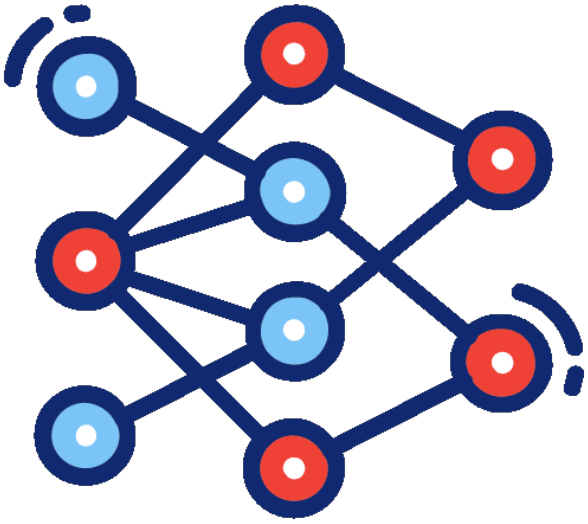
ML - Projet

Réseau de neurones from scratch

Charles Vin, Aymeric Delefosse



DAC | 2022-2023

SORBONNE UNIVERSITÉ

Table des matières

1 Introduction **2**

2 Implémentation **2**

3 Expérimentation **2**

3.1 Classification 2

3.1.1 Effet de l'initialisation des paramètres 2

3.1.2 Effet du taux d'apprentissage (*learning rate*) 4

3.1.3 Effet des fonctions d'activation 4

3.2 Classification et convolution 6

3.2.1 Effet de la taille des *batch* 6

3.2.2 Effet de la taille du kernel 7

3.2.3 Effet du nombre de *feature maps* 7

3.2.4 Diverses architectures 7

3.3 Auto-encodeur 9

3.3.1 Reconstruction 9

3.3.2 Organisation de l'espace latent 11

3.3.3 Débruitage 12

4 Conclusion **12**

1 Introduction

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 : `MNIST`) 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 : très simpliste, tous les paramètres sont initialisés à 1 ;

— Initialisation à 0 : très simpliste, 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 : $W, B \sim \mathcal{N}(0, \sqrt{\frac{2}{\text{input+output}}})$;
  — Loi uniforme : $W, B \sim \mathcal{U}(-\sqrt{\frac{6}{\text{input+output}}}, \sqrt{\frac{6}{\text{input+output}}})$.
— 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 déterminer en pratique l'effet de l'initialisation des paramètres, nous avons mis en place un réseau très simple qui classifie le jeu de données *fashion-mnist*. Celui-ci est composé d'une couche linéaire prenant en entrée l'image entière et classifiant sur dix neurones en sortie. On représentera par la suite les réseaux sous la forme suivante `Linear(784, 10)` → `Sigmoid()`.

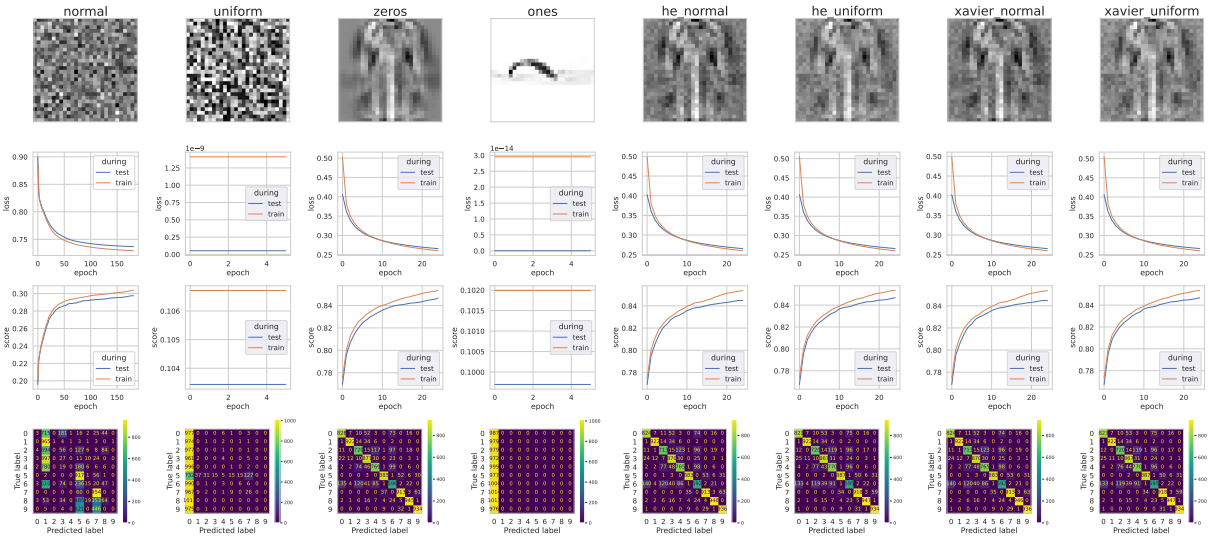


FIGURE 1 – Représentation des paramètres, d'une matrice de confusion, de l'évolution du coût et du score par époque en fonction de l'initialisation des paramètres

La figure 1 représente les résultats de cette expérimentation. On constate des difficultés d'apprentissage sur certaines initialisations. L'initialisation normale tombe dans un minimum local (200 époques). Les initialisations uniforme et à 1 font exploser le gradient, empêchant tout apprentissage.

Cette expérience a montrer l'impact important que peut avoir l'initialisation des paramètres. L'initialisation normale pourrait certainement être fixée en utilisant un autre algorithme d'optimisation, tel qu'Adam.

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.

