MU4IN811 - Projet ML

Ferdinand Bujanowski
Daniel Panariti

Mai 2025

Table des matières

0 Introduction

Le développement de réseaux de neurones pour l'apprentissage automatique est grandement facilité par l'existence de nombreuses bibliothèques et frameworks, disponibles dans la plupart des langages de programmation. En **Python**, des outils tels que `PyTorch` ou `TensorFlow` permettent de construire rapidement des architectures complexes. Ces bibliothèques facilitent le travail des développeurs de manière significative, en créant

automatiquement des couches et des réseaux entiers, ainsi qu'en effectuant automatiquement des calculs de gradient, qui sont indispensables à l'apprentissage automatique.

Pour mieux comprendre le fonctionnement interne de ces outils et les mécanismes fondamentaux de l'apprentissage automatique, nous avons construit une telle bibliothèque à partir de zéro, en implémentant une architecture modulaire en Python qui peut dans la suite être utilisée afin de facilement créer n'importe quel réseau neuronal, et qui peut automatiquement mettre à jour les paramètres internes du réseaux, menant à une apprentissage en fonction des données présentées. Nous allons dans la suite développer plusieurs modules génériques de complexité croissante, en testant chaque module avec des données et des hyperparamètres variés, ainsi qu'en comparant les résultats avec une bibliothèque Python déjà existante : `PyTorch`.

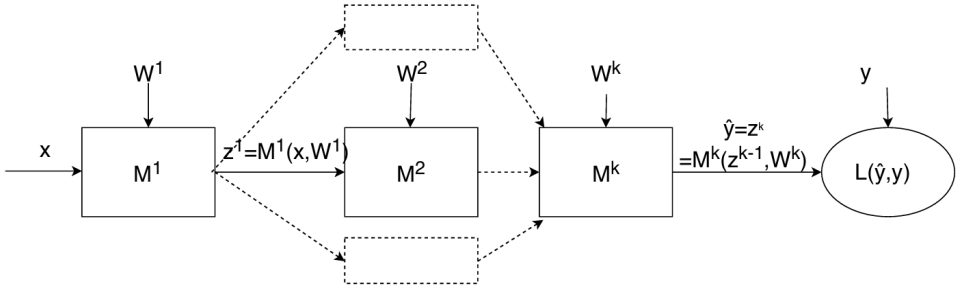

FIGURE 1 – Architecture module d'un réseau

L'idée derrière l'implémentation modulaire des réseaux de neurones est de faciliter la création des réseaux de taille variable, sans devoir réécrire pour chaque réseau ou pour chaque couche les calculs de sortie ainsi que les pas d'apprentissage, en généralisant les calculs qui sont les mêmes pour des "unités neuronales" de n'importe quelle taille. Avec cette approche, chaque couche de neurones (ainsi que chaque fonction d'activation) est vu comme module distinct. Un réseau de neurones est donc constitué de plusieurs modules, en fonction de son architecture exacte.

Afin d'apprendre les "meilleures" valeurs des paramètres internes d'un module donné (en fonction de la tâche posée), ce module doit pouvoir calculer la dérivée partielle d'une fonction globale de coût par rapport à chaque paramètre. Étant donné que ce coût est calculé à la sortie du réseau, on utilise la règle de dérivation en chaîne afin de calculer pour chaque module le gradient des paramètres. Cela veut dire que pour un module quelconque, les valeurs des paramètres des modules en amont ainsi qu'en aval ne sont pas nécessaires à connaître pour trouver les paramètres optimaux de ce module, ce qui facilite grandement les calculs à effectuer à chaque module.

0.1 Architecture générale du code

Nous avons d'abord des classes parentales *Module* et *Loss* qui servent comme les bases abstraites pour la création des couches, des fonctions d'activation, et des fonctions de coût. Ces classes ont des méthodes essentielles comme *forward()* et *update_parameters()* pour la classe *Module* et *forward()* et *backward()* for la classe *Loss*. Les modules et fonctions de coût utilisées dans la suite seront des classes dérivée de ces classes, héritant de leur classe abstraite correspondante.

1 Module Linéaire

Pour cette première partie, nous avons implémenté un module représentant une couche linéaire de neurones, ainsi qu'une fonction de coût correspondant au coût **MSE** ('mean squared error'). Avec ces deux éléments, nous avons donc pu réaliser une régression linéaire. En utilisant des données générées par `numpy` ainsi qu'une boucle effectuant l'algorithme de *descente de gradient*, nous avons testé la vitesse de convergence de l'apprentissage du réseau, en fonction de deux hyperparamètres : le taux d'apprentissage (*learning rate*) ainsi que la taille de mini-lots (*batch size*) utilisée pendant la boucle d'apprentissage. Finalement, nous avons étudié l'effet de la **normalisation des données** sur la vitesse de convergence de la régression.

1.1 Données et boucle d'apprentissage

Pour tester ce module linéaire, on a construit un jeu de données très simple modélisant une équation linéaire d'une dimension :

$$X = \{x_1, ..., x_n\} \qquad \forall i \in \{1, ..., N\}, x_i \in [0, 50]$$

$$Y = m * x_i + \epsilon \qquad \epsilon \sim N(0, 5^2), \forall x_i \in X$$

Ici, m signifie le paramètre qu'on souhaite apprendre avec le modèle. Pour apprendre ce paramètre, on a construit une fonction *train_model()* qui permet l'apprentissage par descente de gradient. On a donc des hyperparamètres tels que le taux d'apprentissage, le nombre d'époques, la taille de mini-lots (le nombre d'exemples traités simultanément à chaque itération, tout en parcourant l'ensemble des données à chaque époque), ainsi que la fonction de coût utilisé. Nous avons fixé le nombre d'époques à 200 et utilisé la fonction de coût **MSE**. Nous avons ensuite étudié l'impacte des différentes valeurs du taux d'apprentissage et de la taille des mini-lots sur le processus d'apprentissage. On a aussi testé l'effet de la normalisation des données sur l'apprentissage.

1.2 Hyperparameter tuning

La **Figure 2** présente les résultats de la régression linéaire pour quatre taux d'apprentissage (TA) différents. Chaque ligne, correspondante à un taux d'apprentissage en [1e-2, 1e-3, 1e-4, 1e-5] respectivement, regroupe le coût et le paramètre estimé par le module linéaire (égale à la pente de la régression) en fonction du nombre d'époques, ainsi qu'une comparaison des données d'entraînement avec la régression après l'apprentissage. Pour cette séries de tests, le nombre d'époques était fixé à 200, et la taille de mini-lots à 1.

