavier avec une loi normale pour les convolutions, et une initialisation de He normale pour les modules linéaires.

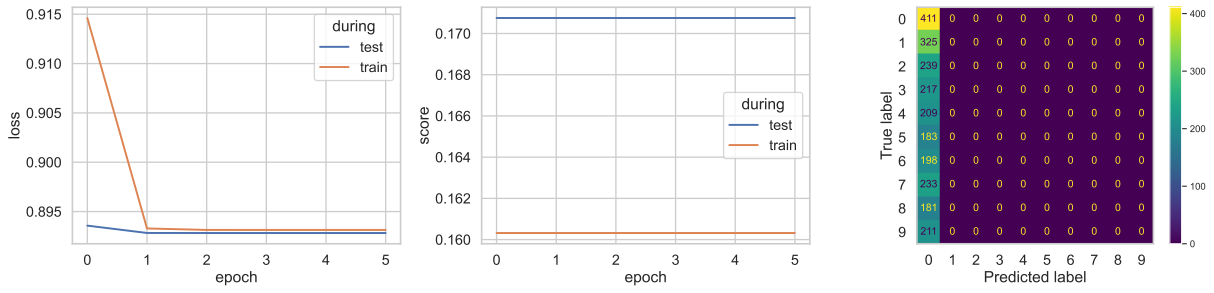


FIGURE 4 – Impact de l'initialisation des poids à zéros sur un réseau de neurones convolutionnel

3.2.1 Effet de la taille des *batch*

Le nombre de batchs est un hyperparamètre crucial de notre modèle. Nous observons que de meilleures performances sont obtenues en effectuant l'apprentissage sur des batchs plus petits, ce qui est une pra-

tique courante et bien connue en apprentissage profond. Cependant, cela peut actuellement entraîner un surapprentissage, même si les résultats en évaluation sont bons. De plus, plus la taille du batch est réduite, plus l'apprentissage prendra du temps. Avec des batchs plus grands, le temps d'apprentissage sera considérablement réduit, mais cela peut également avoir un impact sur les instabilités numériques, comme l'explosion du gradient. Ces résultats sont illustrés dans la figure 5. Il est important de trouver un équilibre entre la taille du batch, les performances du modèle, la durée de l'apprentissage et les problèmes d'instabilité numérique. Nous utiliserons une taille de batch fixe de 32 pour la suite.

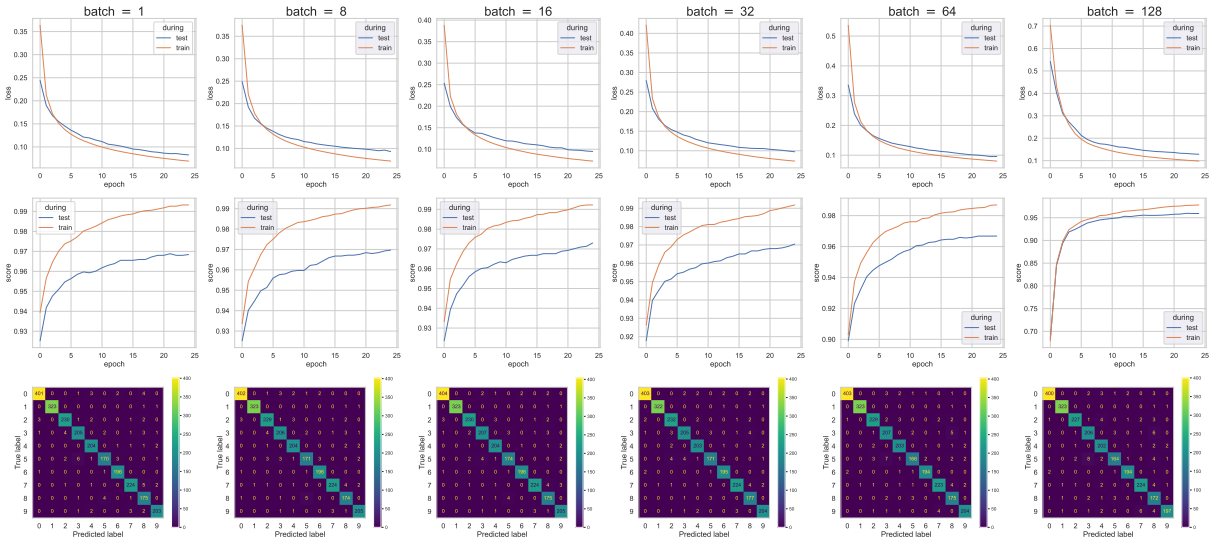


FIGURE 5 – Impact de la taille du kernel pour un réseau de neurones convolutionnel

3.2.2 Effet de la taille du kernel

En ajustant la taille du kernel, nous avons la possibilité de capturer des motifs ou des informations à différentes échelles spatiales dans les données. Par exemple, les kernels de petite taille peuvent être efficaces pour détecter des détails fins, tandis que les kernels de plus grande taille sont plus adaptés pour détecter des caractéristiques plus globales. Lors de l'analyse des chiffres manuscrits, ces chiffres sont des éléments cruciaux de l'image, bien que l'image elle-même soit de petite taille. Même en utilisant un kernel de taille 10, qui commence à être relativement grand, les performances restent similaires à celles obtenues avec un kernel de taille 3. Cependant, des problèmes surviennent lorsque nous utilisons un kernel de taille 15 ou plus, ce qui affecte les performances du modèle.

3.2.3 Effet du nombre de *feature maps*

Le nombre de *feature maps* dans une couche de convolution joue un rôle crucial dans la capacité du modèle à capturer des caractéristiques et des motifs complexes, ce qui améliore la représentation des données et la capacité de généralisation. Cependant, il est important de trouver un équilibre en tenant compte des contraintes de ressources et des exigences spécifiques du problème. Dans notre cas, nous avons constaté qu'une seule *feature map* était étonnamment suffisante, tandis que 128 *feature maps* fonctionnaient également bien. L'utilisation d'une seule feature map peut être attribuée à la simplicité