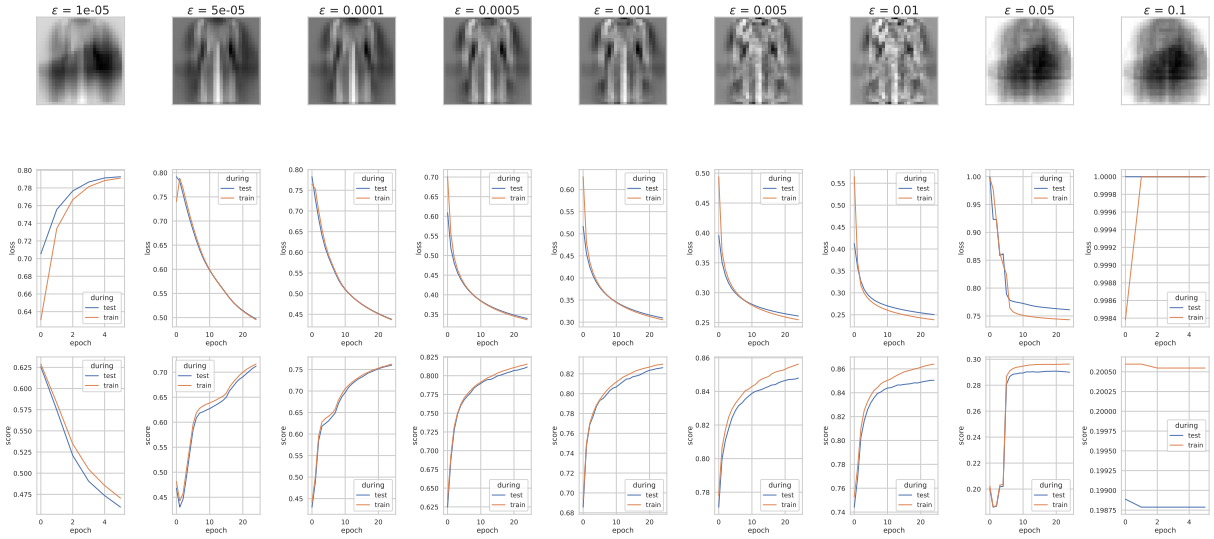


FIGURE 2 – Représentation des paramètres, de l'évolution du coût et du score par époque en fonction du taux d'apprentissage ϵ

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

La fonction d'activation joue un rôle essentiel dans les réseaux de neurones : elle introduit une non-linéarité dans les activations du réseau, ce qui permet au modèle d'apprendre des représentations non linéaires complexes des données.

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* et empêcher la rétropropagation du gradient ;

— 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.

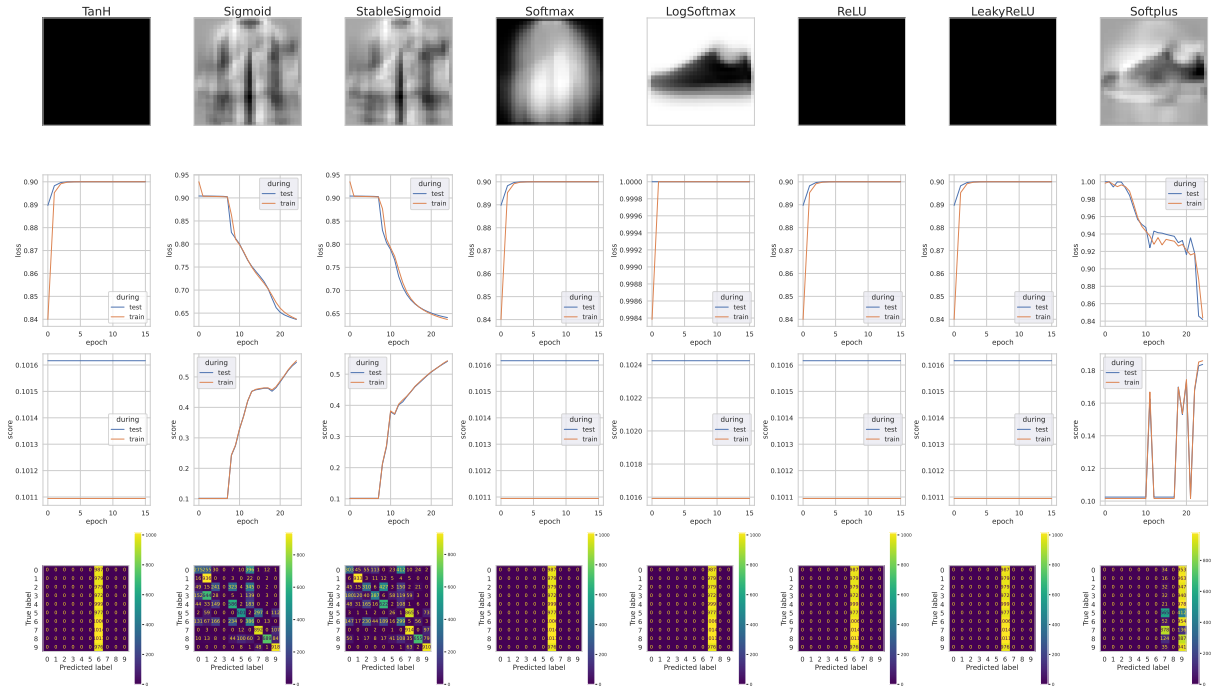


FIGURE 3 – Représentation des paramètres, de l'évolution du coût et du score par époque en fonction du taux d'apprentissage ϵ

3.2 Classification et convolution

Même avec un multi-layer perceptron, on parvient à obtenir des performances correctes sur des images. Et si on utilisait des réseaux de neurones convolutifs ? L'intérêt de la convolution étant de capturer les motifs locaux, de réduire le nombre de paramètres à apprendre, d'obtenir une certaine invariance aux translations et de réduire la dimensionnalité des données, c'est un module particulièrement adapté aux images, aux séquences temporelles ou signaux.

La convolution en une dimension est implémentée et fonctionnelle, ainsi que les deux couches de pooling (Max et Average). Elles ont toutes été implémentées en utilisant le moins de boucles possibles afin d'avoir une implémentation la plus performante possible (seulement une boucle présente dans la convolution, sur la taille du kernel) grâce aux fonctions avancées de `numpy`. Le raisonnement a été 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 le cadre de la reconnaissance de chiffres.

Sur une taille de batch fixée à 32, un nombre d'époques à 50 et un early-stopping à 5, répété 10 fois, on obtient un score en apprentissage de $99.36 \pm 0.0017\%$ et un score en évaluation de $97.08 \pm 0.0016\%$, en 31 ± 6 époques.

Par ailleurs, l'initialisation est très importante dans les couches convolutionnelles, par exemple une initialisation des poids à zéros ne marche pas (figure 4). Ici, les convolutions sont initialisées grâce à une initialisation de Xavier selon une loi normale et les modules linéaires grâce à une itialisation de He normale.

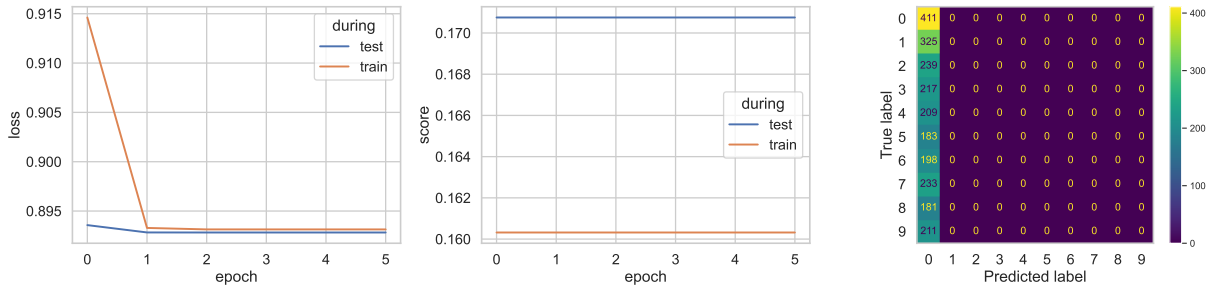


FIGURE 4 – Impact de l'initialisation sur un réseau de neurones convolutionnel

3.2.1 Effet de la taille des *batch*

Le nombre de batchs est un hyper-paramètre assez important de notre modèle. Ainsi, on obtient des meilleurs performances en apprenant sur des batchs plus petit, ce qui est courant et connu en *deep learning*, mais s'apparente actuellement à du sur-apprentissage, même si l'on obtient des bons résultats en évaluation. De plus, plus la taille du batch sera petite, plus long sera l'apprentissage. Avec des batchs plus grand, le temps d'apprentissage sera très réduit mais les performances également ; sans compter que cela affecte également les instabilités numériques (explosion du gradient).