On voit bien que pour les taux d'apprentissage de 0.01 et 0.001, il y a une convergence du coût ainsi que du paramètre estimé bien avant la fin des 200 époques. Plus grand le taux d'apprentissage, plus vite le modèle converge. Pour les deux premiers TAs, la prédiction finale de la pente de la régression semble très pertinente (0.95), en la comparant avec les données d'entraînement. Même pour le troisième TA, pour lequel le modèle converge beaucoup plus lentement, la prédiction finale (0.73) semble assez correcte, même si elle n'est pas aussi pertinente que celle des deux essais précédents. Cependant, nous pouvons conclure que le dernier taux d'apprentissage (1e-05) est beaucoup trop petit pour cette tâche de régression linéaire, vu que le modèle ne converge que très lentement, et étant donné que la prédiction finale n'est pas du tout en lien avec les données d'entraînement (prédiction du paramètre = -0.41).

La **Figure 3** les mêmes types de résultats (coût par époque, estimation du paramètre par époque, comparaison régression et données) pour trois essais indépendants avec des tailles de mini-lots (BS) variables. Pour cette série de tests, la BS variait entre 1, 10 et

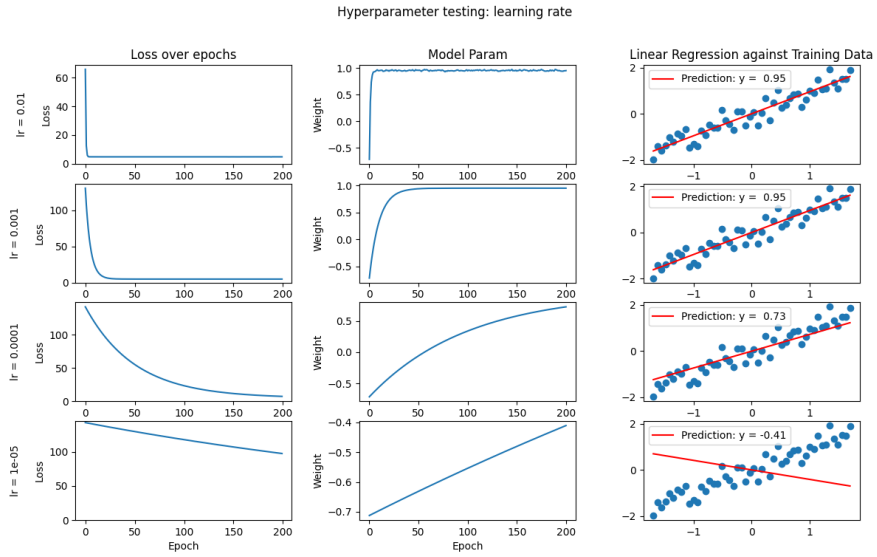

FIGURE 2 – Résultats de la régression linéaire pour 4 taux d'apprentissages différents

100 respectivement, pendant que le taux d'apprentissage était fixé à 0.01 (et toujours avec le nombre d'époques égal à 200). Avec une BS de 1, le modèle prend tous les exemples 1 à 1 par époque et fait sa mise à jour du paramètre après avoir vu chaque exemple, donc on a un paramètre qui est appris assez rapidement, et des variations non-lisses après convergence de cette valeur. Une BS de 100 implique donc une seule mise à jour par époque à partir de la moyenne des gradients sur l'ensemble des données, on a une convergence plus lisse vers la valeur du paramètre, et le coût calculé prend plus du temps a converger. Une BS de 10 semble être un bon compromis entre vitesse de convergence et stabilité des paramètres pour cette tâche.

1.3 Effect de la normalisation sur les résultats

Tous ces résultats précédents ont été obtenus en utilisant des données normalisées (et ayant donc une pente égale à 1). Nous nous sommes intéressés par l'effet de cette normalisation (où son absence) sur les résultats finaux de la régression, ainsi que le temps de convergence du coût et du paramètre appris :

La **Figure 4** montre les mêmes graphes que la figure 2, il s'agit donc des résultats de quatre apprentissages indépendantes, chaque fois utilisant un taux d'apprentissage différent. On peut observer même mieux que dans le cas précédant que plus le TA diminue, plus lentement le modèle converge. On voit aussi qu'avec des TAs assez grands (0.01 et 0.001), après convergence du modèle, la valeur du coût ainsi que celle du paramètre appris fluctuent beaucoup, cependant ce ne semble pas perturber les résultats finaux ; Pour chaque des quatre essais, la pente apprise ne varie que $<1\%$ par rapport à la pente réelle des données (qui est égale à 5). Ce résultat est meilleur que celui du *grid search* précédent, la convergence du modèle semble alors être moins sensible à la valeur précise du taux d'apprentissage lorsque les données ne sont pas normalisés.

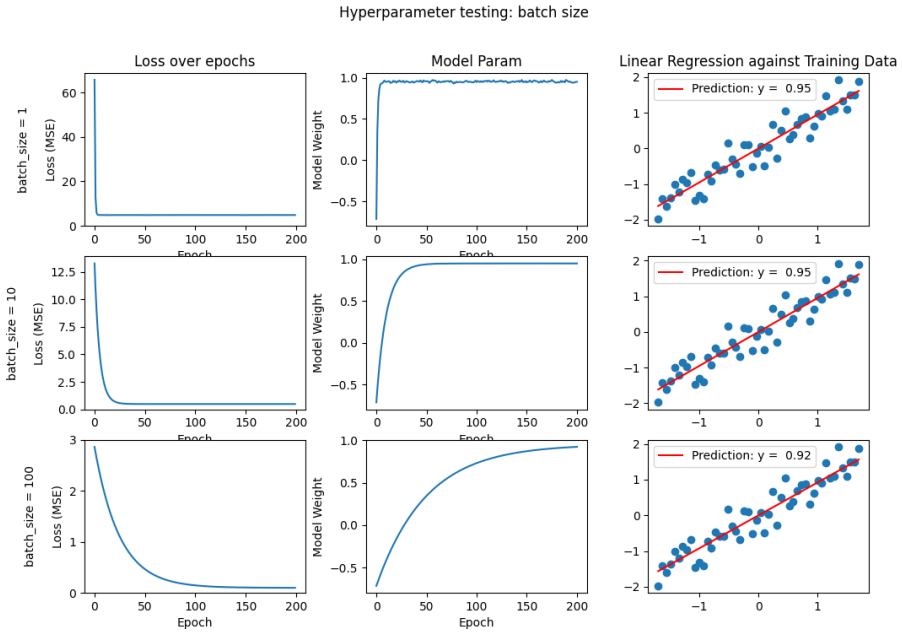

FIGURE 3 – Résultats de la régression linéaire pour 3 tailles de mini-lots différentes

2 Module Non-Linéaire

Pour cette seconde partie, l'objectif était d'implémenter des modules représentant des fonctions d'activation non linéaires, afin de concevoir un réseau capable d'effectuer une **classification binaire** sur des données d'entraînement qui ne sont pas linéairement séparables en deux classes. Ces deux classes sont composées d'une classe positive (typiquement $\mathcal{Y} \in \{+1\}$), et d'une classe négative (typiquement $\mathcal{Y} \in \{-1, 0\}$).