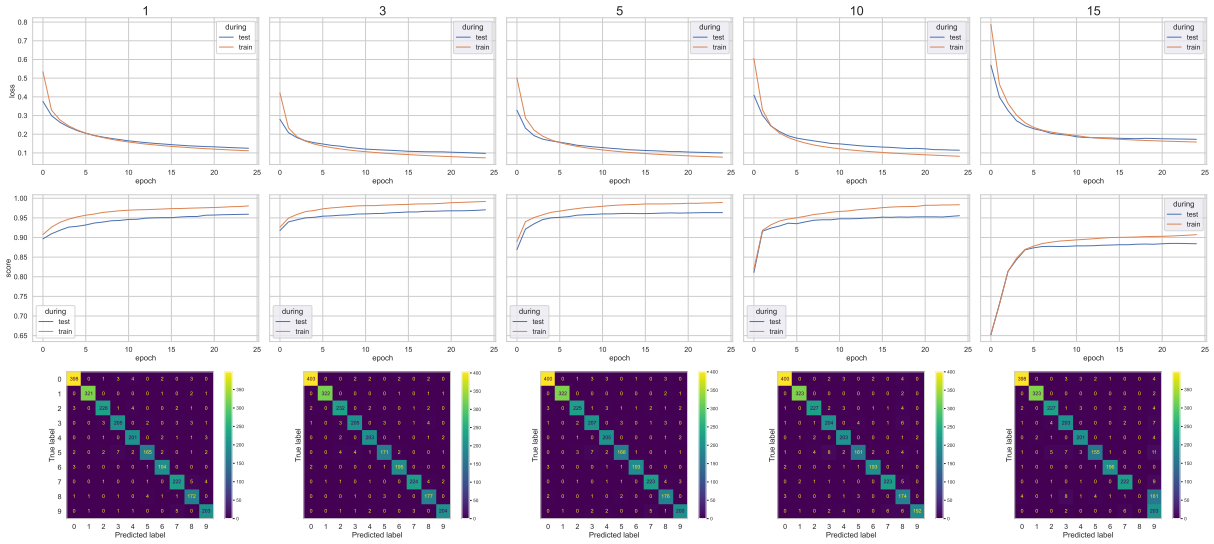


FIGURE 6 – Impact de la taille du kernel pour un réseau de neurones convolutionnel

du problème de reconnaissance de chiffres manuscrits, où les caractéristiques discriminantes peuvent être relativement simples et bien représentées par une seule feature map. En revanche, l'ajout de 128 *feature maps* peut permettre une représentation plus riche et plus complexe des données, ce qui peut être bénéfique pour capturer des détails plus fins ou des motifs plus variés présents dans les images.

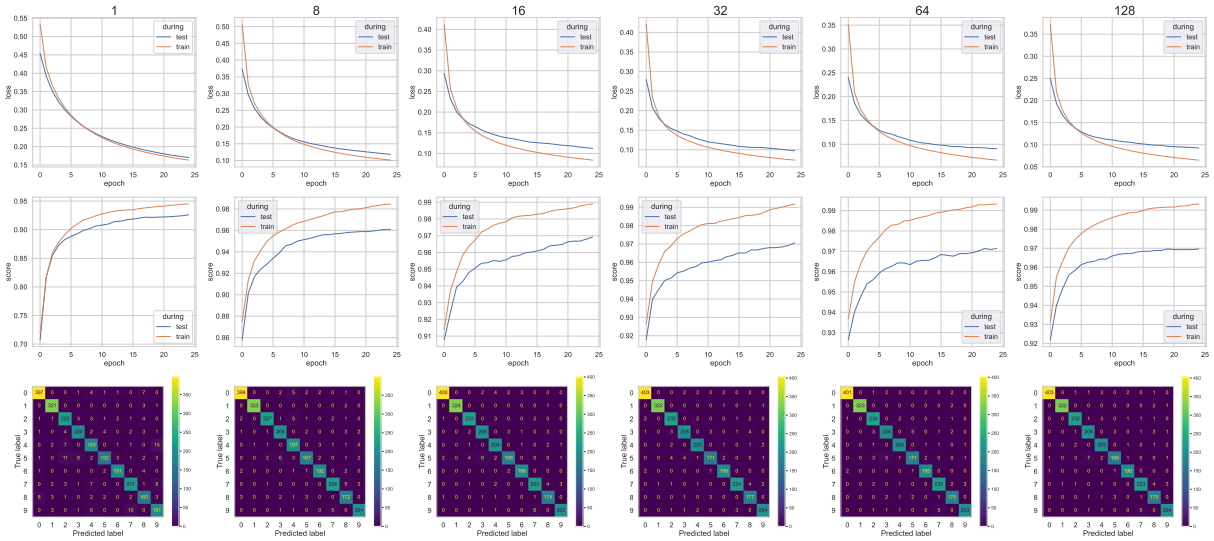


FIGURE 7 – Impact du nombre de *feature maps* pour un réseau de neurones convolutionnel

3.2.4 Diverses architectures

Nous avons également exploré différentes architectures pour évaluer la robustesse de la convolution, en particulier en augmentant la profondeur des réseaux et en appliquant des fonctions d'activation directement sur les convolutions, comme c'est couramment fait. Cependant, ces variations n'ont pas donné de résultats significativement meilleurs. En fait, nous avons constaté que l'augmentation de la taille du réseau de neurones convolutifs ne se traduit pas nécessairement par une meilleure classification.

L'architecture à deux couches convolutionnelles, représentée par `Conv1D(3, 1, 32, 1)` → `ReLU()` → `MaxPool1D(2, 2)` → `Conv1D(3, 32, 32)` → `ReLU()` → `MaxPool1D(2, 2)` → `Flatten()` → `Linear(1984, 10)`, a montré des résultats mitigés (figure 8), où le réseau commence à apprendre avant de régresser soudainement. Les performances obtenues sont nettement inférieures à celles du réseau précédent, indiquant que l'utilisation d'une seule couche linéaire n'est probablement pas suffisante pour classifier correctement nos données.

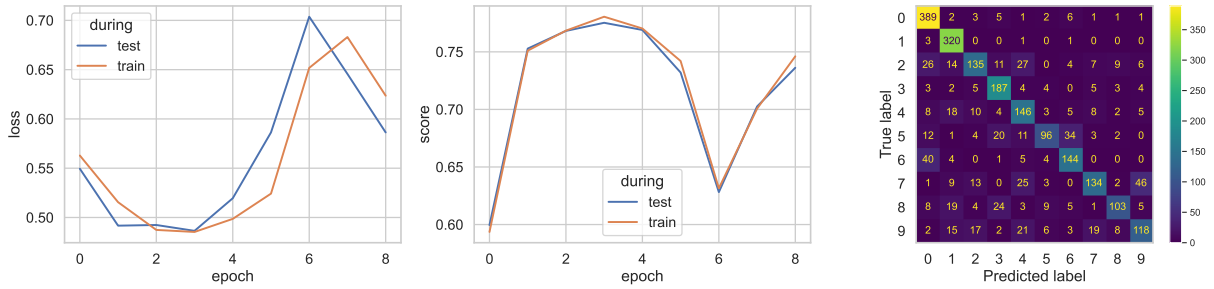


FIGURE 8 – Réseau de neurones convolutionnel à 2 couches

En ce qui concerne l'architecture à quatre couches convolutionnelles, représentée par `Conv1D(3, 1, 64, 1)` → `ReLU()` → `MaxPool1D(8, 2)` → `Conv1D(3, 64, 64)` → `ReLU()` → `MaxPool1D(8, 2)` → `Conv1D(3, 64, 64)` → `ReLU()` → `MaxPool1D(8, 2)` → `Conv1D(3, 64, 64)` → `ReLU()` → `MaxPool1D(8, 2)` → `Flatten()` → `Linear(512, 10)`, les résultats (figure 9) montrent une meilleure capacité d'apprentissage par rapport au réseau précédent, mais les performances ne sont toujours pas satisfaisantes. Nous avons également observé que la classe 2 est celle qui est la moins bien classée, ce qui contribue à ces performances insuffisantes. Il est possible que la taille du kernel dans la couche de Max Pooling soit trop élevée pour conserver un nombre significatif de features, ce qui empêche une identification précise de cette classe.