3.2.2 Effet de la taille du kernel

En faisant varier la taille du kernel, on se permet de capturer des motifs ou des informations de différentes échelles spatiales dans les données (ou non). Par exemple, des kernels de petite taille peuvent être efficaces pour détecter des détails fins, tandis que des kernels de plus grande taille peuvent être plus adaptés pour détecter des caractéristiques plus globales. Sur les données des chiffres manuscrits, les chiffres font partie intégrante de l'image, mais l'image est petite.

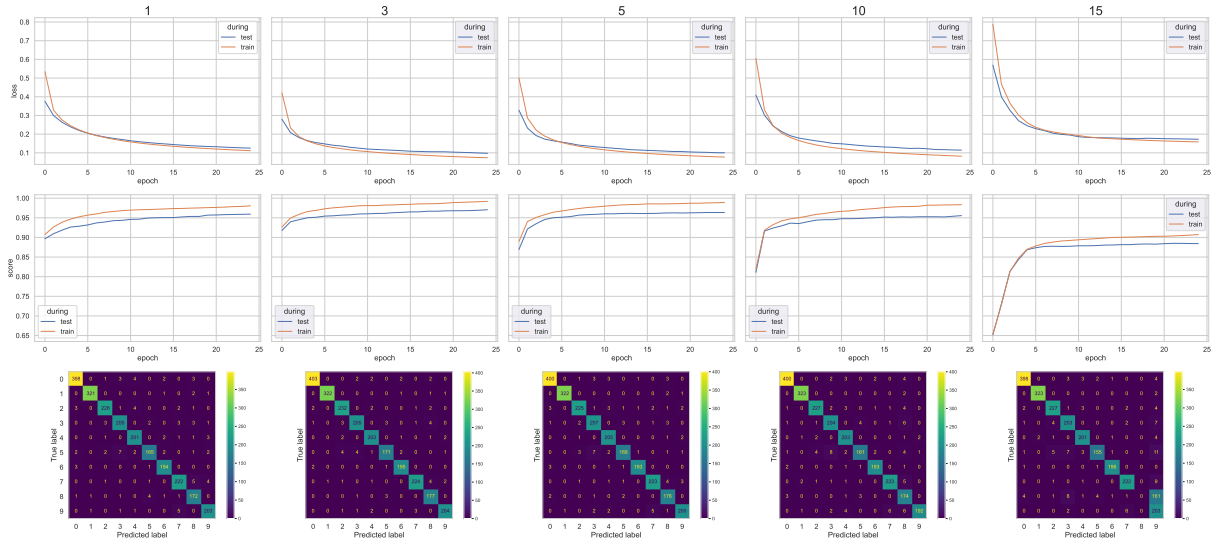


FIGURE 5 – Impact de la taille du kernel pour un réseau de neurones convolutionnel

3.2.3 Effet du nombre de *feature maps*

Le nombre de feature maps dans une couche de convolution influence la capacité du modèle à capturer des caractéristiques et des motifs complexes, améliorant ainsi la représentation des données et la capacité de généralisation. Cependant, il faut trouver un équilibre en tenant compte des contraintes de ressources et des exigences spécifiques du problème.

3.2.4 Diverses architectures

Diverses architectures ont également été testées, pour voir la robustesse (ou non) de la convolution et sa capacité à surapprendre ou non, en particulier en agrandissant la profondeur des réseaux. Quelques variantes ont été utilisées en appliquant une fonction d'activation directement sur la convolution, ce qui est courant de faire, sans grand résultats. Finalement, un plus gros réseau convonlutionnel n'est pas synonyme de meilleure classification.

Deux couches convolutionnelles `Conv1D(3, 1, 32, 1)` → `ReLU()` → `MaxPool1D(2, 2)` → `Conv1D(3, 32, 32)` → `ReLU()` → `MaxPool1D(2, 2)` → `Flatten()` → `Linear(1984, 10)`

Les résultats sont sur la figure 7 : les résultats ne sont pas catastrophiques, le réseau commence à apprendre, avant de subitement désapprendre. Les performances sont bien en-deça de celui du réseau précédent, avoir une couche linéaire n'est sûrement pas suffisante pour correctement classifier nos données.

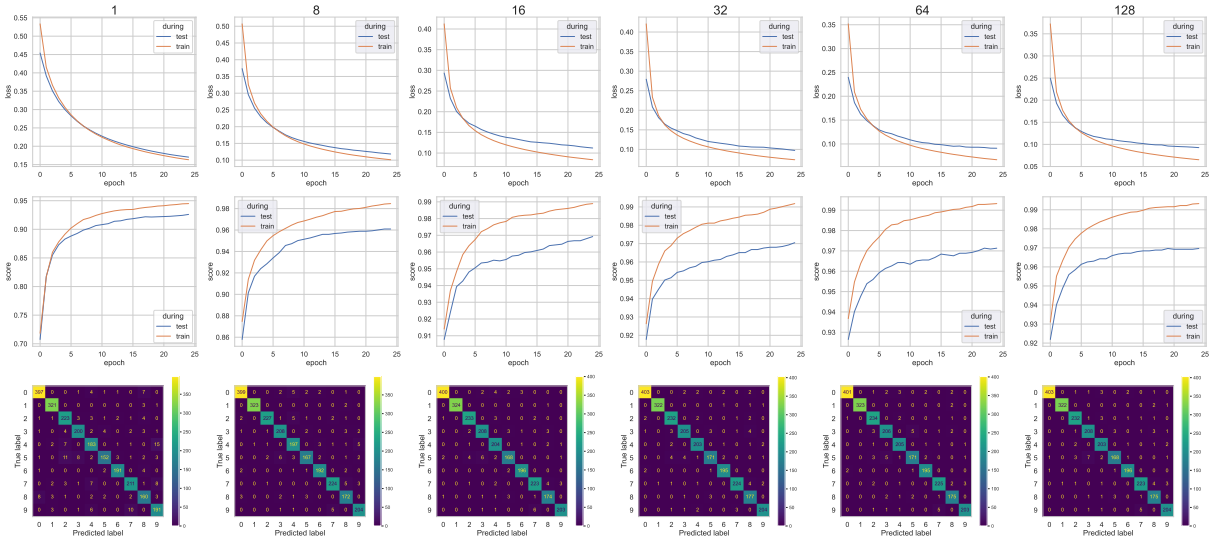


FIGURE 6 – Impact du nombre de *feature maps* pour un réseau de neurones convolutionnel