En termes généraux, pour pouvoir effectuer une classification, la valeur (ou le *score*) de chaque neurone de sortie du modèle étant donné une entrée de N dimensions doit être interprétable comme une **probabilité d'appartenance** à une classe donnée. Vu que dans notre cas la classification est **binaire et mono-label** (i.e. chaque élément n'appartient qu'à une seule classe), il suffit d'avoir un seul neurone de sortie, avec une fonction d'activation **sigmoïde**. Cette fonction, étant donné un *score* réel, le transforme en nombre entre 0 et 1. En choisissant un seuil entre ces deux nombres (ici $s = \frac{1}{2}$, prendre une décision sur la classe d'une entrée est donc égal à vérifier si le score pour cette entrée est $\geq s$.

Une autre fonction d'activation que nous allons utiliser pour ce modèle est celle de la **tangente hyperbolique**. Cette fonction correspond à peu près à une version dérivable de l'opération *sign()*, retournant alors le signe d'une valeur quelconque. Ceci n'est pas seulement très utile pour le calcul des gradients utilisé pour l'apprentissage automatique, mais en plus cette fonction introduit la **non-linéarité** dans notre modèle, le rendant plus fort dans son pouvoir décisif.

2.1 Architecture modulaire, données et apprentissage

Nous avons transformé ces deux fonctions d'activations susmentionnées en modules, afin de les utiliser dans notre modèle de classification binaire : Des données de taille D, sont passées à une première couche linéaire de N neurones (dans notre cas, $N = 4$) par

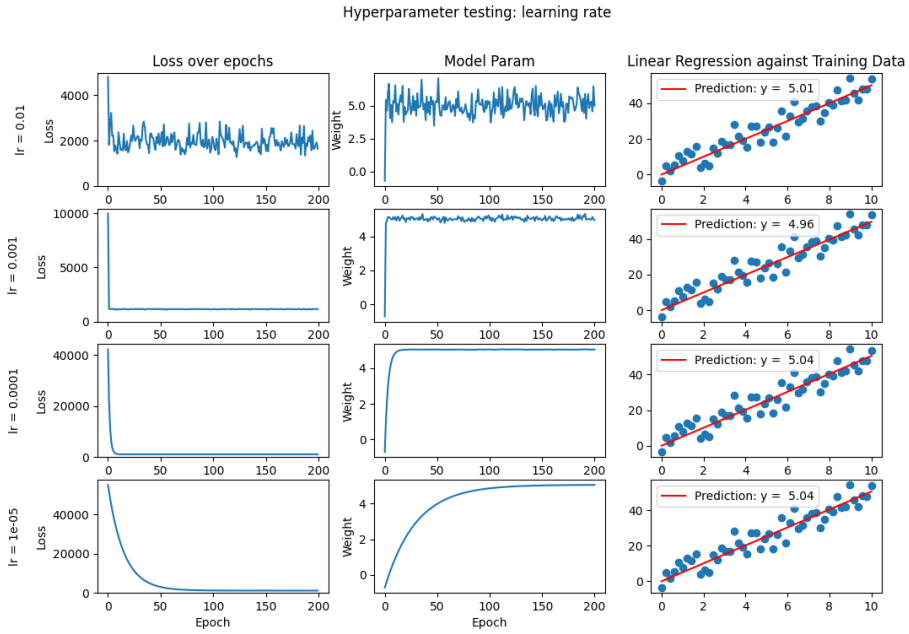

FIGURE 4 – Résultats de la régression linéaire pour 4 taux d'apprentissages différents - données non normalisées

combinaison linéaire avec une matrice de poids W_1, puis la sortie de cette couche est passée à la fonction d'activation **TanH**. La sortie de cette opération est alors passée à une deuxième couche linéaire, ayant un seul neurone (par combinaison linéaire avec la matrice W_2). Finalement, cette sortie réelle est passée à la fonction sigmoïde pour obtenir le *score* final attribué à l'entrée originale. Afin d'apprendre les valeurs optimales des deux matrices de poids, ces prédictions de classe $\hat{y}$ sont comparés aux vraies étiquettes de classe y par la fonction de coût **MSE**, la valeur de laquelle est utilisée dans le calcul de gradient.

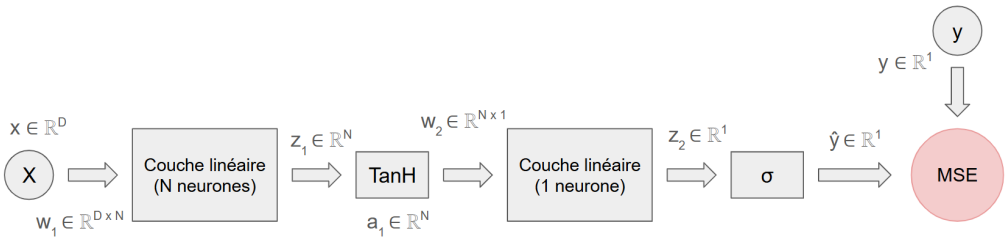

FIGURE 5 – Modules du modèle non-linéaire pour la classification binaire

Afin de tester ce modèle, nous avons créé un jeu de données générique à deux dimensions ($D = 2$), correspondant à deux *clusters* centrés autour des points (1, 2) pour la classe positive et (3, 5) pour la classe négative, avec du bruit gaussien. Pour chaque cluster on a crée 100 points, soit 200 points en total dans le jeu de données. Avant de commencer l'apprentissage, nous avons mélangé les données, puis elles ont été divisées en ensembles *train* et *test*, avec une proportion 20% pour l'ensemble *test*.

Nous avons créé deux modèles indépendants de même architecture (**Figure 5**), pour

tester l'effet du nombre d'époques sur les résultats de classification : Pour le premier modèle, 20 époques d'apprentissage ont été effectués, pendant que le deuxième modèle apprenait pendant 100 époques. Pour chaque époque, l'ordre des exemples *train* a été randomisé, puis chaque exemple a été passé dans le modèle (ici, le *batch size* était égal à 1) afin de calculer le *score*. Ce score a été utilisé pour calculer le coût MSE, puis à partir de ce coût les gradients de chaque module ont été calculé de l'arrière vers l'avant (*backwards propagation*), et les paramètres du modèle ont été mis à jour avec un taux d'apprentissage égal à 0.01. Après apprentissage, les prédictions respectives de classe, le coût en fonction d'époque, ainsi que les matrices de confusion *train* et *test* ont été visualisés.