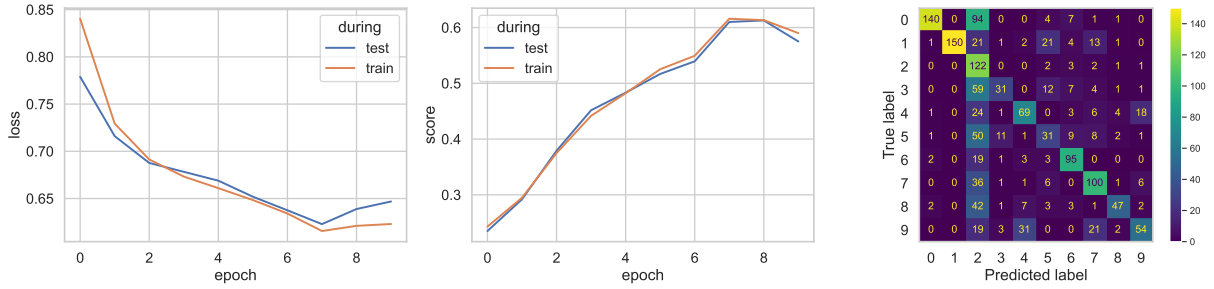


FIGURE 9 – Réseau de neurones convolutionnel à 4 couches

Un modèle trop complexe n'est donc pas synonyme de bonne performance.

3.2.5 Et sur `fashion-mnist` ?

Avec une architecture similaire à celle utilisée pour les chiffres, l'utilisation d'un réseau convolutionnel n'améliore pas de manière significative les scores de classification (passant de 84 % à un peu plus de 85 % de précision). En revanche, pour le jeu de données USPS, l'ajout d'une couche de convolution permet une

nette amélioration des performances, passant de 85 % à 97 % de précision.

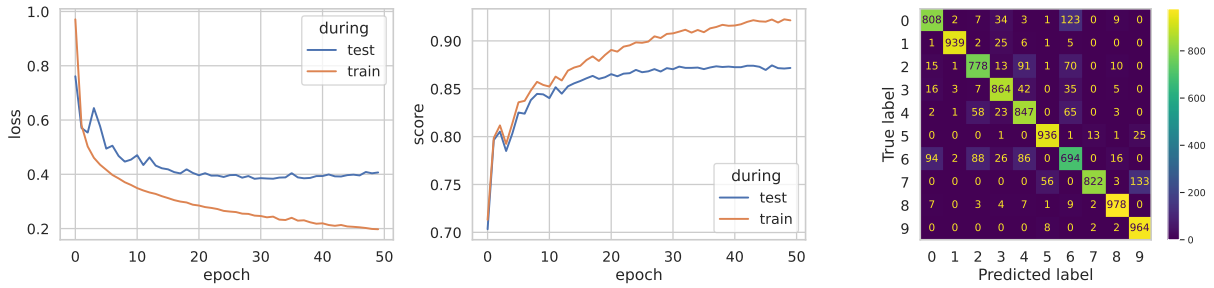


FIGURE 10 – Réseau de neurones convolutionnel à 1 couche

Avec un modèle à 3 couches, où l'objectif est d'avoir une sortie plus petite dans la première couche linéaire, l'apprentissage devient plus difficile, tout comme pour les chiffres. De plus, la classe 6 est mal classée et se confond avec les classes 0, 2 et 4 (qui correspondent toutes à des hauts). Par conséquent, l'application successive de trois convolutions sur les données entraîne une perte d'informations trop importante, en particulier dans une dimension, ce qui se traduit par des performances inférieures.

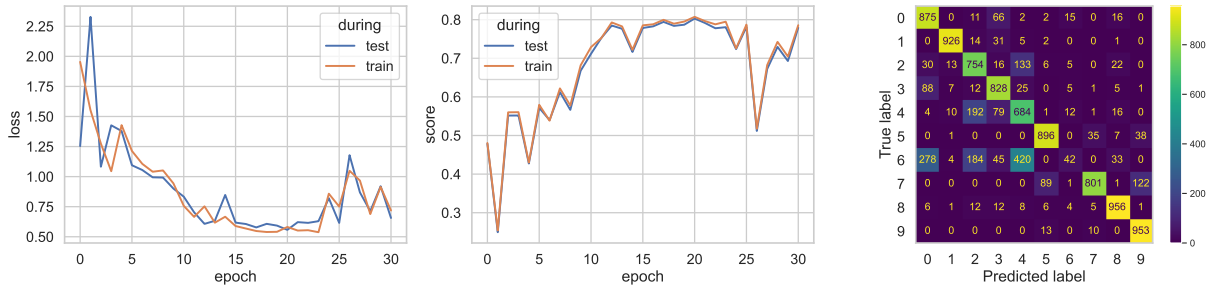


FIGURE 11 – Réseau de neurones convolutionnel à 3 couches

3.3 Auto-encodeur

Nous avons également exploré l'utilisation d'architectures encodeur-décodeur (ou auto-encodeurs), pour des tâches de reconstruction ou de génération de données.

3.3.1 Reconstruction

Dans notre cas, nous avons entraîné un modèle pour reconstruire les images d'origine 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'objectif de cette expérience était d'évaluer la capacité du modèle à capturer les caractéristiques distinctives des images et à reconstruire des images de qualité. La performance du modèle peut être évaluée en mesurant la différence entre les images reconstruites et les images d'origine.

De plus, l'architecture de l'encodeur-décodeur peut également être utilisée pour générer de nouvelles images similaires à celles du jeu de données MNIST : en fournissant une représentation latente aléatoire

au décodeur, il est possible de générer de nouvelles images qui ressemblent aux images du jeu de données.

Nous avons entrainé un auto-encodeur sur les deux jeux de données précédents (`USPS` et `fashion-mnist`).

`fashion-mnist` Dans le cas du jeu de données `fashion-mnist`, nous avons testé trois architectures pour l'auto-encodeur :

1. `Linear(784, 64)` → `TanH()` → `Decoder`
2. `Linear(784, 256)` → `TanH()` → `Linear(256, 64)` → `TanH()` → `Decoder`
3. `Linear(784, 512)` → `TanH()` → `Linear(512, 256)` → `TanH()` → `Linear(256, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()` → `Decoder`

`Decoder` représente l'architecture transposée du réseau avec une `Sigmoid()` en sortie et l'utilisation d'une `BCELoss`.

