

Projet ML - Réseaux de neurones

DAUTRICHE Maxime, WU Lin-Jie

May 2025

Table des matières

1	Introduction	2
2	Modules	2
2.1	Module linéaire	2
2.2	Module non-linéaire	2
2.3	Sequentiel	2
2.4	Multi-classe	3
2.5	Tests	3
3	Auto-Encoder	3
4	Analyse expérimentale des auto-encoders	4
4.1	Paramètres des modèles	4
4.2	Comparaison des performances	5
4.3	Utilisations possibles des auto-encoders	13
4.3.1	Débruitage	14
4.3.2	Reconstruction de parties manquantes	17
5	Conclusion	19

1 Introduction

Nous avons réalisé, au cours de ce projet, une implémentation en python d'un réseau de neurones modulaire. L'ensemble des classes réalisées et nos analyses expérimentales seront présentées dans les sections suivantes.

2 Modules

Un réseau de neurones est représenté par une agrégation de modules. Chaque module est implémenté dans un fichier différent, et hérite de la classe abstraite *Module* implémentée dans le fichier *neural_network.py*. Nous avons également réalisé une classe abstraite *Loss*, permettant de représenter une fonction de coût.

2.1 Module linéaire

Le module linéaire correspond à une simple couche de neurones, représentés par des perceptrons. Nous avons implémenté la classe *Linear* dans le fichier *linear_module.py*. Le constructeur de la classe prend en paramètres la dimension des entrées ainsi que la dimension des sorties.

Nous avons également implémenté dans le même fichier la classe *MSELoss*, correspondant à un coût aux moindres carrés. Le constructeur ne prend aucun argument.

2.2 Module non-linéaire

Les deux classes permettant de rendre les réseaux non-linéaires sont implémentées dans le fichier *non_linear_module.py*.

La classe *TanH* permet d'appliquer la fonction tangente hyperbolique pour transformer les entrées, renvoyant pour chaque dimension une valeur entre -1 et 1. La dimension des sorties est la même que celles des entrées, le constructeur ne prend donc aucun argument. De même, la classe *Sigmoïde* permet d'appliquer la fonction sigmoïde pour transformer les entrées, renvoyant pour chaque dimension une valeur entre 0 et 1.

2.3 Séquentiel

Le fichier *sequence.py* contient deux classes permettant de faciliter l'utilisation des réseaux de neurones.

La classe *Séquentiel* permet de représenter un réseau. Son constructeur prend en paramètres la liste des modules qui le composent. Il faut cependant s'assurer que la dimension des sorties de chaque module correspond bien à la dimension des entrées du module suivant. Cette classe contient notamment deux fonctions *save* et *load*, prenant en argument un nom de fichier et permettant respectivement de sauvegarder le réseau dans un fichier ou de le charger depuis ce fichier.

La classe *Optim* permet d’optimiser automatiquement un réseau. Son constructeur prend en paramètres un réseau (classe *Sequentiel* ou *AutoEncoder*, présenté dans une section ultérieure de ce rapport), une fonction de coût, et un pas ϵ pour la descente de gradient. Appeler la méthode *step* de la classe *Optim* permet d’effectuer une itération de l’optimisation du réseau. Cette méthode prend en paramètres un jeu de données X et les étiquettes associées Y . Enfin, la méthode *SGD* permet d’effectuer un entraînement complet du réseau. Elle prend en arguments les paramètres suivants :

- X : jeu de données
- Y : étiquettes associées
- *batch_size* : taille des batchs pour l’apprentissage
- *num_epochs* : nombre d’époques (itérations) à effectuer
- *X_test* : optionnel, permet de tester l’accuracy après chaque époque
- *Y_test* : optionnel, étiquettes des données de test
- *log* : variable permettant d’indiquer si on veut afficher les logs ou non.

Cette fonction renvoie la liste des valeurs de loss après chaque époque, ainsi que la liste des accuracy après chaque époque si un jeu de test est fourni.

2.4 Multi-classe

Les fonctions permettant une classification multiclasse sont implémentées dans le fichier *multi_class.py*.

La classe *Softmax* permet d’appliquer une fonction softmax aux entrées.

La classe *CrossEntropyWithLogSoftmax* représente une fonction de coût cross-entropique avec application d’un softmax et de la fonction log.

2.5 Tests

Nous avons réalisé un fichier de test par classe permettant de tester le fonctionnement des modules. Ils servent également d’exemples pour l’utilisation de notre implémentation. De plus, le dossier *scripts_benchmark* contient plusieurs scripts qui nous ont permis de réaliser les affichages de ce rapport.

3 Auto-Encoder

Nous avons implémenté des fonctions permettant de gérer facilement des auto-encoders dans le fichier *auto_encoder.py*.

La classe *BinaryCrossEntropy* représente un coût de cross-entropie binaire, utile pour l’apprentissage des auto-encoders.

Enfin, la classe *AutoEncoder* fonctionne comme le module *Sequentiel*, mais permet de générer automatiquement des auto-encoders. Son constructeur prend en paramètres la dimension

des entrées (qui sera également la dimension des sorties), une dimension minimum (qui correspond à la dimension de sortie de l'encodeur et de sortie du décodeur), et une variable *steps* correspondant au nombre de couches de l'encodeur et du décodeur. Le nombre de neurones dans chaque couche intermédiaire est généré automatiquement.

Une instance de la classe *AutoEncoder* peut être passée en paramètres de la classe *Optim* pour être entraîné.