2.2 Résultats de l'apprentissage

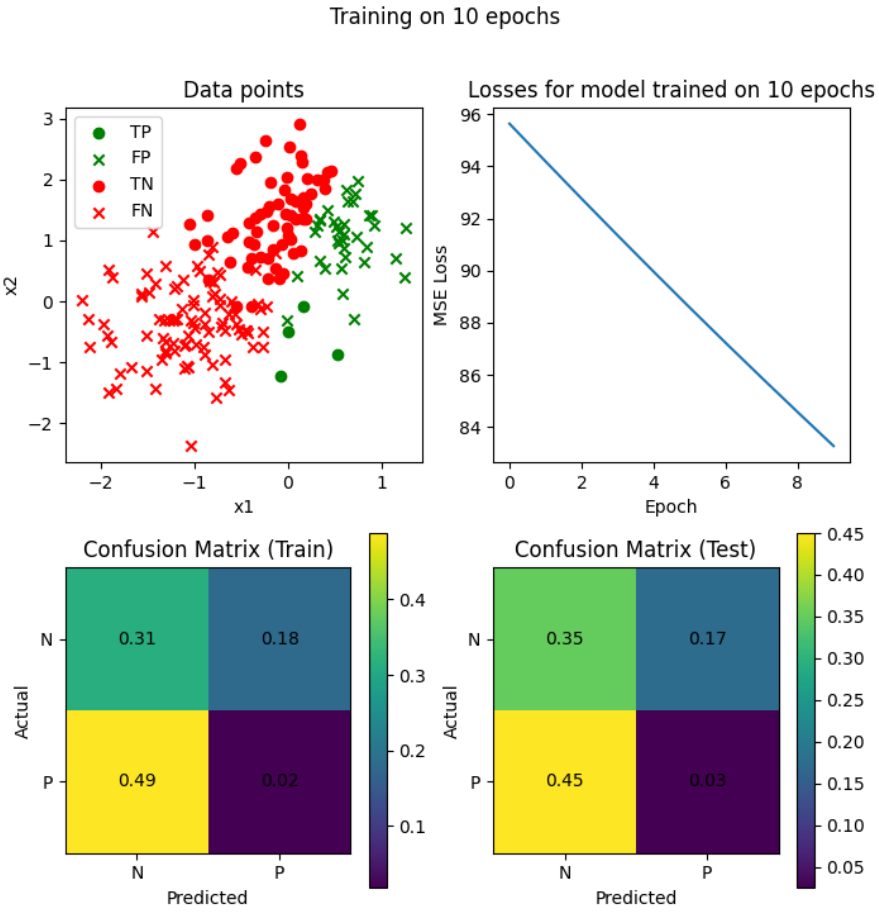

FIGURE 6 – Résultats d'apprentissage du modèle non-linéaire (20 époques)

La **Figure 6** regroupe les résultats du premier modèle, qui a pu apprendre pendant 10 époques. On voit bien que ce nombre d'époques n'est pas suffisant pour une convergence du modèle : Il y a une proportion assez grande d'exemples qui sont mal-classifiés, ce qu'on voit sur le graphe d'espace des données ainsi que sur les matrices de confusion

train et *test*. On voit également par la pente du coût à la dernière époque qu'il n'a pas encore convergé.

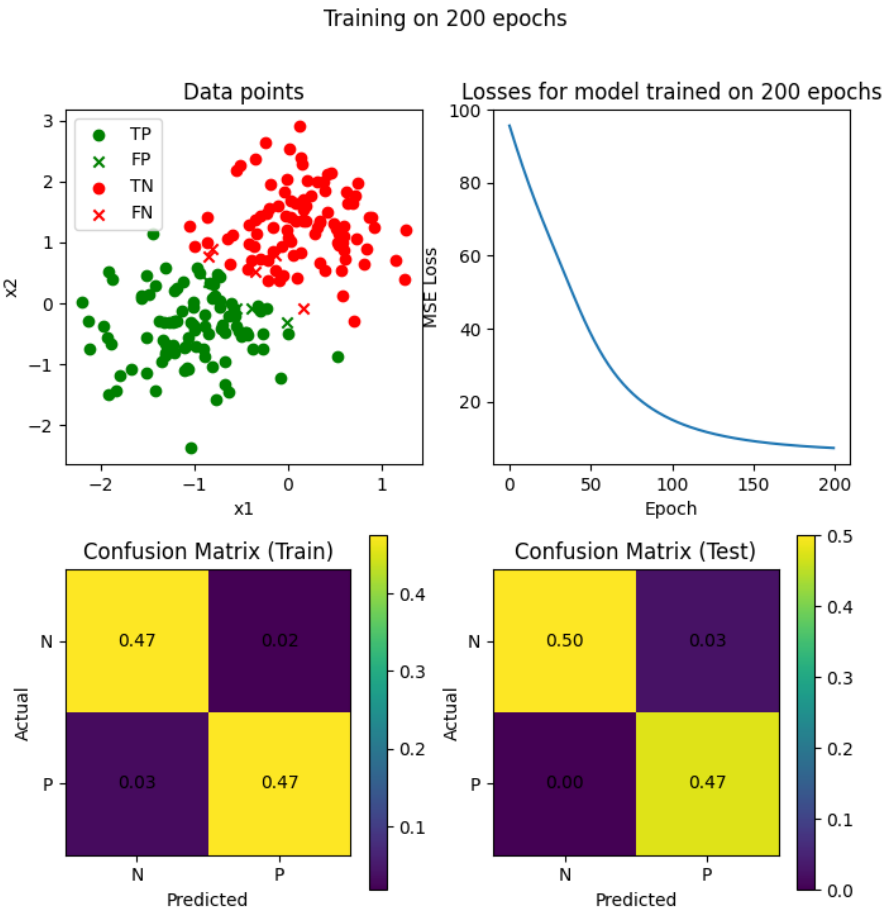

FIGURE 7 – Résultats d'apprentissage du modèle non-linéaire (200 époques)

Sur la **Figure 7**, correspondant au modèle qui a pu apprendre pendant 200 époques, on remarque des résultats beaucoup plus performants. La grande majorité des exemples sont correctement classés, comme en témoignent à la fois la représentation dans l'espace des données et les deux matrices de confusion, désormais fortement diagonales. La courbe du coût montre également une convergence bien plus marquée que dans le cas du premier modèle. Quelques faux positifs et faux négatifs subsistent, ce qui reste attendu étant donné que les données ne sont pas linéairement séparables.

3 Encapsulage

Dans les sections précédentes, pour effectuer la passe-forward et l'apprentissage par *backwards propagation*, nous avons enchaîné à la main les opérations correspondantes pour chaque module. Avec un nombre assez petit de modules, ceci est très faisable, cependant pour un grand réseau ayant pleins de modules, ce type d'enchaînage manuel

n'est pas très efficace, vu que les opérations sont très répétitives. C'est la raison pour laquelle, dans la suite, nous avons créé un module d'encapsulage *Sequential* qui effectue ces opérations de passe forward et backward automatiquement, pour un nombre quelconque de modules. Ceci n'est pas du tout une approche nouvelle, mais elle est le standard dans des librairies de réseaux de neurones comme `PyTorch`.

Afin d'optimiser encore plus notre code, nous avons également crée des fonctionnalités pour automatiser respectivement l'optimisation (qui enchaîne alors la passe forward et backward d'un réseau donné) ainsi que la descente de gradient stochastique (qui découpe automatiquement un jeu de données en *mini-lots* et entraîne un réseau pour un nombre d'époques donné). Les tests pour ces fonctionnalités seront développés plus en détail dans les sections suivantes.