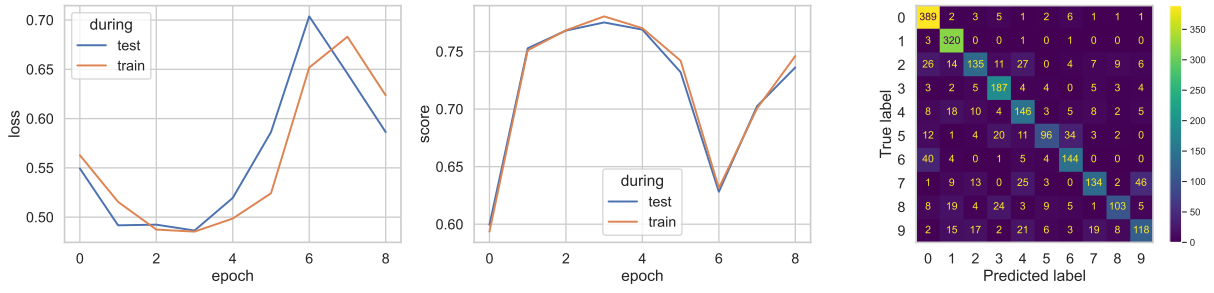


FIGURE 7 – Réseau de neurones convolutionnel à 2 couches

Quatre couches convolutionnelles `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)` → `Flatten()` → `Linear(512, 10)`

Les résultats sont sur la figure 8 : ici, il parvient mieux à apprendre que le réseau précédent, mais peine à obtenir des performances correctes. On voit par ailleurs que la classe 2 est celle qui est la moins bien classée et est la cause de ce manque de performance : la taille du kernel dans le max pooling est surement trop élevé pour garder un nombre de features significatifs et donc identifier correctement cette classe.

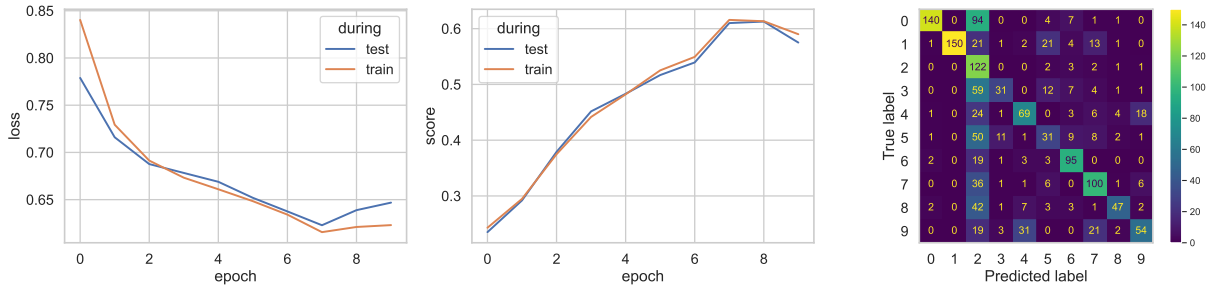


FIGURE 8 – Réseau de neurones convolutionnel à 4 couches

3.3 Auto-encodeur

Nous nous sommes également attelés à des tâches de reconstruction ou de génération de données à l'aide d'architecture encodeur-décodeur (ou auto-encodeur).

3.3.1 Reconstruction

Dans notre cas, on entraîne le modèle à reconstruire les images d'origine à partir de la représentation latente. Pour cela, l'encodeur prend les images d'entrée et les transforme en une représentation latente de dimension réduite. Ensuite, le décodeur prend cette représentation latente et essaie de générer des images qui sont similaires aux images d'origine. L'objectif de cette expérience est de voir comment le modèle est capable de capturer les caractéristiques distinctives des images et de reconstruire des images de qualité. On peut évaluer la performance du modèle 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 Ici, trois architecture ont été testé :

1. `Linear(784, 64)` → `TanH()` → `décodeur`
2. `Linear(784, 256)` → `TanH()` → `Linear(256, 64)` → `TanH()` → `décodeur`

3. `Linear(784, 512)` → `TanH()` → `Linear(512, 256)` → `TanH()` → `Linear(256, 128)` → `TanH()` → `Linear(128, 64)` → `TanH()` → `décodeur`

Avec `décodeur` représentant l'architecture transposé du réseau mais avec une `Sigmoid()` en sortie.

Ces trois architectures, plus ou moins complexes, ont toutes mené rapidement à une stagnation du coût d'évalutation durant l'entrainement (voir figure 9). Cette valeur de stagnation varie légérement en fonction de la complexité du réseau. Dans la figure 10, on remarque effectivement que la reconstruction est plus précise en fonction de la complexité du réseau.

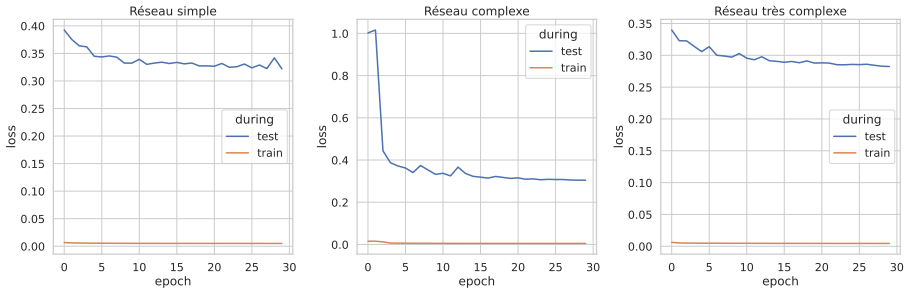


FIGURE 9 – Valeur de la `BCELoss()` pour chaque réseau sur 30 epoch

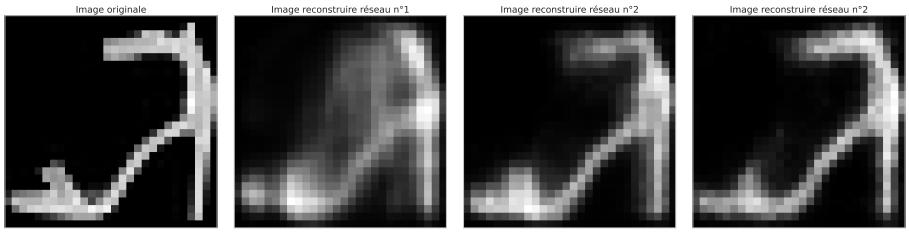


FIGURE 10 – Exemple de reconstruction en fonction de la complexité du réseau

Pour le réseau n°1, une étude de l'influence de la fonction d'activation placé avant l'espace latent a été réalisé. Le réseau a donc la forme suivante : `Linear(784, 64)` → `acc_fct()` → `décodeur`. Les résultats sont présenté dans la figure 11.