Cette classe dispose également de méthodes *save* et *load* pour pouvoir enregistrer le réseau dans un fichier texte.

Le dossier *networks* contient un ensemble de 84 auto-encoders entraînés avec des paramètres variables. Ils peuvent être chargés avec la fonction *load* présentée précédemment.

4 Analyse expérimentale des auto-encoders

Pour évaluer la performance des auto-encoders, nous avons premièrement décidé d'analyser leurs performances pour la compression d'images. Pour cela, nous avons utilisé la base de données de chiffres MNIST. Cette dernière contient un peu plus de 56 000 images de chiffres en résolution 28x28, et constitue donc une bonne base d'entraînement. De plus, comme les images sont de basse résolution, cela permet de garder des réseaux de taille plutôt limité, et donc des temps d'apprentissage raisonnables.

En effet, puisque l'implémentation dépend intégralement de Numpy, nous ne pouvons pas profiter d'accélération matérielles liées au GPU comme le font les bibliothèques standard de machine learning (PyTorch pour ne citer qu'elle).

Nous avons donc entraîné différents auto-encoders, avec des fonctions de coût différentes, et en faisant varier les paramètres des modèles. Nous essaierons donc d'obtenir le meilleur taux de compression tout en assurant une bonne qualité pour la reconstruction des images.

4.1 Paramètres des modèles

Nous avons entraîné les 84 auto-encoders en faisant varier les critères suivants :

- **Fonction de coût** : nous avons utilisé deux fonctions de coût différentes pour l'apprentissage : la *MSE* et la *BCE*.
- **Learning rate** : comme la fonction de coût *BCE* semble nécessiter un apprentissage plus agressif, nous avons utilisé un pas de 0.01 pour la *MSE* et de 0.1 pour la *BCE* pour la descente de gradient.
- **Nombre d'epochs** : Pour chaque configuration, nous avons fait varier le nombre d'itérations pour l'apprentissage. Les configurations utilisant la *MSE* ont été entraînées pendant 10 à 20 itérations, et celles utilisant la *BCE*, nécessitant un apprentissage plus long, pendant 10 à 50 itérations.
- **Batch size** : nous avons également fait varier la taille des batchs pour l'apprentissage entre 32 et 64 éléments. Nous pourrions ainsi déterminer s'il s'agit d'un paramètre

important pour la qualité des inférences.

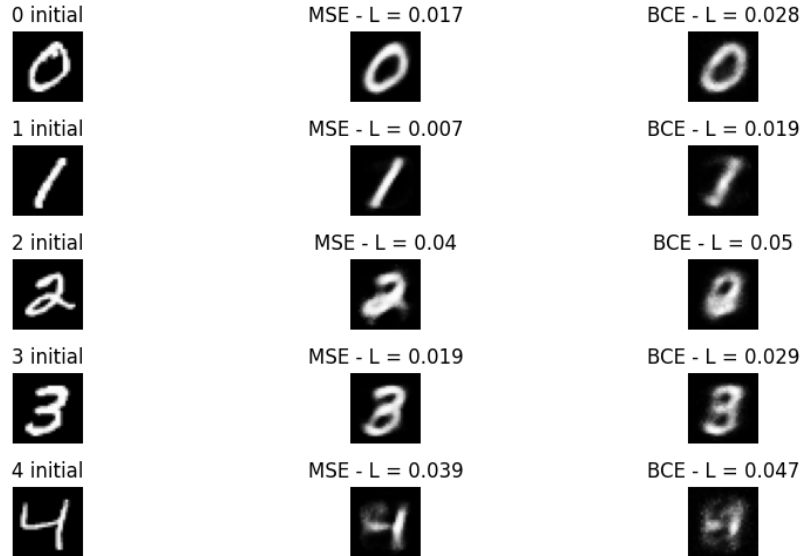
- **Nombre de neurones par couche** : le nombre de neurones par couche est paramétré par le nombre de valeurs en sortie de l'encoder. Le nombre de neurones à chaque couche intermédiaire est généré automatiquement pour être "bien réparti". Nous avons, pour chaque configuration, testé des couches à 16 ou 32 neurones.
- **Nombre de couches** : enfin, nous avons fait varier le nombre de couches dans l'encoder et le decoder pour chacune des configurations précédentes. Nous avons entraîné des modèles contenant 2, 4 ou 8 couches dans l'encoder (et autant dans le decoder).

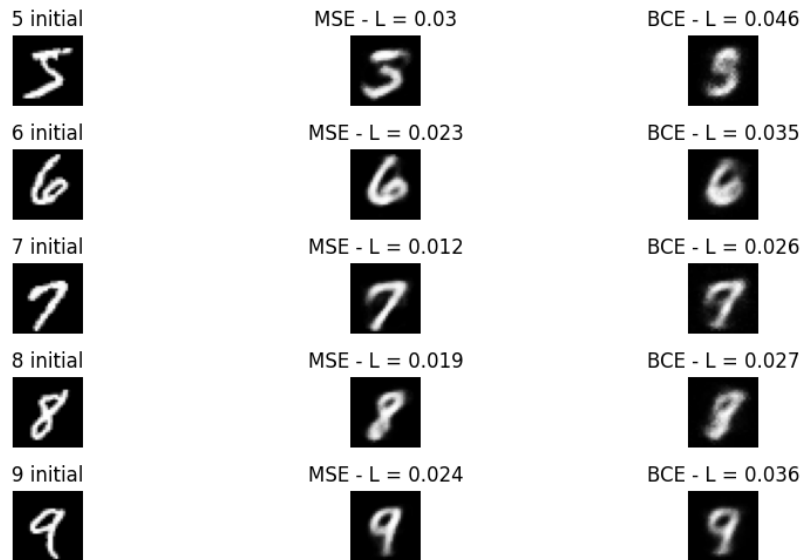
4.2 Comparaison des performances

Nous pouvons maintenant comparer l'impact des différents paramètres sur les auto-encoders.