3.1 Tests sur l'encapsulage

Pour tester le module *Sequential* qui effectue automatiquement une encapsulage de modules, nous avons recréé l'architecture du modèle précédent (classification binaire, **Figure 5**), encapsulé dans ce nouveau module. Pendant les pas d'apprentissage, le module a alors effectué automatiquement les pas forward et backward de tous les *sous*-modules. Nous avons utilisé le même jeu de données que dans le chapitre 3, et comparés ainsi les résultats de classification des deux modèles. Nous avons également créé un réseau de même architecture sous `PyTorch` pour nous assurer que les paramètres appris étaient les mêmes (étant donnée le même nombre d'époques, taux d'apprentissage ainsi que *seed*).

Model	**Training**	**Score**	**Layer 1 Weights**				**Layer 2 Weights**			
Sequential	Before Training	27.50%	1.33	0.72	-1.54	-0.01	1.33	0.72	-1.54	-0.01
			0.62	-0.72	0.27	0.11				
	10 epochs	37.50%	1.29	0.64	-1.45	-0.01	1.15	0.78	-1.48	-0.03
			0.55	-0.81	0.52	0.11				
	200 epochs	97.50%	1.22	-0.95	-0.28	0.34	-0.59	1.47	-0.93	-0.91
			0.40	-1.01	1.27	0.11				
PyTorch	Before Training	27.50%	1.33	0.72	-1.54	-0.01	1.33	0.72	-1.54	-0.01
			0.62	-0.72	0.27	0.11				
	10 epochs	37.50%	1.29	0.64	-1.45	-0.01	1.15	0.78	-1.48	-0.03
			0.55	-0.81	0.52	0.11				
	200 epochs	97.50%	1.22	-0.95	-0.28	0.34	-0.59	1.47	-0.93	-0.91
			0.40	-1.01	1.27	0.11				

TABLE 1 – Comparaison des poids appris du module *Sequential* et un réseau `PyTorch` pour différents nombres d'époques

Sur la **Figure 8**, on voit que pour n'importe quel des nombres d'époques testés, les réseaux *Sequential* et `PyTorch` ont les mêmes poids, et donc la même performance globale sur l'ensemble de données *test*. Ceci démontre que la passe forward ainsi que backward du module *Sequential* fonctionnent bien, vu que les méthodes correspondantes des sous-modules (couches linéaires et fonctions d'activations) n'ont pas été appelées explicitement par la boucle d'entraînement utilisée pour ces tests.

3.2 Tests sur la classe Optim

Après nous être assuré que le module *Sequential* fonctionne bien, nous avons implémenté la classe *Optim*, qui permet d'effectuer automatiquement un pas complet d'apprentissage à partir d'un réseau donné, un module de coût ainsi qu'un taux d'apprentissage ; en appelant la méthode *step()* de cette classe, elle calcule d'abord la sortie du réseau

pour une taille de mini-lots donné, puis elle calcule le coût, et finalement elle met à jour les paramètres du réseau en fonction du taux d'apprentissage donné.

Afin de tester si notre classe *Optim* fonctionne bien, nous avons récréé la tâche de régression linéaire du premier chapitre (*module linéaire*). Nous avons créé deux réseaux : le premier a été entraîné en appelant manuellement ses méthodes *forward()*, *backward()* et *update_parameters()*, pendant que le deuxième modèle a été entraîné en utilisant une boucle prenant ce modèle ainsi qu'une instance de la classe *Optim*, effectuant donc automatiquement ces trois pas d'apprentissage par méthode *step()* de cette instance. Pour les deux boucles d'apprentissage, le nombre d'époques était égal à 50, le taux d'apprentissage était égal à 0.01 et la *batch size* égale à 1.

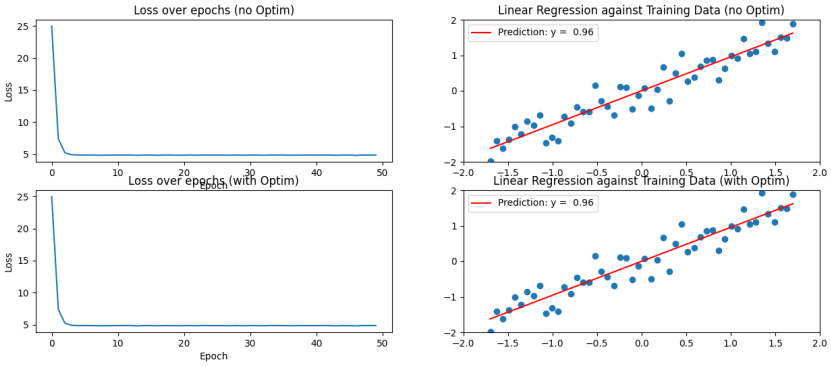

FIGURE 8 – Comparaison d'une régression linéaire sans et avec la classe Optim

La **Figure 8** montre les mêmes résultats de l'apprentissage pour les deux modèles, mettant en évidence que les pas d'apprentissage effectués pendant les boucles d'entraînement ont été effectués d'exactement la même manière. Ceci démontre que la classe *Optim* fonctionne correctement.

3.3 Descente de gradient stochastique

Finalement, afin d'automatiser encore plus les étapes d'entraînement d'un réseau donné, nous avons implémenté une fonction SGD qui effectue l'algorithme de la **descente de gradient stochastique**. Cette fonction prend un jeu de données, un réseau, une instance de la classe *Optim*, une taille de *batch* et un nombre d'époques en entrée, et s'occupe automatiquement de découper le jeu de données en batches et de mettre à jour les paramètres du réseau en utilisant l'optimiseur.

Pour tester cette fonction, nous avons effectué la même *hyperparamter search* que dans le premier chapitre (régression linéaire), et nous avons comparé les résultats entre l'approche manuelle originale et cette automatisation en utilisant la fonction *SGD()*. Étant donné que les résultats de cette *hyperparameter search* (**Figure 9**) sont exactement les mêmes que dans le premier chapitre, nous concluons alors que la fonction *SGD()* fonctionne correctement.