Ces trois architectures, de complexité croissante, ont toutes conduit à une stagnation rapide du coût d'évaluation pendant l'entraînement (figure 12). Cette valeur varie légèrement en fonction de la complexité du réseau. Dans la figure 13, nous pouvons effectivement constater que la précision de la reconstruction des images s'est améliorée à mesure que la complexité du réseau augmentait, ce qui suggère que des architectures plus complexes sont capables de capturer et de reconstruire les caractéristiques distinctives des images avec une meilleure fidélité.

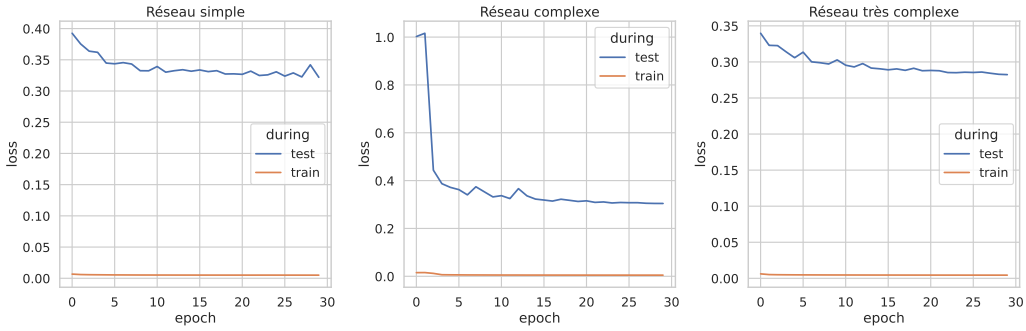


FIGURE 12 – Coût cross-entropique binaire, pour chaque modèle, sur 30 époques

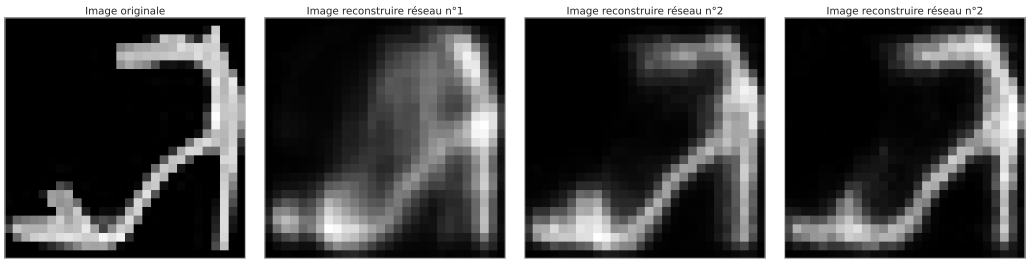


FIGURE 13 – Exemple de reconstruction en fonction de la complexité du réseau

Pour le réseau n° 1, une étude de l'influence de la fonction d'activation placée avant l'espace latent a été réalisée. Le réseau a donc la forme suivante : `Linear(784, 64)` → `Activation()` → `Decoder`. Les résultats sont présentés dans la figure 14.

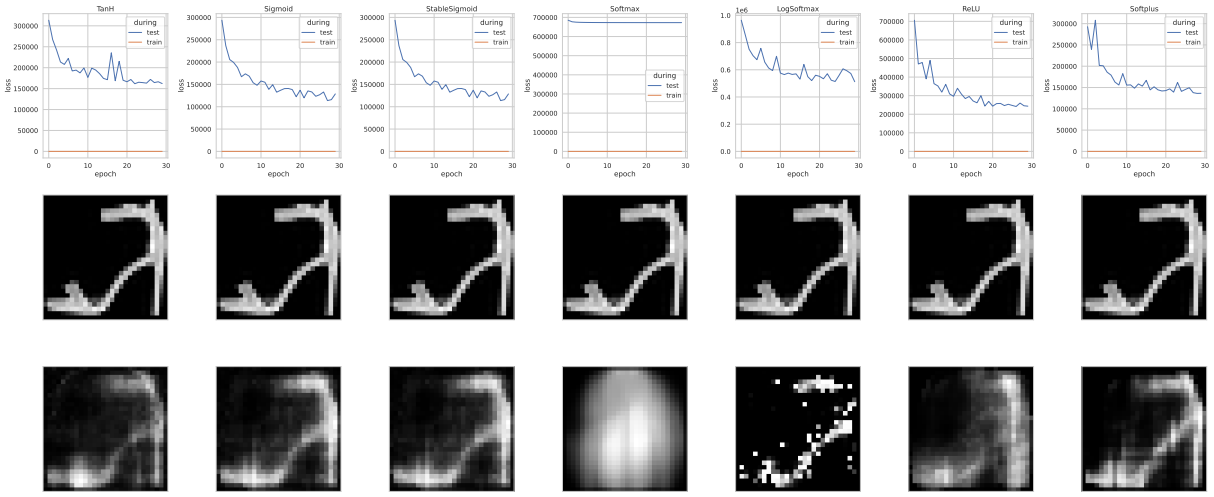


FIGURE 14 – Fonction de coût et reconstruction pour une architecture de type réseau n° 1

Il est observé que les fonctions d'activation `Softmax()` et `LogSoftmax()` rencontrent des difficultés d'apprentissage importantes. De plus, la fonction d'activation `ReLU` apprend moins rapidement que les autres fonctions d'activation.

Sur les figures 15 et 16, nous nous sommes également amusés à visualiser la reconstruction de l'image au fil des époques d'apprentissage, afin d'observer l'évolution de la qualité de la reconstruction à mesure que le modèle apprend.