Comparaison des fonctions de coût

Premièrement, nous comparerons la qualité de la reconstruction des images selon la fonction de coût utilisée pendant l'apprentissage. À part la fonction de coût, les paramètres utilisés sont similaires : 10 itérations pour l'apprentissage, 8 couches au total, et une compression en 16 neurones. Les données d'entraînements ont été découpées en batchs de 32 éléments. Nous utilisons ici un coût *MSE* pour évaluer la qualité de la reconstruction (valeur L au dessus des images). Nous pouvons ainsi observer les résultats suivants :



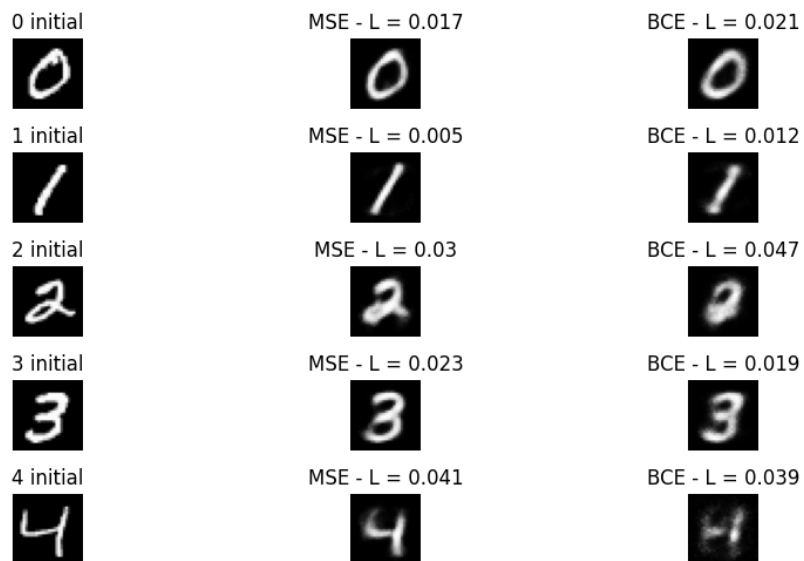


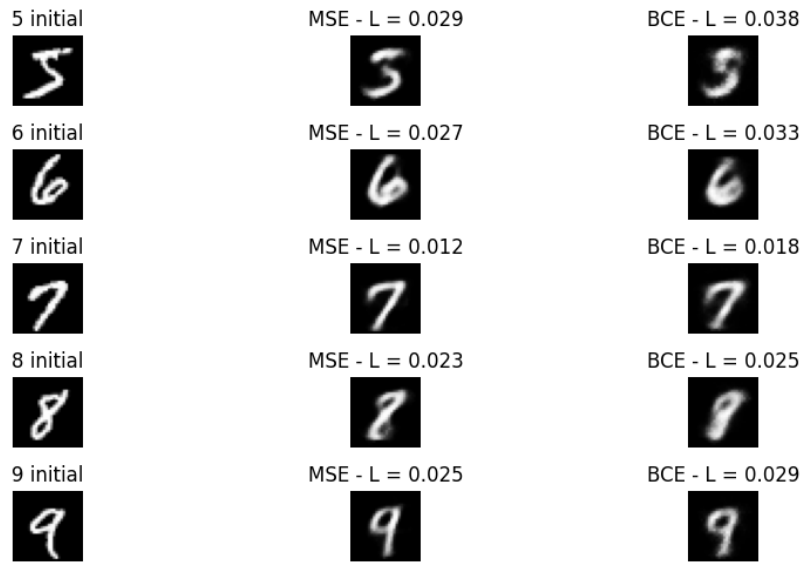
Si les résultats sont plutôt similaires visuellement pour les formes "simples" (1, 0), on remarque un net avantage pour le réseau entraîné avec la *MSE*.

Les contours sont naturellement plus flous que sur les images d'origine, mais la forme reste bien plus facile à distinguer que sur la sortie du réseau utilisant la *BCE*. Cette différence de performance se traduit également dans le coût, qui est environ 1,5 fois plus élevé sur les sorties du réseau *BCE* que sur celles du réseau *MSE*.

On peut tout de même noter que les deux réseaux rencontrent des difficultés avec l'image du 4, possédant initialement un tracé plus fin que les autres chiffres.