4 Multi-Classe

La classification des données en fonction de leur appartenance à une ou plusieurs catégories est un problème importante en apprentissage automatique. Dans cette section, nous allons détailler la classification multi-classe, les différences entre la classifi-

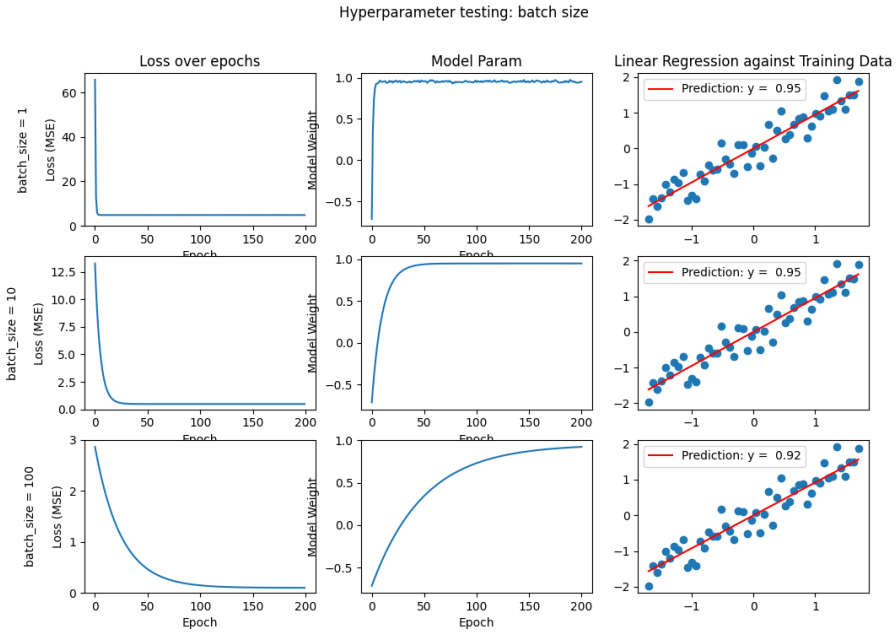

FIGURE 9 – *Hyperparameter search* pour la régression linéaire en utilisant la fonction *SGD()*

cation binaire et multi-classe, ainsi que les mécanismes sous-jacents aux fonctions de coût adaptées à ce type de problème, comme la fonction de coût *Cross-Entropy* et la transformation des sorties par la fonction *Softmax*.

Contrairement à la classification binaire, où il y a uniquement deux classes, ici, un modèle doit prédire à quelle classe l'exemple appartient parmi un ensemble de K classes. Les réseaux traitant les problèmes de classification multi-classe ont donc K neurones de sortie, avec chaque neurone représentant la probabilité d'appartenir à la classe correspondant à son indice. Chaque exemple est composé d'une seule classe correcte (il ne s'agit pas donc des exemples multi-label). Pour gérer les étiquettes dans un problème de classification multi-classe, un encodage *one-hot* est couramment utilisé où chaque étiquette est représentée par un vecteur de taille K, qui contient des zéros partout, sauf à l'index correspondant à la classe correcte, où la valeur est égale à 1. Cette représentation permet au modèle de prédire une probabilité pour chaque classe.

Pour rendre la couche de sortie interprétable comme des probabilités, nous applications la fonction *Softmax* à la sortie du réseau. *Softmax* est une fonction d'activation qui transforme les logits (valeurs réelles) en une distribution de probabilités, tel que la somme des probabilités est égale à 1. La probabilité de l'appartenance au classe i peuvent s'écrire de la façon suivante :

$$softmax(z)_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

Dans des problèmes de classification multi-classe, la fonction de coût *Cross-Entropy* est couramment utilisé. Cette fonction mesure la différence entre les distributions de sortie et celles du cible (les deux encodée en représentation *one-hot*). Cette fonction pénalise plus fortement les prédictions incorrectes que la MSE.

4.1 Base de données MNIST

Nous avons décidé de travailler avec la base de données MNIST (Modified National Institute of Standards and Technology), un standard dans l'apprentissage automatique. Cette base de données est composé des 70 000 images des chiffres allant de 0 à 9, de 28x28 pixels (**Figure 10**). Cette base de données est assez pertinente pour un problème de classification multi-classe. Comme précédemment, nous avons divisé les données dans une ensemble *train* et *test*. Pour ne pas travailler avec *tous* les données, ce qui prendrait beaucoup plus de temps pour entraîner, nous avons pris que 10 000 images en ensemble *train*. Le nombre d'exemples pour chaque classe sont présentés dans le **Tableau 2**.

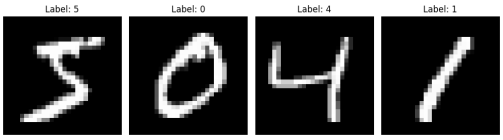

FIGURE 10 – Visualisation des données MNIST

Classe	Nombre d'exemples	Proportion
0	1001	10.01%
1	1127	11.27%
2	991	9.91%
3	1032	10.32%
4	980	9.8%
5	863	9.63%
6	1014	10.14%
7	1070	10.70%
8	944	9.44%
9	978	9.78%

TABLE 2 – Données MNIST constituant la partie *train*

4.2 Architecture du réseau et entraînement

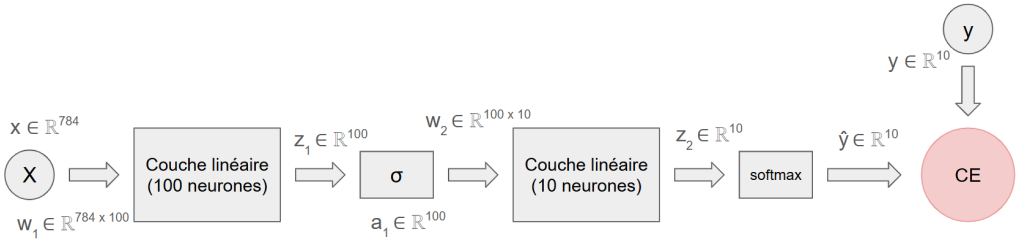

FIGURE 11 – Architecture du modèle multiclass

Pour tester l'apprentissage sur l'ensemble MNIST, nous avons construit un réseau non-linéaire avec deux couches linéaires ayant des fonctions d'activation non-linéaires ; l'entrée, étant de taille (28x28) correspondant à un image de 28 fois 28 pixels, est aplatit

en une dimension (donc 1x784) avant d'être passé à la première couche linéaire ayant 100 neurones. La fonction d'activation pour cette couche est la fonction sigmoïde. La sortie de cette fonction est à la suite passée à la deuxième couche linéaire, ayant 10 neurones (représentant les 10 classes possibles). Afin de pouvoir interpréter ces 10 sorties comme probabilités d'appartenance, comme expliqué dans la section précedente, la fonction *softmax* sera appliqué. Finalement, afin d'optimiser les matrices de poids de ce réseau, la sortie est comparé avec la vraie classe de l'entrée donnée (transformé en vecteur *one-hot*) par le coût *cross-entropy*, qui est donc utilisé pour calculer et propager les gradients.

Pour tester l'apprentissage de ce réseau, la classe *Optim* ainsi que la fonction *SGD* du chapitre précédent ont été utilisé, afin de rendre le plus efficace possible le code de boucle d'entraînement. Comme hyperparamètres, nous avons choisi un nombre d'époques égal à 500, une *batch size* égale à 100 ainsi qu'un taux d'apprentissage égal à 1e-4 (0.0001).