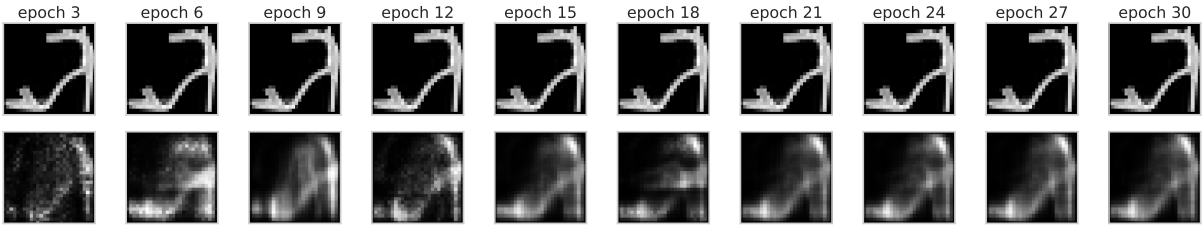


FIGURE 15 – Reconstruction par époque pour le modèle n° 1

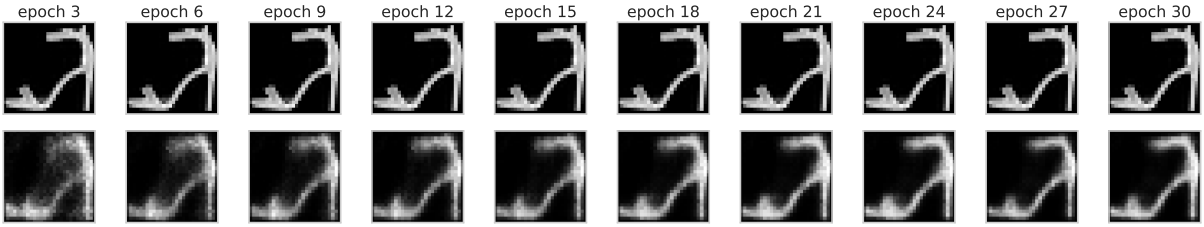


FIGURE 16 – Reconstruction par époque pour le modèle n° 3

3.3.2 Organisation de l'espace latent

Afin de visualiser l'espace latent de 64 dimensions, nous avons utilisé l'algorithme t-SNE pour le réduire à deux dimensions afin de le rendre visualisable. Cette procédure a été appliquée aux deux jeux de données et à chaque ensemble d'entraînement et de test, en utilisant l'auto-encodeur avec le plus de

paramètres. Les t-SNE sur le jeu de test présentent des similarités avec ceux du jeu d'entraînement, et nous présenterons ici uniquement les visualisations des données d'entraînement, illustrées par les figures 17 et 18.

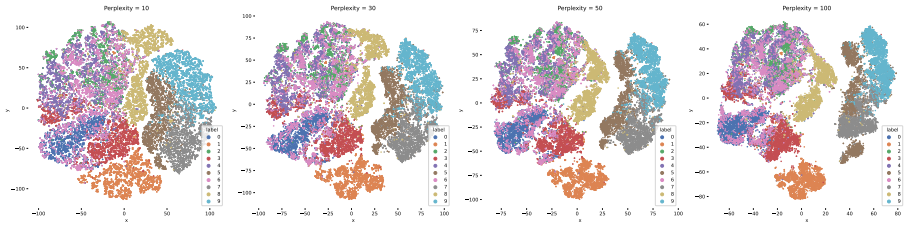


FIGURE 17 – t-SNE sur l'espace latent du modèle n° 3 pour `fashion-mnist` en utilisant les données d'entrainement

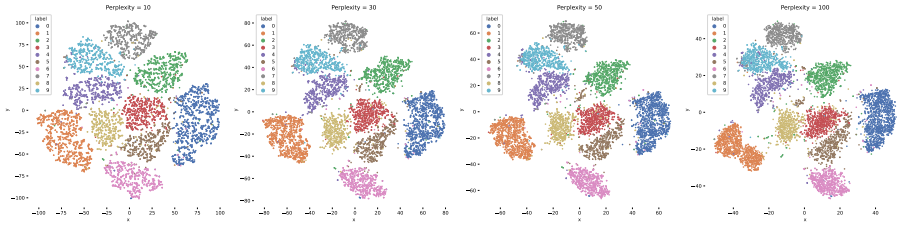


FIGURE 18 – t-SNE sur l'espace latent d'un plus gros modèle pour `USPS` en utilisant les données d'entrainement

Les résultats sont plutôt intéressants. En effet, on observe que les classes 0, 2, 4 et 6 ne forment pas vraiment des groupes distincts. Ces classes correspondent respectivement aux T-shirts/hauts, aux pulls, aux manteaux et aux chemises. Les classes des hauts étant visuellement similaires, ces classes se mélangent davantage lors de la réduction à deux dimensions. En revanche, des classes telles que les pantalons (classe 1) ou les chaussures (classe 5) ont des caractéristiques plus distinctes, ce qui les rend plus facilement discernables dans l'espace latent réduit.

3.3.3 Débruitage

Notre objectif est de générer des versions débruitées des images d'origine, ce qui équivaut à un travail de restauration / reconstruction. Cela démontre la capacité du modèle à éliminer le bruit et à récupérer les informations essentielles des images initiales. Les trois modèles précédemment entraînés ont été utilisés pour cette tâche.

En appliquant un bruit gaussien de 20 % sur une image de chaque classe, le modèle le plus simple, présenté dans la figure 19, s'en sort plutôt bien. Bien qu'il y ait une perte d'informations, notamment au niveau des contours, il parvient à retrouver la forme générale de chaque classe tout en préservant fidèlement les niveaux de gris. Il reste donc limité quant à la capture des détails fins des images.

Le deuxième modèle, illustré dans la figure 20, offre de meilleures performances en termes de reconstruction en parvenant à identifier plus précisément les contours et les couleurs des objets. Néanmoins, il présente quelques difficultés avec certaines classes, comme la confusion entre les chaussures de tennis

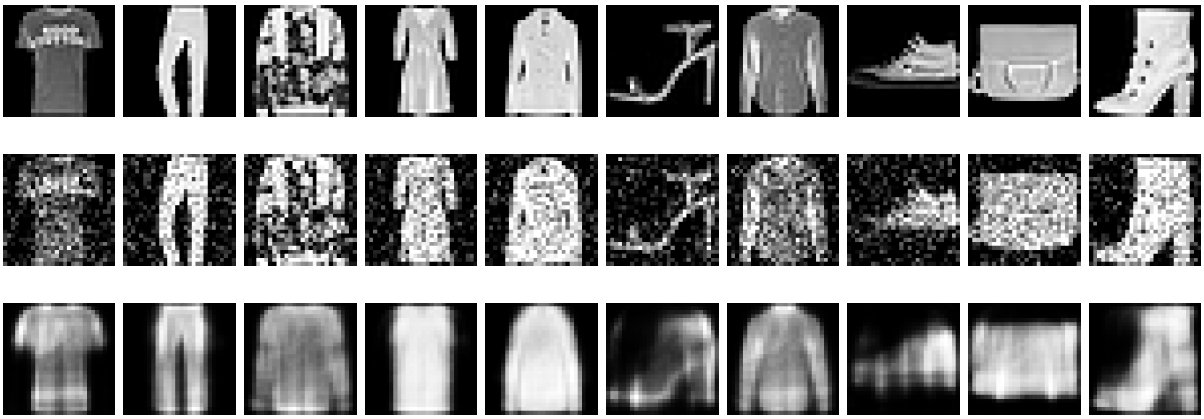


FIGURE 19 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 1

(classe 7) et les talons, ce qui peut être attribué à une similarité visuelle entre ces deux catégories.