En comparant cette fois-ci les résultats obtenus après un entraînement des deux mêmes réseaux pendant 20 itérations, les résultats diffèrent légèrement :



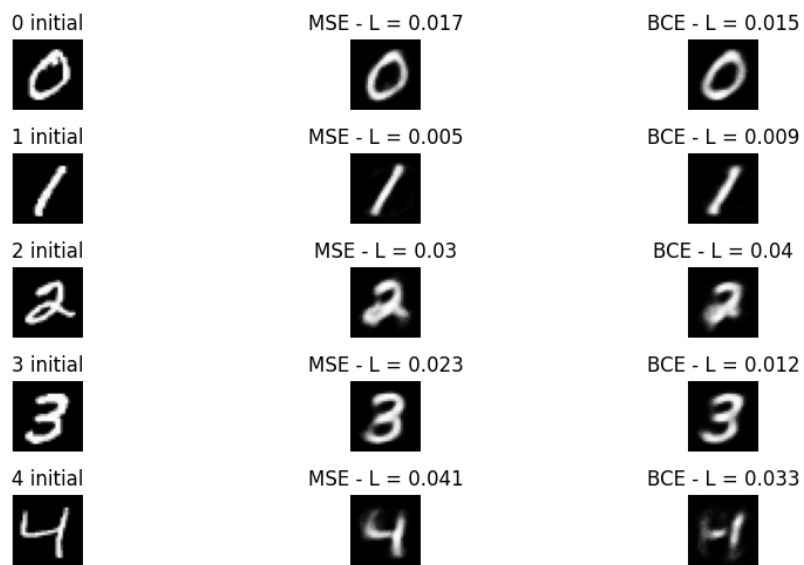

















Les résultats sont - comme attendu - meilleurs qu'en sortie des deux réseaux précédents. Le 4 est cette fois-ci mieux dessiné avec le réseau *MSE*, mais reste très approximatif en sortie du réseau *BCE*.

Bien que l'écart semble moins important, on note toujours visuellement un léger avantage pour le réseau *MSE*, ce qui confirme le fait que le réseau *BCE* nécessite plus d'itérations pour fournir de bons résultats.

Cependant, la différence au niveau du coût est cette fois-ci parfois en faveur du réseau *BCE*.

Nous avons donc décidé de conserver le réseau *MSE* tel quel, et de le comparer avec un réseau *BCE* similaire entraîné après 50 itérations.























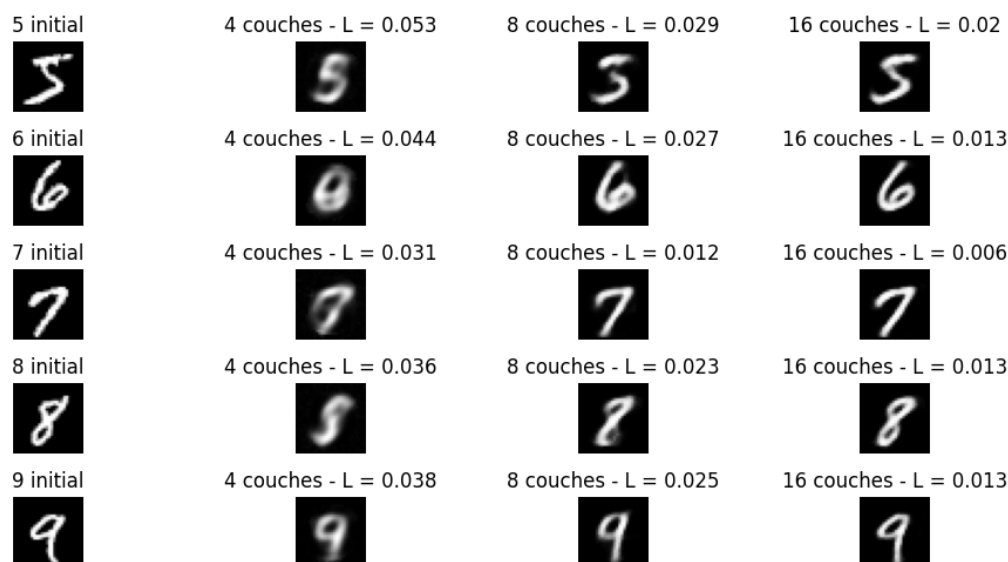
5 initial 	MSE - L = 0.029 	BCE - L = 0.031 
6 initial 	MSE - L = 0.027 	BCE - L = 0.027 
7 initial 	MSE - L = 0.012 	BCE - L = 0.012 
8 initial 	MSE - L = 0.023 	BCE - L = 0.022 
9 initial 	MSE - L = 0.025 	BCE - L = 0.022 

Même après 50 itérations, le réseau *MSE* continue à présenter des résultats supérieurs pour certaines images. Nous pouvons donc confirmer le fait que, pour ce cas, les deux fonctions de coût produisent des résultats plutôt similaires, mais que le réseau *BCE* nécessite un temps d'apprentissage plus conséquent.

Impact du nombre de couches

Nous pouvons maintenant nous intéresser à l'impact du nombre de couches dans le réseau. Pour cela, nous utiliserons un réseau entraîné avec un coût *MSE*. Les réseaux ont été entraînés sur des batches de 32 éléments, pendant 20 itérations. La couche minimale contient une nouvelle fois 16 neurones. En faisant varier le nombre de couches totales dans le réseau à 4, 8, ou 16, on obtient les résultats suivants :