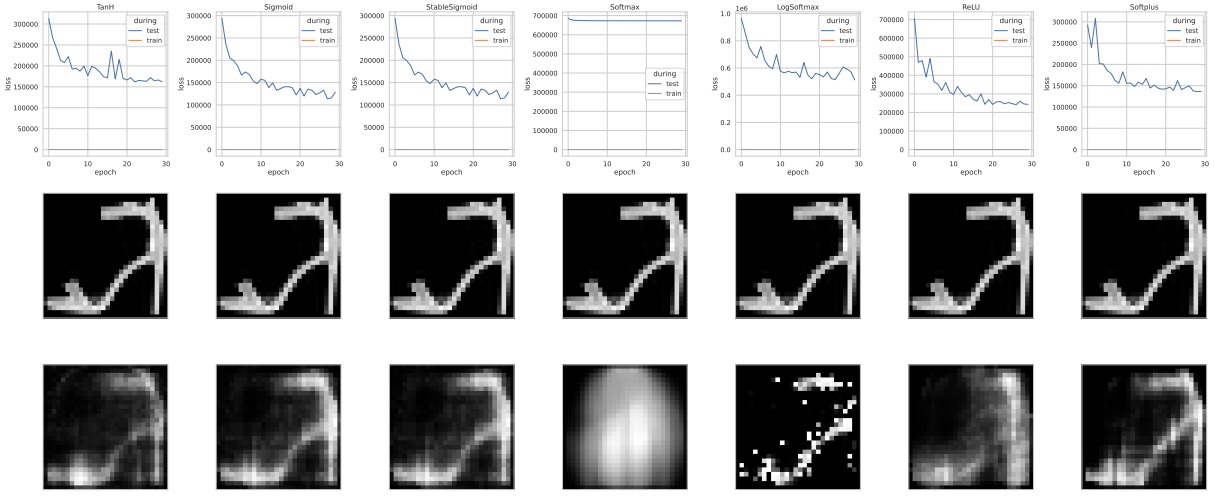


FIGURE 11 – Fonction de cout et reconstruction pour une architecture type réseau n°1

On peut voir que le `Softmax()` et le `LogSoftmax()` ont des grosses difficultées d'apprentissage. On remarque également que la `ReLU` apprends beaucoup moins vite que les autres fonction d'activation.

On s'est aussi amuser à visualiser la reconstruction de l'image au fur et à mesure des époques d'apprentissage dans les figures 12 et 13.

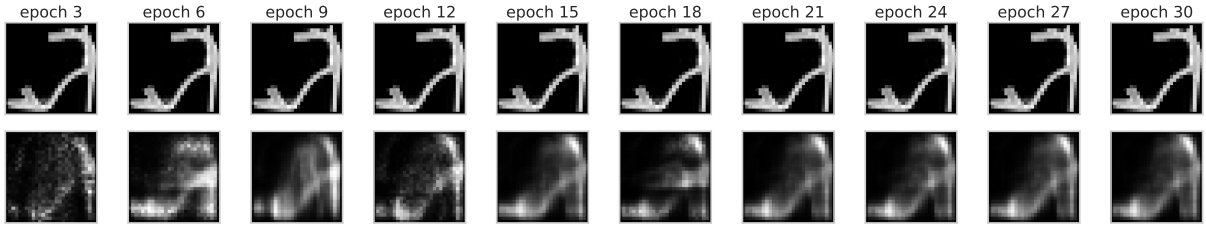


FIGURE 12 – Reconstruction par epoch pour le model n°1

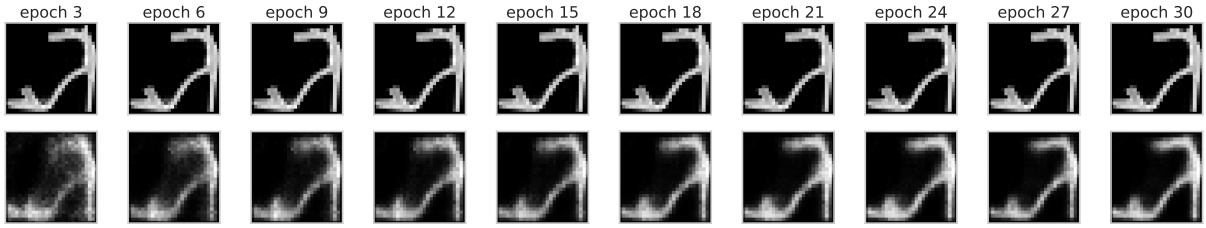


FIGURE 13 – Reconstruction par epoch pour le model n°3

3.3.2 Organisation de l'espace latent

Afin de visualiser l'espace latent de 64 dimension, nous avons utiliser la t-SNE afin de le réduire à deux dimensions visualisables. Ce travail a été fait sur les deux jeu de données et sur chacune des séparations entrainement et de test, en utilisant l'auto-encoder le riche en paramètre. Les t-SNE sur le jeu de test sont assez similaire à celle pour le train, nous ne montrerons ici que celle des données d'entrainement. Ces visualisations sont dans les figures 14 et 15.

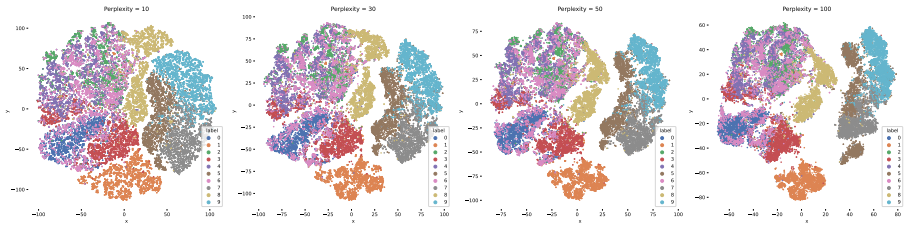


FIGURE 14 – t-SNE sur l'espace latent du modèle n°3 pour *fashion-mnist* en utilisant les données d'entrainement

Les résultats sont assez intéréssant. En effet on peut voir que les classes zéro, deux, quatre et six n'ont pas de vraiment de cluster, représentant respectivement les T-shirt/top, les Pullover, les mateaux et les chemises. Ces classe étant plus dicifile à distinguer entre elle qu'un pantalon (classe une) ou d'une chaussure (classe 5), elle sont mixé dans l'espace latent.