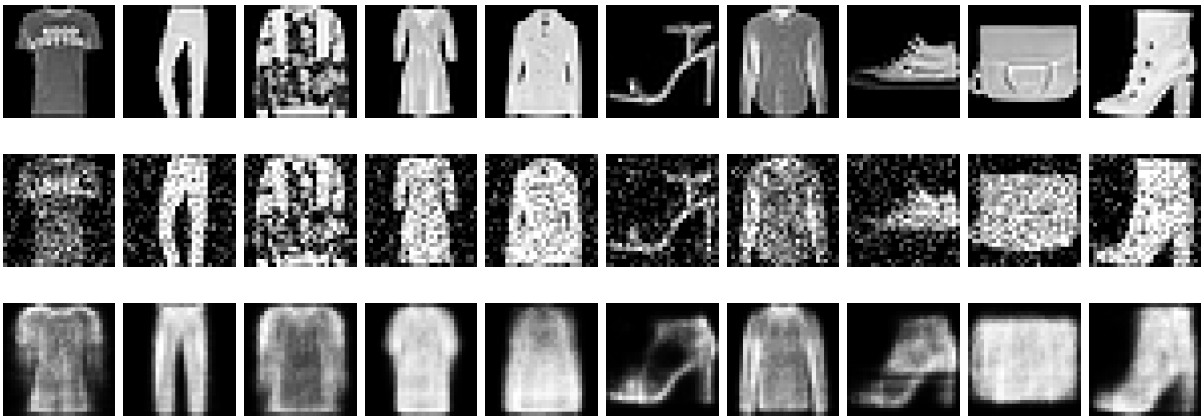


FIGURE 20 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 2

Le troisième modèle, représenté dans la figure 21, obtient les meilleurs résultats de reconstruction. Ce modèle parvient à supprimer efficacement le bruit et à retrouver les informations essentielles des images d'origine, offrant ainsi une meilleure fidélité visuelle dans la reconstruction et démontrant sa capacité à apprendre des représentations plus riches et détaillées.

Les figures 22, 23, 24, 25 présentent les résultats d'une expérience où nous avons introduit différents niveaux de bruit gaussien pour évaluer la robustesse des modèles de reconstruction. Les images débruitées montrent souvent une perte de fidélité aux images originales, avec une tendance à reconnaître des traits caractéristiques des vestes ou des t-shirts plutôt que de préserver les détails spécifiques de chaque classe. Ces résultats soulignent les limites des modèles lorsqu'ils sont confrontés à des niveaux de bruit élevés et mettent en évidence la difficulté de supprimer efficacement le bruit tout en conservant les caractéristiques distinctives des images d'origine.

Une amélioration significative de l'auto-encodeur peut être obtenue en utilisant des réseaux de neurones convolutionnels qui nécessitent des couches de convolutions transposées pour "déconvolutionner" les données ou des fonctions d'upsampling. Une approche plus simple serait d'utiliser une convolution en 2D. Le *padding* serait également nécessaire pour garder la taille d'origine des données.