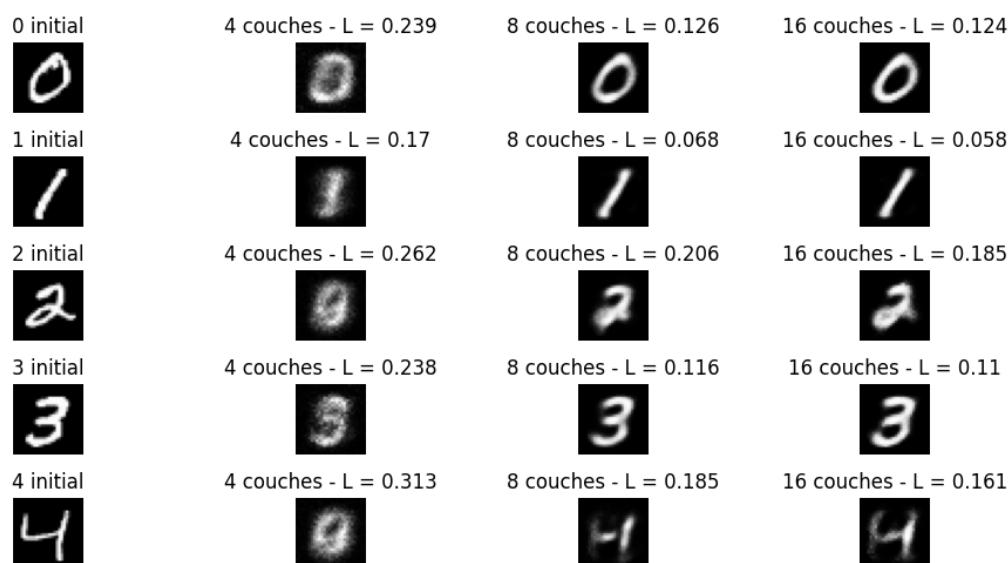
0 initial 	4 couches - L = 0.034 	8 couches - L = 0.017 	16 couches - L = 0.01 
1 initial 	4 couches - L = 0.032 	8 couches - L = 0.005 	16 couches - L = 0.002 
2 initial 	4 couches - L = 0.057 	8 couches - L = 0.03 	16 couches - L = 0.02 
3 initial 	4 couches - L = 0.057 	8 couches - L = 0.023 	16 couches - L = 0.011 
4 initial 	4 couches - L = 0.068 	8 couches - L = 0.041 	16 couches - L = 0.024 























On remarque cette fois-ci que le nombre de couches impacte grandement la qualité des résultats. Le coût en sortie des réseaux à 16 couches est très largement inférieur à celui des réseaux à 4 couches. De plus, visuellement, les résultats sont très similaires aux entrées, alors que les images, ayant initialement 784 dimensions (28x28), sont compressées en 16 valeurs en sortie de l'encodeur.

Nous pouvons cependant légitimement nous demander s'il est intéressant de conserver 16 couches plutôt que 8, car la différence entre les sorties est bien moins notable qu'entre les réseaux à 4 et 8 couches, alors que le nombre de paramètres est bien plus élevé.

Nous avons également réalisé le même test pour des réseaux entraînés avec un coût BCE, pendant 50 itérations :


















5 initial 	4 couches - L = 0.246 	8 couches - L = 0.167 	16 couches - L = 0.157 
6 initial 	4 couches - L = 0.228 	8 couches - L = 0.155 	16 couches - L = 0.128 
7 initial 	4 couches - L = 0.227 	8 couches - L = 0.105 	16 couches - L = 0.09 
8 initial 	4 couches - L = 0.22 	8 couches - L = 0.137 	16 couches - L = 0.128 
9 initial 	4 couches - L = 0.194 	8 couches - L = 0.126 	16 couches - L = 0.115 
















Nous pouvons tirer les mêmes conclusions que pour les réseaux *MSE*.

Impact du nombre de neurones par couches

Pour comparer l'impact du nombre de neurones par couches, nous avons comparé deux réseaux entraînés pendant 20 itérations, sur des batch de 32 éléments, et possédant 8 couches au total. En faisant varier le nombre de neurones dans la plus petite couche entre 16 et 32 (et par conséquent le nombre de neurones dans les couches intermédiaires), nous obtenons les résultats suivants.
















Premièrement, pour un réseau entraîné avec une fonction de coût *MSE* :
















0 initial 	16 neurones - L = 0.017 	32 neurones - L = 0.023 
1 initial 	16 neurones - L = 0.005 	32 neurones - L = 0.005 
2 initial 	16 neurones - L = 0.03 	32 neurones - L = 0.027 
3 initial 	16 neurones - L = 0.023 	32 neurones - L = 0.011 
4 initial 	16 neurones - L = 0.041 	32 neurones - L = 0.036 

5 initial 	16 neurones - L = 0.029 	32 neurones - L = 0.031 
6 initial 	16 neurones - L = 0.027 	32 neurones - L = 0.016 
7 initial 	16 neurones - L = 0.012 	32 neurones - L = 0.008 
8 initial 	16 neurones - L = 0.023 	32 neurones - L = 0.016 
9 initial 	16 neurones - L = 0.025 	32 neurones - L = 0.018 

À première vue, les résultats semblent très similaires. On ne note pas de différence très importante entre les résultats en sortie, tant que l'on ne s'intéresse pas au coût de la reconstruction. En effet, on observe tout de même que le coût en sortie du réseau à 32 neurones est sensiblement inférieur à celui en sortie du réseau à 16 neurones, malgré quelques anomalies (pour le chiffre 3 notamment).