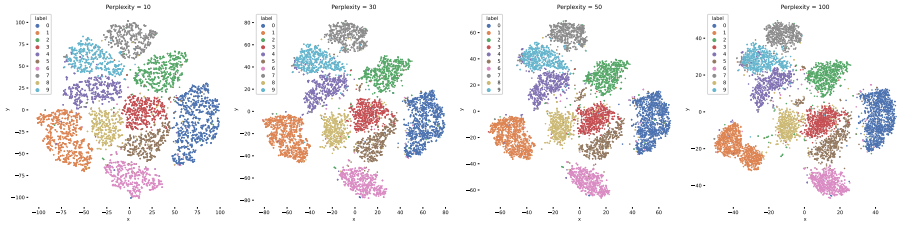


FIGURE 15 – t-SNE sur l'espace latent d'un plus gros modèle pour *usps* en utilisant les données d'entrainement

3.3.3 Débruitage

Ici, l'objectif est de reconstruire des images d'origine à partir de versions bruitées, qui est similaire à un travail de reconstruction. Elle montre comment le modèle est capable de supprimer le bruit et de récupérer les informations essentielles des images d'origine. Les trois modèles appris précédemment ont servi à cette tâche.

Avec un bruit gaussien de 20 %, sur une image de chaque classe.

Le réseau le plus simple, sur la figure 16 s'en sort plutôt bien. Bien que l'on perd tout de même en information, notamment au niveau des contours, on parvient à retrouver la forme générale de chaque classe tout en restant fidèle au niveau de gris.

Le réseau légérement plus complexe, sur la figure 17 s'en sort bien également. Il arrive mieux à identifier les contours et les couleurs, pas pour toutes, notamment pour la chaussure de tennis (classe 7), qu'il confond avec des talons.

Le réseau plus complexe, sur la figure 18 s'en sort le mieux.

Sur les figures ??, ??, ??, ??, on fait varier le pourcentage de bruit gaussien ajouté aux données pour comparer la robustesse des modèles. Ainsi, les images débruitées se transforment en tout sauf quelque chose d'original : on reconnait bien souvent une image issue de la classe des vestes ou des t-shirts.

Une grosse amélioration de l'auto-encodeur peut se faire avec des réseaux de neurones convolutionnels mais requièrent des couches de convolutions transposées, qui "déconvolutionnent" des données, ou encore des fonctions d'upsampling. Cela serait également beaucoup plus simple avec une convolution en 2D. Nous avons tenté, en vain, d'utiliser un auto-encodeur avec une architecture ressemblant à : `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()`. Le *padding* serait également nécessaire pour garder la taille d'origine des données.

4 Conclusion

Transfert de style aurait été cool mais besoin de conv2d et on avait pas le temps mashallah

traduction

opti gpu et compilation

reconstruct_noisy_all_small.pdf

FIGURE 16 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 1

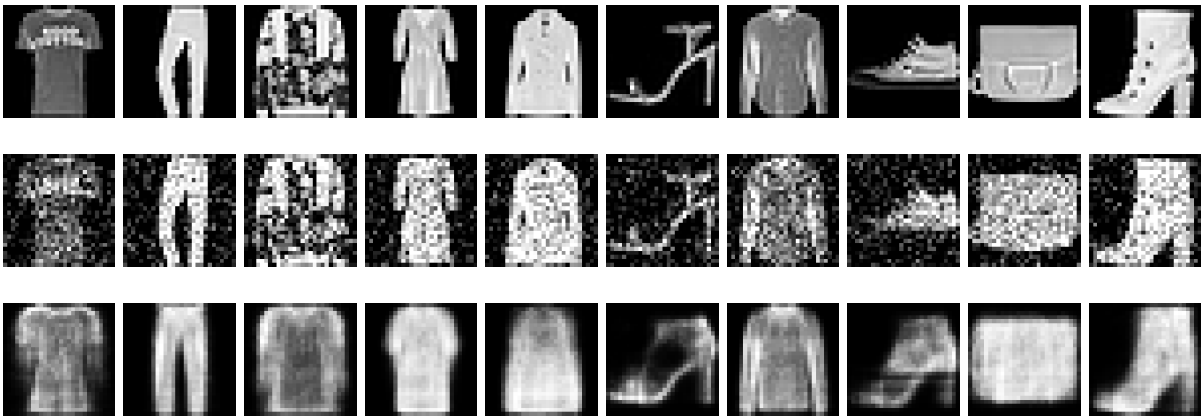


FIGURE 17 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 2

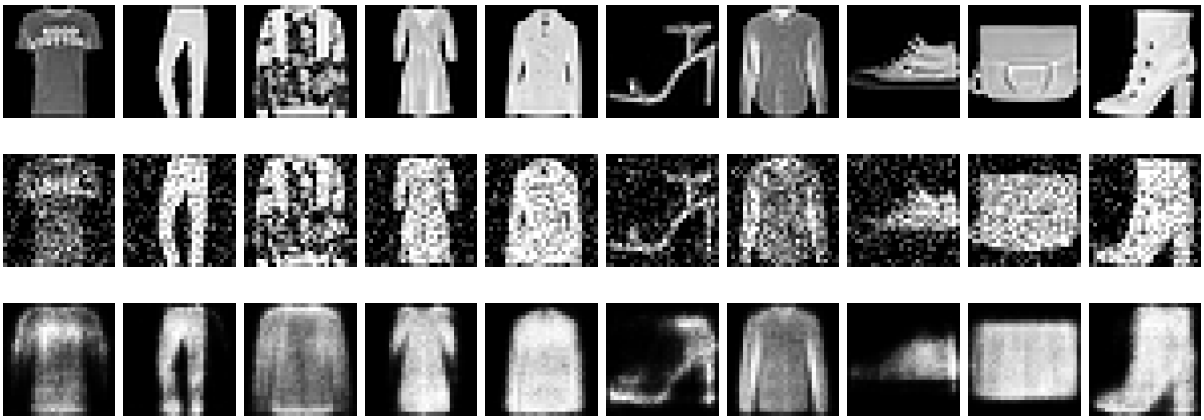


FIGURE 18 – Reconstruction d'une image bruitée de chaque classe pour le réseau n° 3