4.3 Résultats du modèle

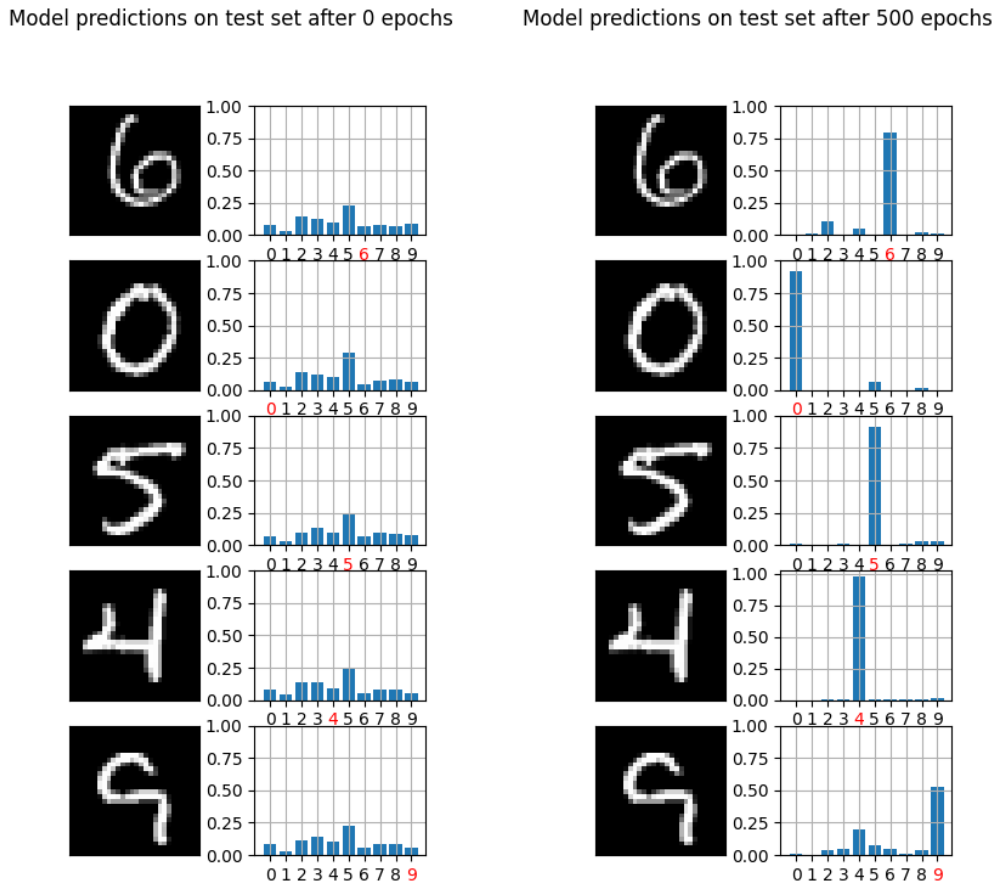

FIGURE 12 – Prédiction du modèle sur des données MNIST dans la proportion *test*, avant et après entraînement

La **Figure 12** regroupe les vecteurs des sortie du réseau multi-classe pour quelques

entrées exemplaires, avant et après l'entraînement précisé dans la section précédente. Pour chaque exemple, la vraie étiquette de cet exemple est indiqué en rouge dans le vecteur de sortie (représentant les probabilités d'appartenance). En comparant les vecteurs du modèle avant et après les 500 époques d'entraînement, on voit que les entropies respectives des vecteurs de sortie diminuent beaucoup, ce qui est logique étant donné que la fonction de coût utilisé pour entraîner ce modèle est la cross-entropy. Pendant que les vecteurs de sortie avant l'apprentissage sont totalement aléatoires, ils permettent de prédire correctement les vraies étiquettes après 500 époques (précision de 100% pour les 5 exemples montrés). Le modèle semble donc avoir appris avec succès les valeurs optimales des matrices de poids.

Il est intéressant d'observer le dernier exemple : un chiffre 9 mal écrit, dont la boucle supérieure n'est pas complètement fermée, ce qui le fait ressembler partiellement à un 4. Le modèle prédit tout de même le 9 comme classe la plus probable (environ 50%), mais attribue également une probabilité non négligeable au 4 (environ 24%). Cela montre que le réseau est capable de reconnaître le bon chiffre, tout en capturant l'ambiguïté liée à la forme écrite.

On remarque également une convergence de coût cross-entropy jusqu'à 500 époques (**Figure 13**). Ce nombre semble donc être un bon nombre pour une apprentissage effective, augmenter encore le nombre d'époques ne semble pas justifiable.

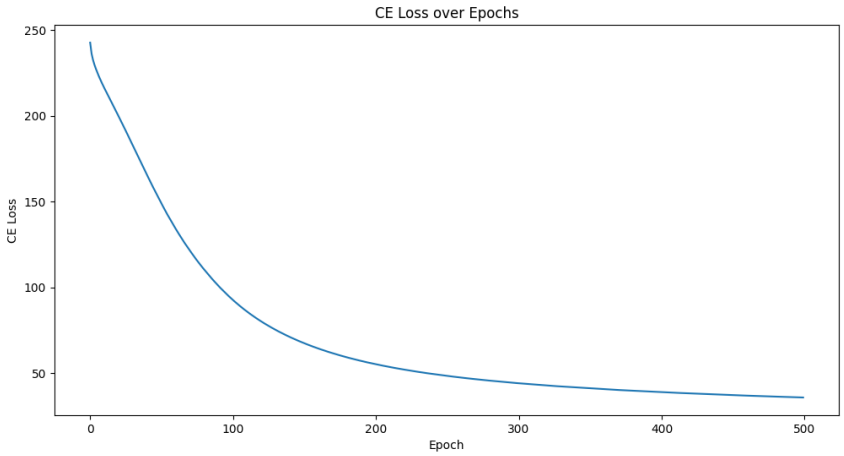

FIGURE 13 – Coût CE du réseau multiclasse

5 Auto-Encodeur

Nous allons finalement travailler avec des auto-encodeurs, un réseau de neurones capable d'appendre un encodage des données de dimension D en une représentation latente de dimension d, tel que d $\ll$ D. Ce réseau est composé d'une partie encodeur qui transforme une donnée dans un espace plus petit, et une partie décodeur qui reconstruit la donnée originale à partir de cette représentation latente.