En appliquant les mêmes configurations à des réseaux entraînés avec un coût *BCE* pendant 50 itérations, nous obtenons les résultats suivants :
















0 initial 	16 neurones - L = 0.126 	32 neurones - L = 0.119 
1 initial 	16 neurones - L = 0.068 	32 neurones - L = 0.058 
2 initial 	16 neurones - L = 0.206 	32 neurones - L = 0.174 
3 initial 	16 neurones - L = 0.116 	32 neurones - L = 0.107 
4 initial 	16 neurones - L = 0.185 	32 neurones - L = 0.164 
















5 initial 	16 neurones - L = 0.167 	32 neurones - L = 0.151 
6 initial 	16 neurones - L = 0.155 	32 neurones - L = 0.141 
7 initial 	16 neurones - L = 0.105 	32 neurones - L = 0.095 
8 initial 	16 neurones - L = 0.137 	32 neurones - L = 0.117 
9 initial 	16 neurones - L = 0.126 	32 neurones - L = 0.11 

Une nouvelle fois, les conclusions tirées sont similaires à celles des réseaux *MSE*. Nous avons donc, lors de la conception du réseau, à faire un choix entre le léger gain en qualité obtenu avec le réseau à 32 neurones, et la compression 2 fois plus importante offerte par le réseau à 16 neurones.

Impact de la taille des batchs

Enfin, nous pouvons comparer l'impact de la taille des batchs utilisée pour le découpage des données à l'apprentissage. Nous comparerons les résultats sur un réseau entraîné avec un coût *BCE* pendant 50 itérations, sur un réseau possédant 8 couches et une compression des données en 16 neurones. Nous entraînerons le réseau avec des batchs de taille 32 pendant 20 itérations, et celui avec des batchs de taille 64 pendant 40 itérations.

0 initial 	Batch Size = 32 - L = 0.148 	Batch Size = 64 - L = 0.15 
1 initial 	Batch Size = 32 - L = 0.085 	Batch Size = 64 - L = 0.085 
2 initial 	Batch Size = 32 - L = 0.23 	Batch Size = 64 - L = 0.216 
3 initial 	Batch Size = 32 - L = 0.145 	Batch Size = 64 - L = 0.148 
4 initial 	Batch Size = 32 - L = 0.213 	Batch Size = 64 - L = 0.209 

5 initial 	Batch Size = 32 - L = 0.193 	Batch Size = 64 - L = 0.209 
6 initial 	Batch Size = 32 - L = 0.176 	Batch Size = 64 - L = 0.169 
7 initial 	Batch Size = 32 - L = 0.127 	Batch Size = 64 - L = 0.126 
8 initial 	Batch Size = 32 - L = 0.15 	Batch Size = 64 - L = 0.15 
9 initial 	Batch Size = 32 - L = 0.149 	Batch Size = 64 - L = 0.15 

Nous pouvons globalement observer un léger avantage pour le réseau entraîné sur des batchs de taille 32, mais la différence est insignifiante. Nous pouvons donc conclure que, tant que la taille des batchs reste faible par rapport à la quantité de données, ce paramètre n'impacte pas la qualité des résultats en sortie.

Conclusions sur l'évaluation des paramètres

En conclusion, nous pouvons donc tirer les interprétations suivantes :

- Augmenter le nombre d'itérations pour l'apprentissage améliore dans tous les cas la qualité de la reconstruction en sortie, au détriment du temps d'entraînement du réseau.
- Augmenter le nombre de couches dans le réseau améliore également grandement la qualité de la reconstruction en sortie, au détriment de la taille du réseau et du nombre de paramètres total.
- Augmenter le nombre de neurones par couche améliore en premier lieu la qualité de la reconstruction en sortie. Cependant, cette différence est relativement faible, et se fait au détriment de la taille des données compressées.
- La taille des batchs pour l'apprentissage, tant qu'il reste faible, n'impacte pas la qualité des données en sortie.
- Les deux fonctions de coût, MSE et BCE , offrent des résultats convenables. Les résultats obtenus avec les réseaux entraînés avec un coût BCE offrent des résultats parfois un peu plus nets ou fidèles aux entrées, mais nécessite un temps d'apprentissage bien plus important.

4.3 Utilisations possibles des auto-encoders

Nous allons maintenant nous intéresser à deux utilisations pratiques des auto-encoders : le débruitage et la reconstruction de données manquantes.

Nous pouvons donc sélectionner un modèle particulièrement prometteur selon les critères analysés précédemment, et évaluer ses performances sur nos deux cas d'études.