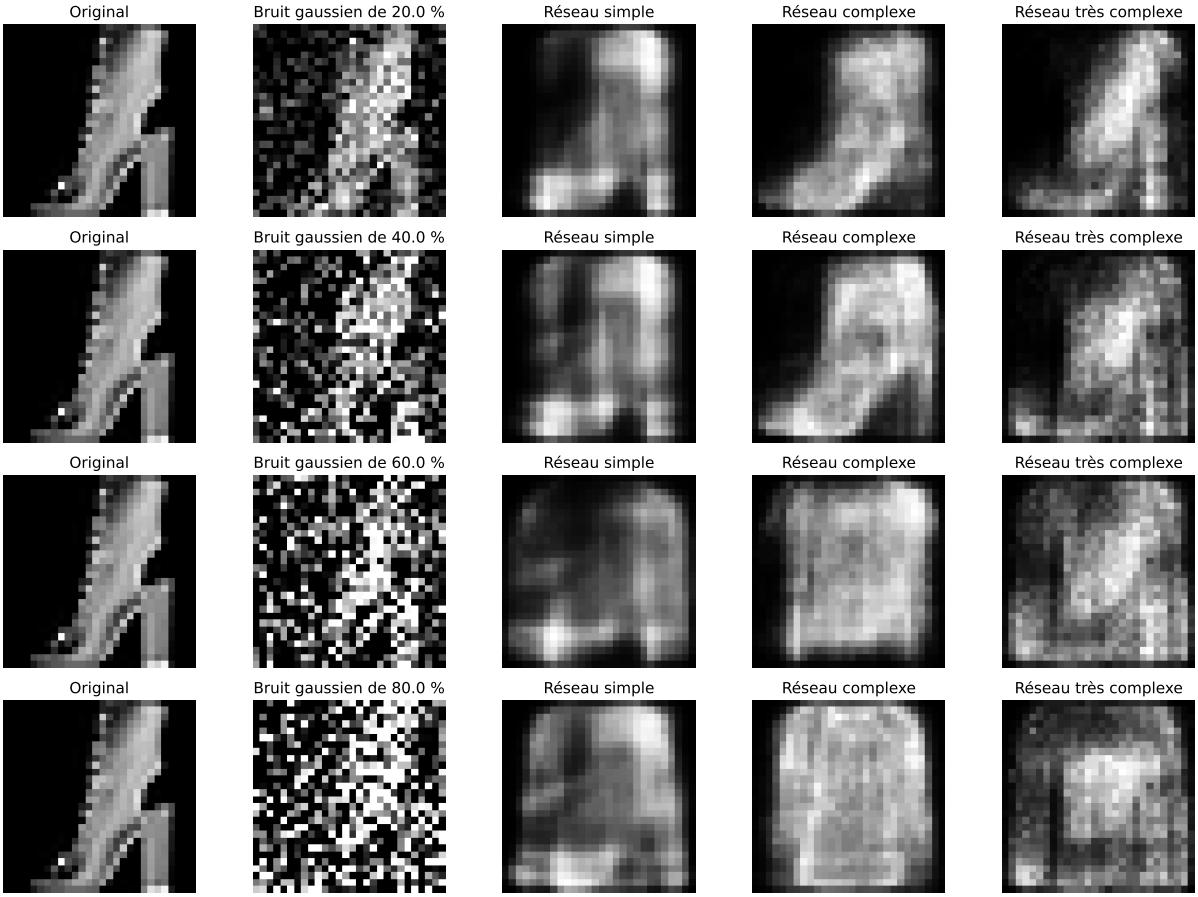


FIGURE 19 – Reconstruction d'une image bruitée de chaussure

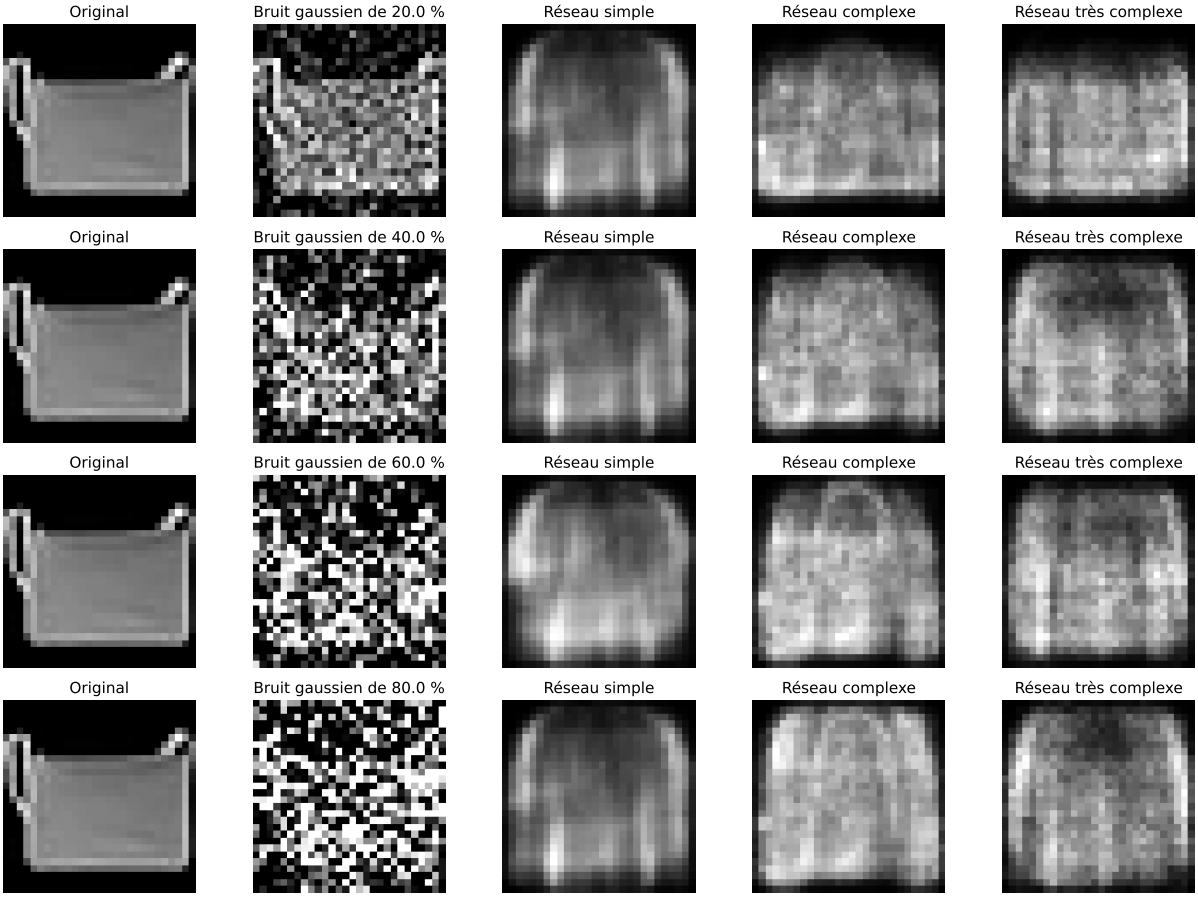


FIGURE 20 – Reconstruction d'une image bruitée de sac