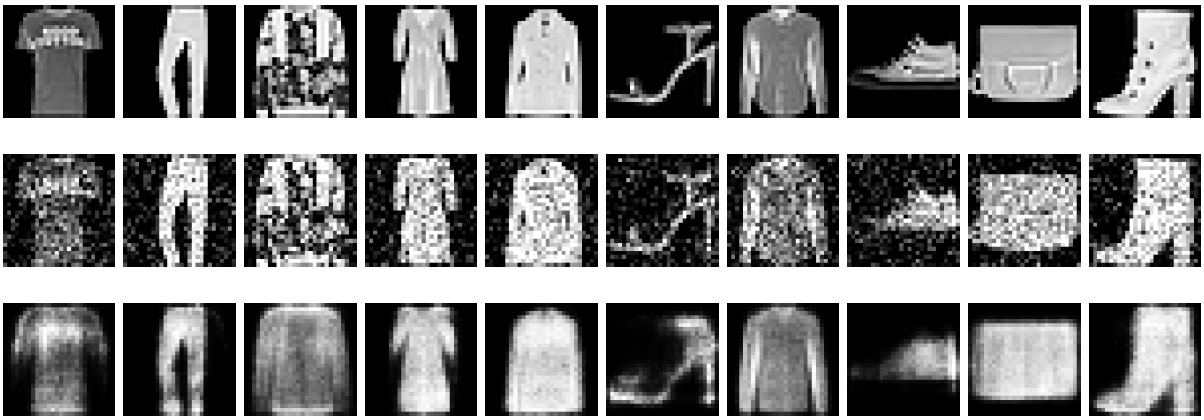


FIGURE 21 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 3

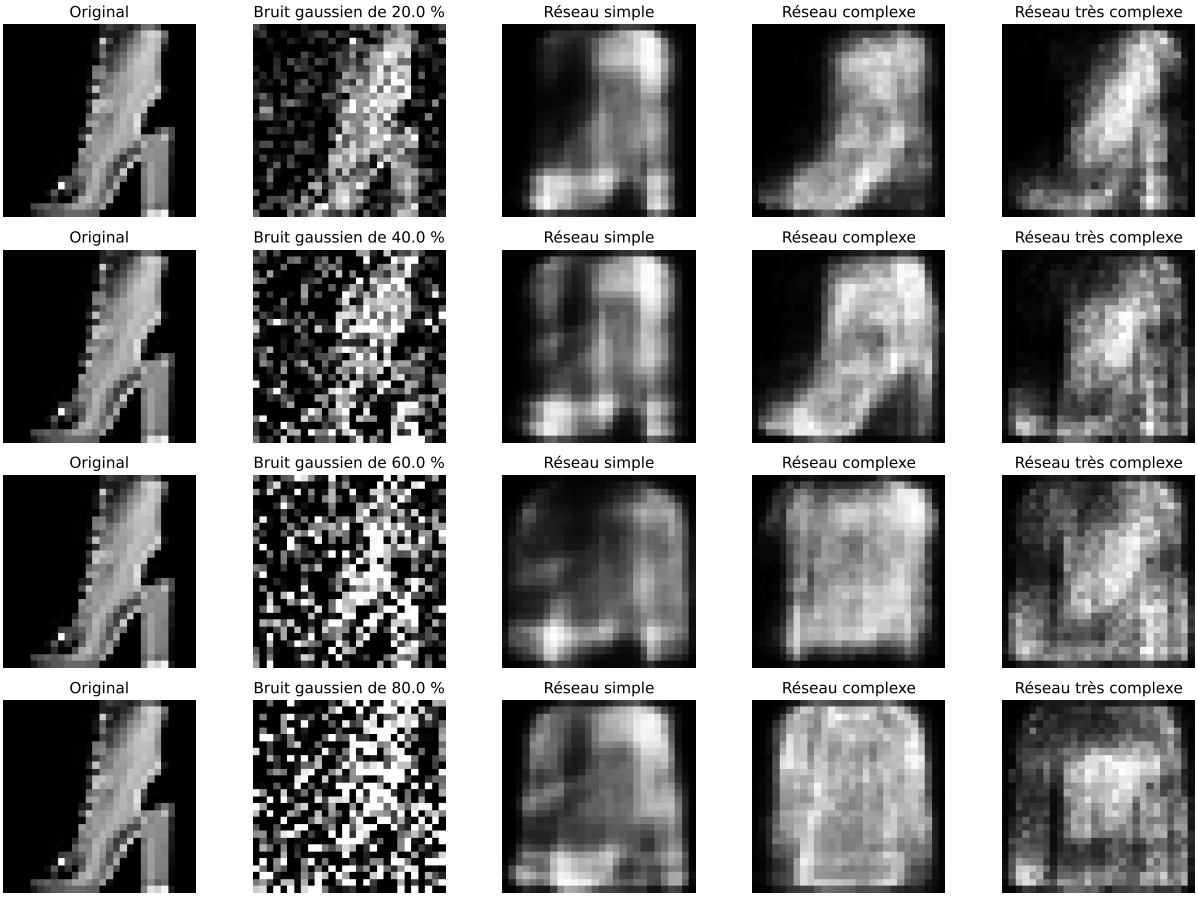


FIGURE 22 – Reconstruction d'une image bruitée de chaussure

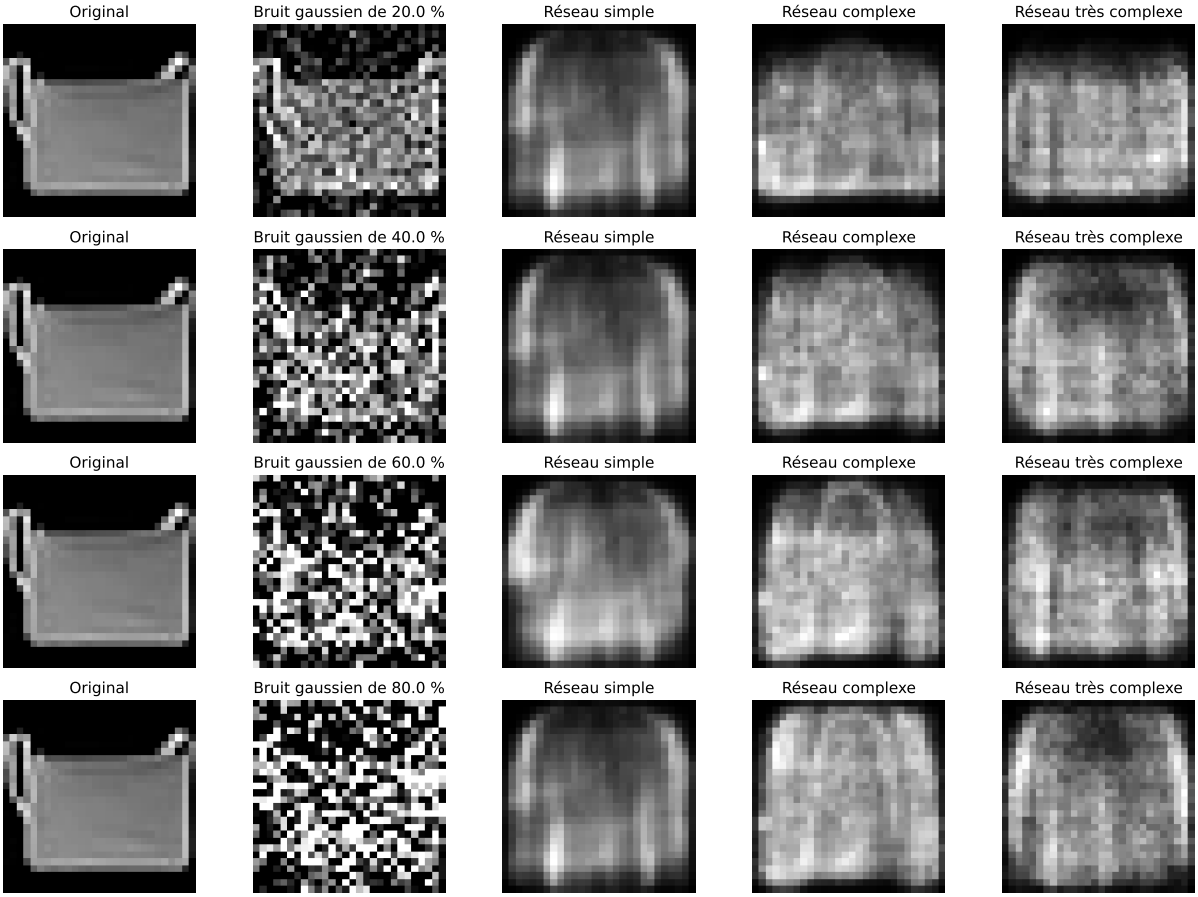


FIGURE 23 – Reconstruction d'une image bruitée de sac

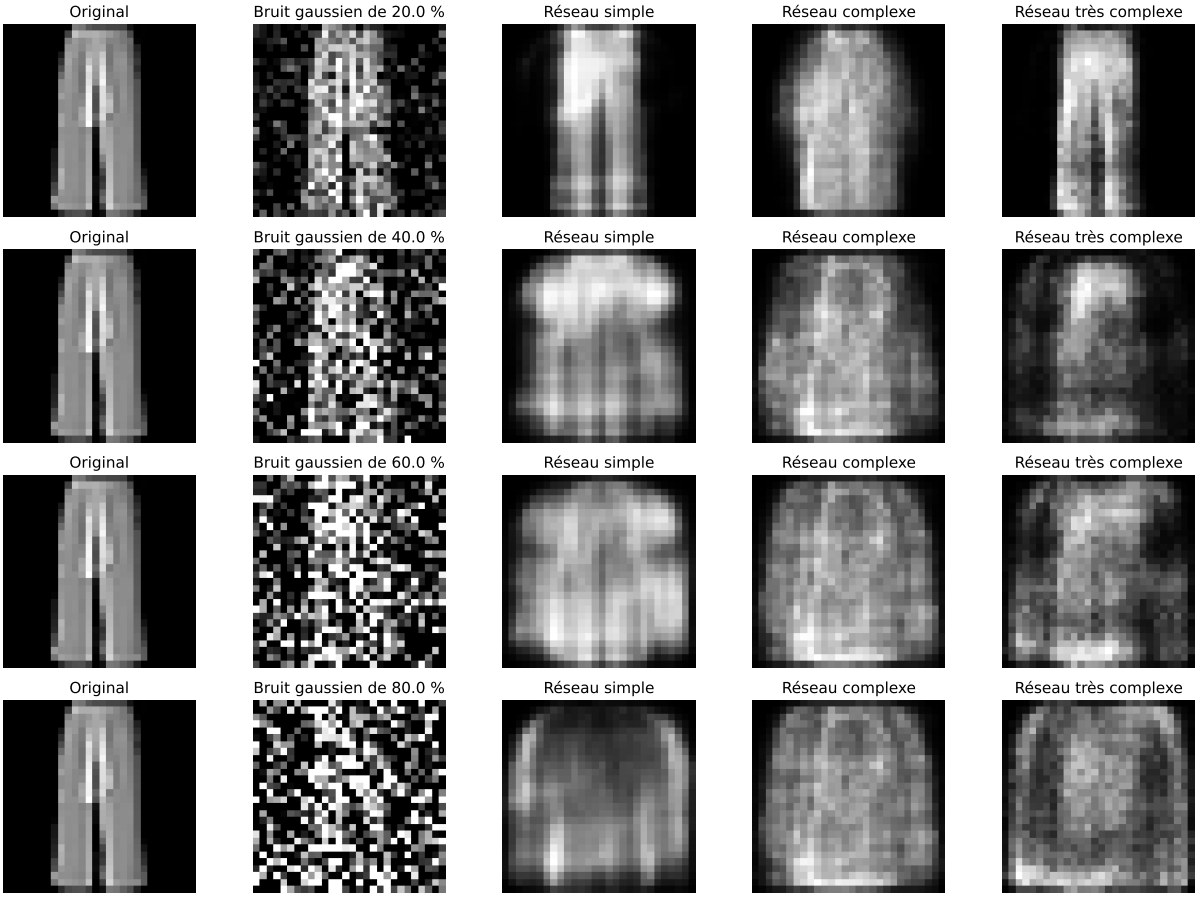


FIGURE 24 – Reconstruction d'une image bruitée de pantalon

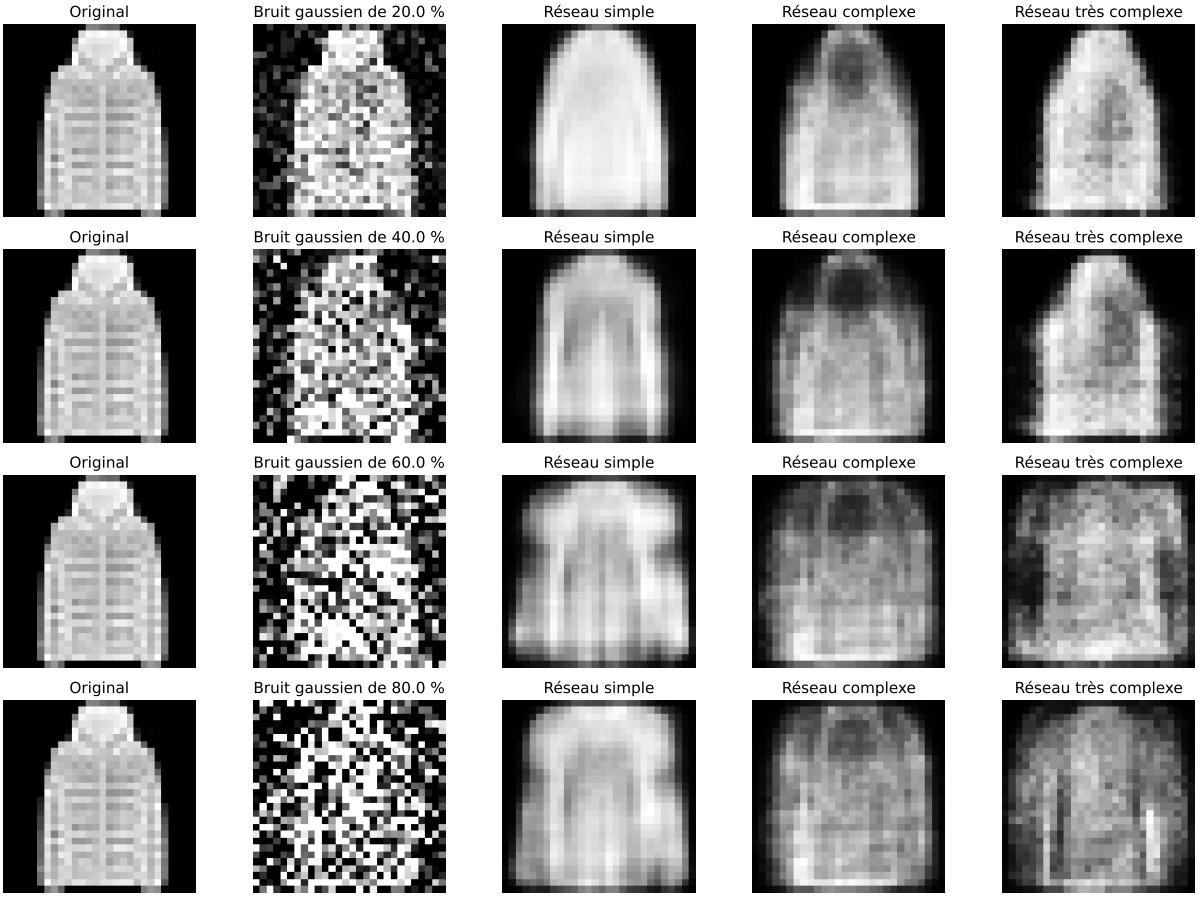


FIGURE 25 – Reconstruction d'une image bruitée de manteau

Nous avons tenté, sans succés, d'utiliser un auto-encodeur avec une architecture utilisant des convolutions 1D de la forme : `Conv1D(3, 1, 32)` → `ReLU()` → `MaxPool1D(2, 1)` → `Conv1D(3, 32, 16)` → `ReLU()` → `MaxPool1D(2, 1)` → `Conv1D(3, 16, 16)` → `ReLU()` → `Conv1D(2, 16, 32)` → `ReLU()` → `Conv1D(2, stride=2)` → `Sigmoid()`.

4 Conclusion

L'implémentation de cette bibliothèque de deep learning a été une expérience enrichissante, nous offrant une compréhension approfondie des concepts algébriques et de la beauté de la rétropropagation. Bien que nous ayons exploré différentes tâches telles que la reconstruction d'images et la génération de données, nous n'avons malheureusement pas eu le temps d'implémenter des fonctionnalités plus avancées telles que le transfert de style ou la traduction avec un auto-encodeur.

Cependant, nous reconnaissons l'importance de l'optimisation matérielle, telle que l'utilisation de GPU et la compilation pour améliorer les performances de nos modèles. Cela aurait pu accélérer les calculs et nous permettre d'explorer des architectures plus complexes et des ensembles de données plus volumineux.

En conclusion, bien que notre exploration ait été limitée par des contraintes de temps, cette expérience nous a permis de mieux comprendre les fondements du deep learning et de réaliser l'importance des choix d'architecture, d'optimisation matérielle et de l'implémentation de fonctionnalités avancées pour atteindre de meilleures performances et explorer des domaines plus vastes.