4.3.1 Débruitage

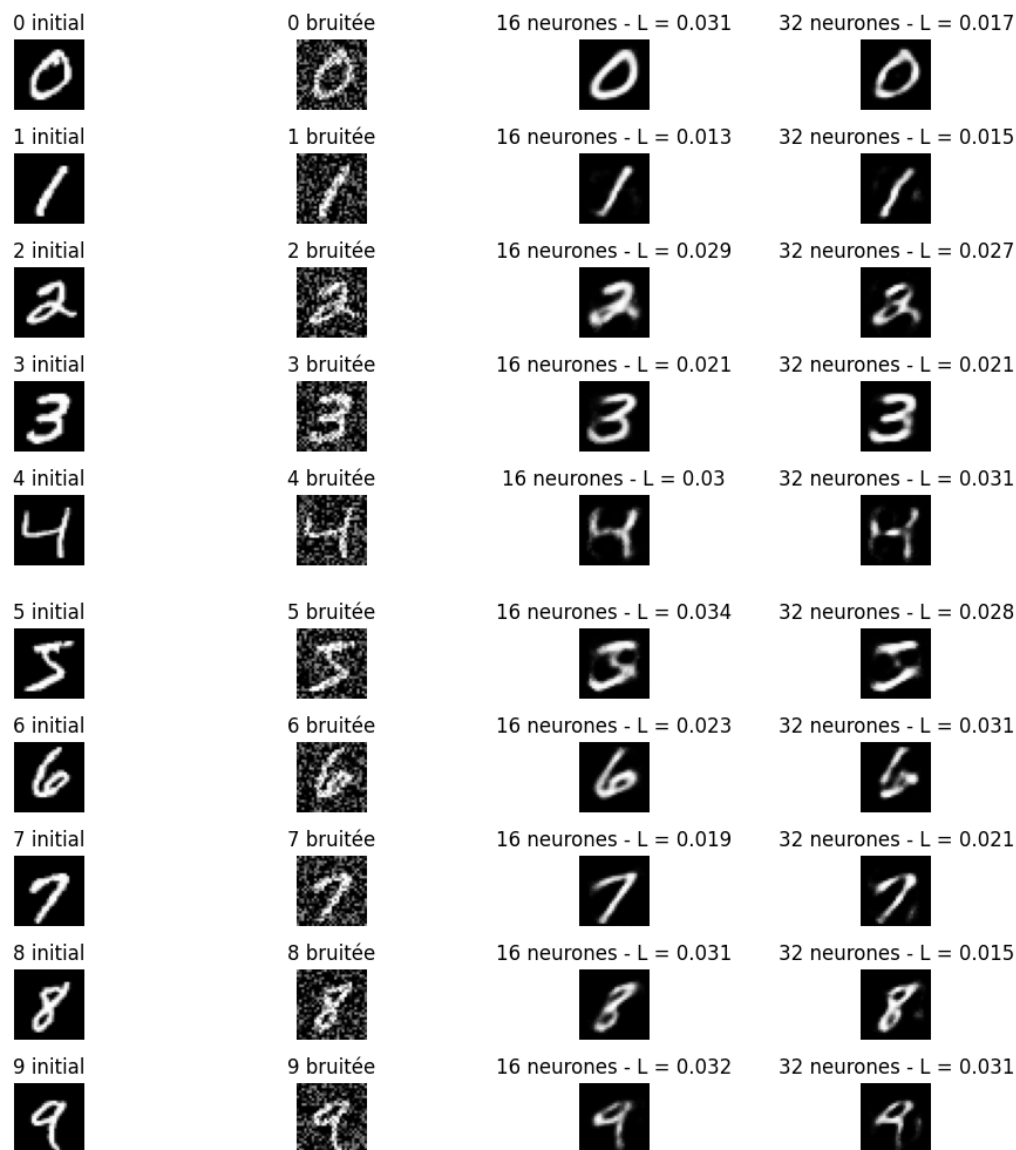
Évaluation des performances

Premièrement, nous comparerons l'efficacité de deux réseaux entraînés avec un coût MSE pendant 20 itérations, possédant 16 couches et respectivement 16 ou 32 neurones dans leur couche la plus petite.

Nous avons appliqué un bruit faible bruit pouvant ajouter ou retirer jusqu'à 0.4 à la valeur de chaque pixel.







Nous remarquons premièrement que le deux réseaux offrent de très bonnes performances pour le débruitage, avec des coûts de reconstruction par rapport à l'image initiale particulièrement faibles. Le réseau à 32 neurones semble cependant légèrement plus performant. Cependant, lorsqu'on augmente la puissance du bruitage (plus ou moins 0.6), nous obtenons des résultats différents :



Cette fois-ci, le réseau à 16 neurones égale et même dépasse les performances du réseau à 32 neurones en terme de coût. De plus, les résultats en sortie du réseau à 16 neurones sont visuellement plus précis et significativement plus proches des entrées que celles en sortie du réseau à 32 neurones.

Nous pouvons donc conclure qu'il faut ajuster le nombre de neurones par couches en fonction de la puissance du bruitage : un nombre de neurones plus faibles fournit un débruitage plus

puissant, mais moins précis si le bruitage des données est faible.
 Nous pouvons maintenant comparer les performances du réseau à 16 neurones avec un réseau équivalent, entraîné avec un coût *BCE* pendant 50 itérations :

0 initial 	0 bruitée 	MSE - L = 0.029 	BCE - L = 0.036 
1 initial 	1 bruitée 	MSE - L = 0.012 	BCE - L = 0.013 
2 initial 	2 bruitée 	MSE - L = 0.031 	BCE - L = 0.043 
3 initial 	3 bruitée 	MSE - L = 0.021 	BCE - L = 0.024 
4 initial 	4 bruitée 	MSE - L = 0.028 	BCE - L = 0.055 
5 initial 	5 bruitée 	MSE - L = 0.027 	BCE - L = 0.041 
6 initial 	6 bruitée 	MSE - L = 0.042 	BCE - L = 0.032 
7 initial 	7 bruitée 	MSE - L = 0.016 	BCE - L = 0.019 
8 initial 	8 bruitée 	MSE - L = 0.027 	BCE - L = 0.028 
9 initial 	9 bruitée 	MSE - L = 0.03 	BCE - L = 0.047 

Nous pouvons remarquer que le réseau *MSE* offre dans presque tous les cas des résultats supérieurs au réseau *BCE*.

Conclusions sur le débruitage

Nous pouvons donc déduire qu'un réseau entraîné avec un coût *MSE* est particulièrement performant pour débruiter des données. Le nombre de neurones par couche, quand il est diminué, permet d'améliorer la puissance du débruitage, au détriment de la qualité en sortie si les données sont peu bruitées.