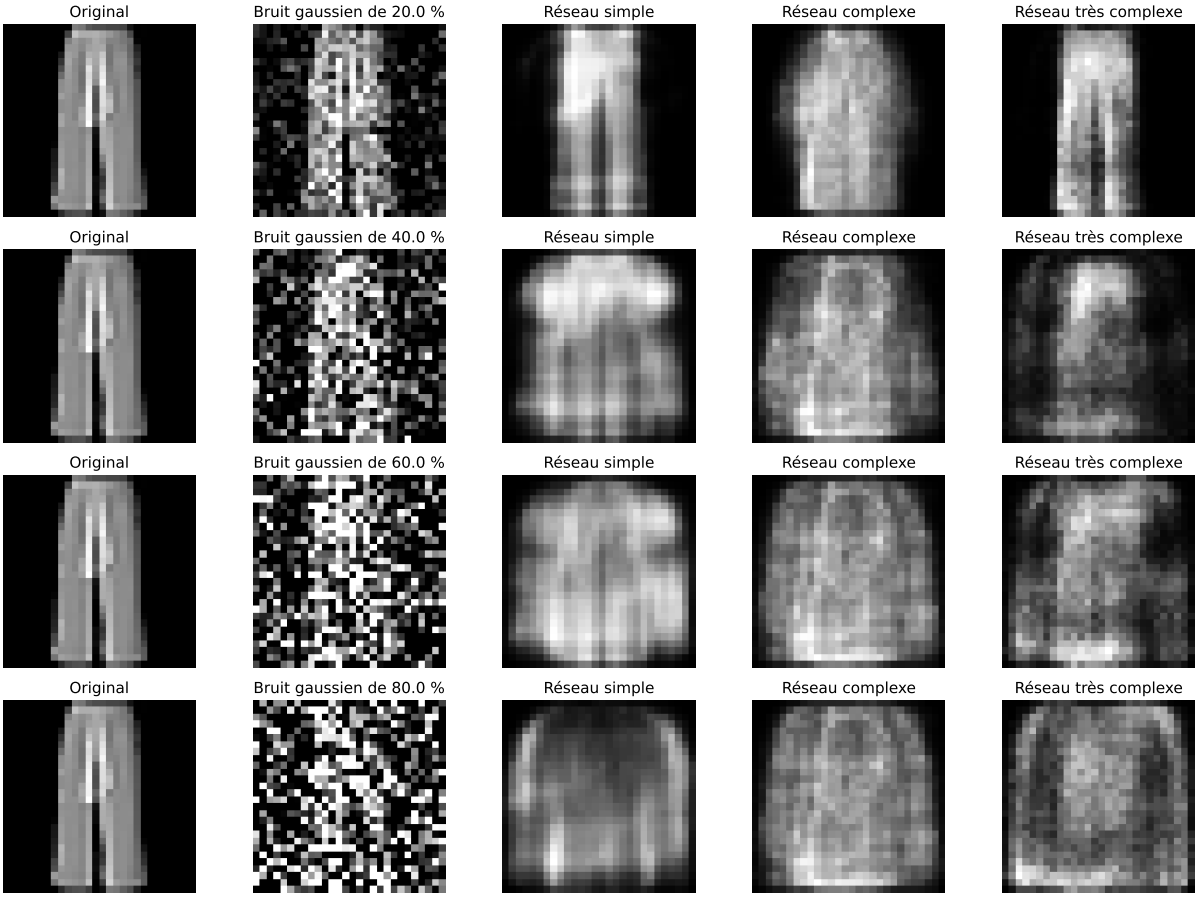


FIGURE 21 – Reconstruction d'une image bruitée de pantalon

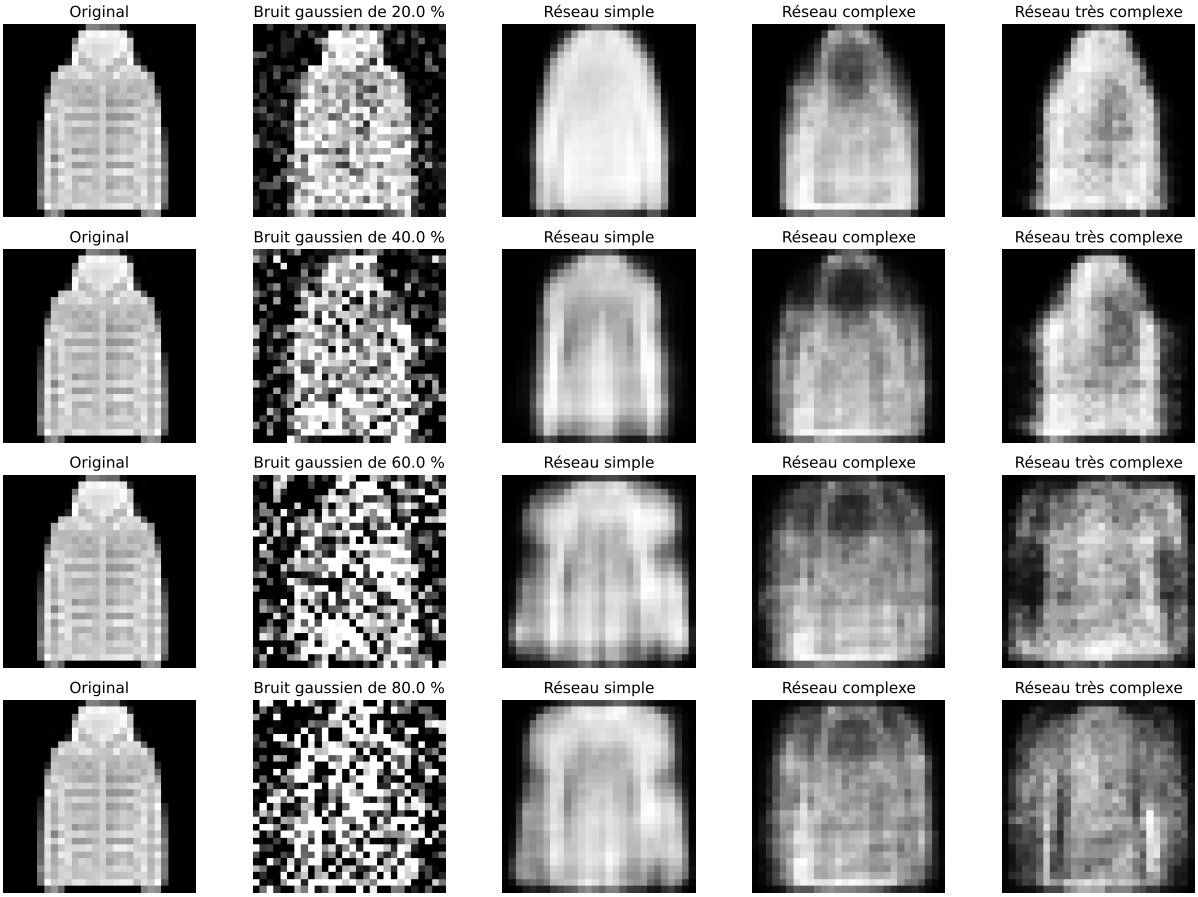


FIGURE 22 – Reconstruction d'une image bruitée de manteau