4.3.2 Reconstruction de parties manquantes

Évaluation des performances

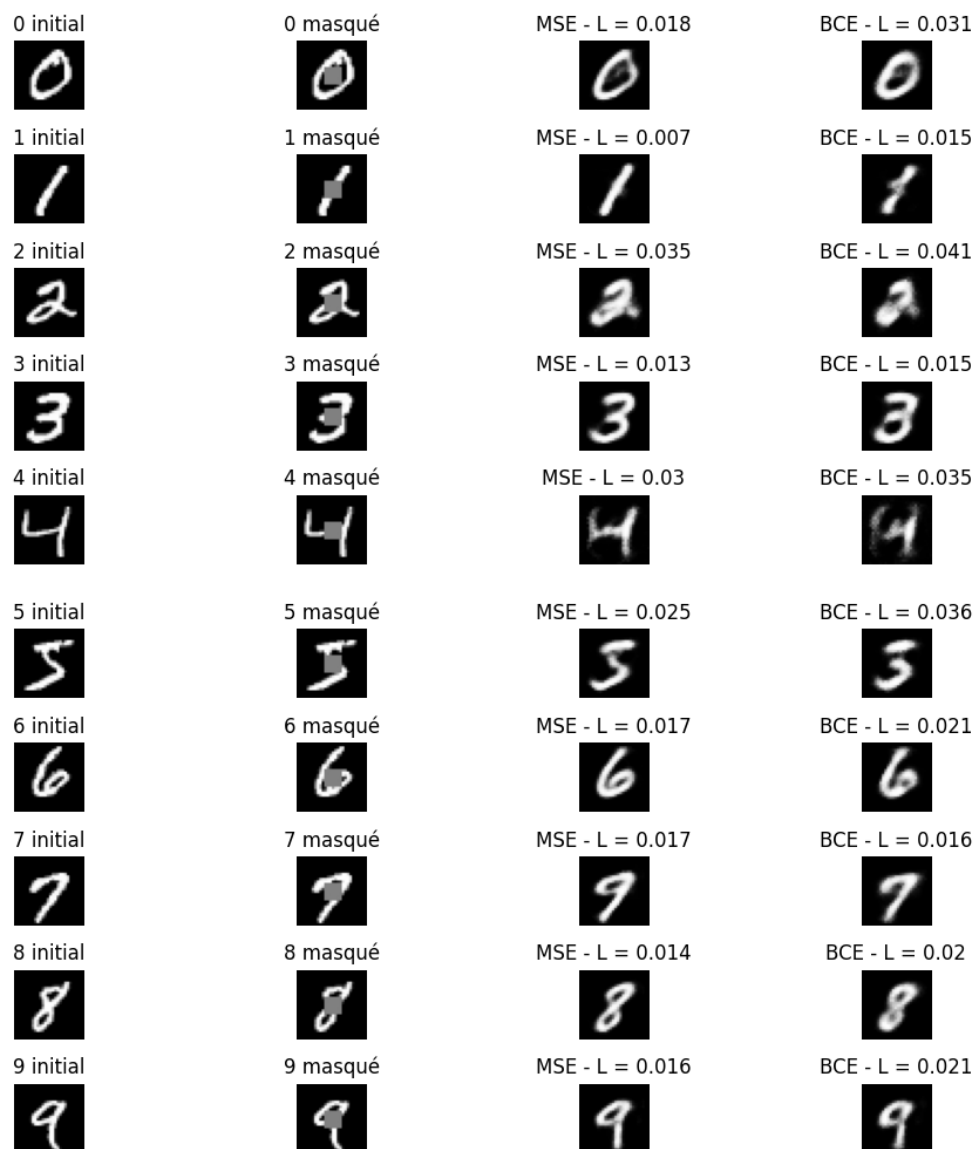
Cette fois-ci, nous supprimerons une partie des images pour analyser la capacité de l'auto-encoder à combler les données manquantes. Nous avons utilisé les deux mêmes réseaux que pour le premier test de débruitage (deux réseaux *MSE*). En masquant une petite partie de l'image initiale, nous obtenons les reconstructions suivantes :



Bien qu'en terme de coût les deux reconstructions soient plus ou moins équivalentes, on remarque tout de même visuellement des reconstructions plus fidèles avec le réseau utilisant une couche minimale à 16 neurones sur la plupart des chiffres. Les performances pour la

reconstruction sont donc très satisfaisantes, tant que la zone de données manquantes reste limitée.

Nous pouvons désormais comparer le réseau à 16 neurones avec son équivalent, entraîné avec un coût *BCE* :



Nous pouvons remarquer que les performances du réseau *BCE* sont également très satisfaisantes. Cependant, le coût de reconstruction est souvent légèrement supérieur à celui du réseau *MSE*.

Conclusions sur la reconstruction des données manquantes

Nous pouvons donc conclure que les auto-encoders sont également un bon moyen de reconstruire des parties manquantes dans les images. Bien que le résultat soit légèrement plus

approximatif que pour le débruitage, la qualité de la reconstruction est tout de même très prometteuse tant que la zone de donnée manquante reste limitée. Un réseau entraîné avec un coût MSE offre de meilleurs résultats, et il est important de ne pas avoir trop de neurones dans la couche de taille minimale.

5 Conclusion

Les résultats de nos expérimentations permettent de conclure qu'il existe de nombreux paramètres influant sur les performances des auto-encoders. Bien que notre cas d'étude soit limité (puisque se basant uniquement sur une base de données de petites images de chiffres), il permet déjà de se rendre compte de l'impact des différents choix d'architecture.

De plus, nous avons montré la capacité des auto-encoders à résoudre plusieurs problèmes courants, à savoir débruiter des images et les reconstruire lorsqu'une partie des données est manquante.