Notre auto-encodeur sera comprise de trois couches d'encodage et trois couches de décodage, avec des fonctions *TanH()* comme fonction d'activation pour les couches intermédiaires, et *Sigmoid()* pour la dernière couche, afin de restreindre nos pixels entre 0 et 1. Nous utiliserons la MSE pour entraîner ce réseau. L'architecture de ce réseau est

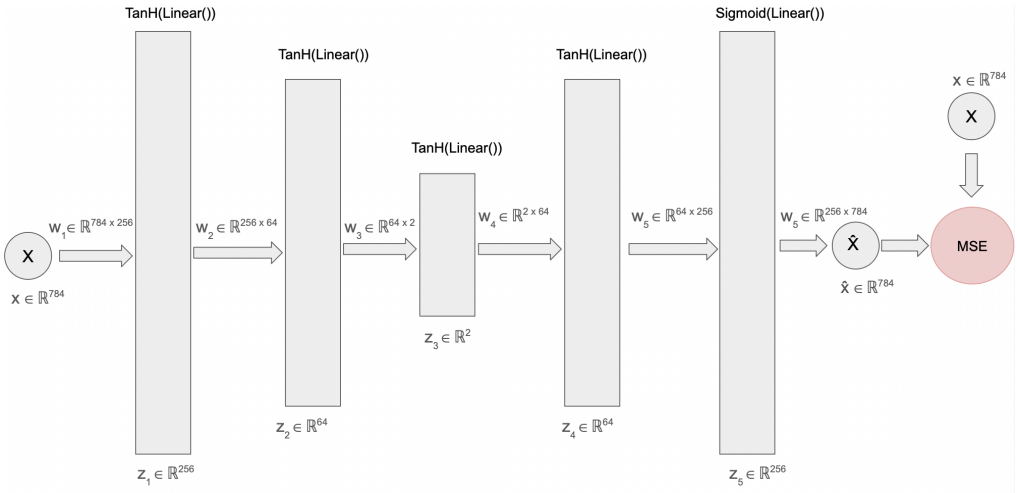

FIGURE 14 – Architecture de l'auto-encodeur

présenté sur la **Figure 14**. Nous fixerons le taux d'apprentissage à 0.5, et la taille de mini-lots à 2048.

5.1 Base des données FashionMNIST

Pour tester ce réseau, nous avons décidé de travailler avec la base de données FashionMNIST, une base de données des images de vêtements de 28x28 pixels, où chaque classe correspond au nom du vêtement (**Figure 15**). Pour gagner du temps dans l'apprentissage, nous avons décider de travailler qu'avec 10,000 exemples de cette base de données (**Tableau 3**). Comme précédemment, ces données seront divisé dans un batch test et un batch d'entraînement.

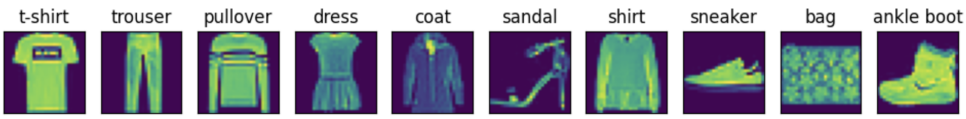

FIGURE 15 – Base de données FashionMNIST

5.2 Performance du réseau

Avec ce réseau, on voit bien qu'avec 0 époques d'apprentissage, l'image reconstruite est constitué des valeurs aléatoires, produisant du bruit visuel (**Figure 16**). En revanche, si on entraîne ce réseau pour 2000 époques (**Figure 17**), on arrive à reconstruire des images reconnaissables. Le réseau semble être capable de bien reconstruire des baskets, ainsi que des blouses.

On note parfois une reconstruction intermédiaire entre une blouse et un pantalon pour les pantalons, et le réseau n'arrive pas à reconstruire des des sacs, reconstruisant des formes intermédiaires entre un basket / blouse / pantalon. Cela s'explique en partie par la similarité visuelle entre certains articles vestimentaires comme les t-shirts, pulls, blouses, etc. qui partagent tous des profils similaires. De même, les articles tels que les

Classe	Article	Nombre d'exemples	Proportion
0	T-shirt	759	94.875%
1	Pantalon	823	10.2875%
2	Pull	813	10.1625%
3	Robe	819	10.2375%
4	Manteau	759	9.4875%
5	Sandale	794	9.925%
6	Blouse	833	10.4125%
7	Basket	814	10.175%
8	Sac	790	9.875%
9	Bottine	796	9.95%

TABLE 3 – Données FashionMNIST utilisées dans la proportion *train*

pantalons sont assez similaires avec ces articles précédemment mentionnées, partagent beaucoup des pixels en commun, ce qui rend leur représentation latente de ces articles relativement proche dans l'espace de codage.

Le coût du réseau pendant l'apprentissage est présenté sur la **Figure 18**. On peut observer que le coût diminue assez rapidement, arrivant à un plateau de valeur d'environ 0.35. Le réseau reste à ce plateau pendant environ 500 époques, ce qui suggère une stagnation dans un minimum local. Par la suite, le coût reprend sa diminution et arrive à un seconde plateau de moins de 0.3. Étant donné que le premier plateau s'est maintenu pendant environ 500 époques, il serait intéressant d'entraîner le modèle pendant une durée plus longue, par exemple 3000 époques. Néanmoins, cela dépasse le cadre de ce projet, en raison du temps d'entraînement relativement long requis pour ce réseau.

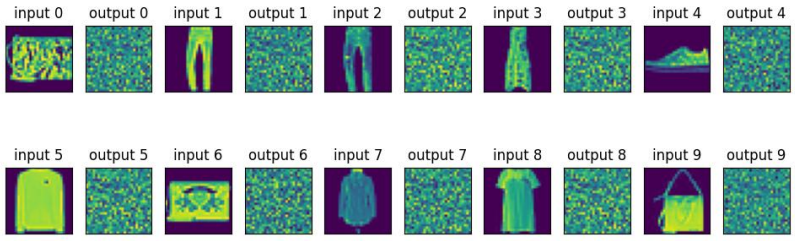

FIGURE 16 – Reconstruction des images du réseau après 0 époques d'apprentissage

5.3 Effet du bruit

Nous souhaitons ensuite analyser l'effet du bruit sur les capacités de reconstruction de ce réseau. Pour ce faire, nous avons construit une fonction *add_noise()* qui prend en paramètre un facteur d'échelle (noté ici c). Pour chaque pixel, nous ajoutons du bruit à partir d'une distribution gaussienne $\mathcal{N}(\mu = 0, \sigma = c)$ On peut observer l'effet du bruit sur la reconstruction des images en **Figure 19**.

On observe qu'avec un facteur d'échelle inférieur ou égale à 0.1, le réseau est capable de reconstruire les images comme s'il n'y avait pas du bruit. Les images reconstruites semblent identiques. À partir d'un facteur d'échelle de 0.5, on commence à avoir un bruit

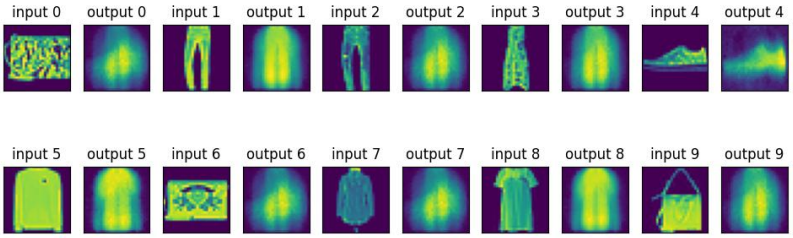

FIGURE 17 – Reconstruction des images du réseau après 2000 époques d'apprentissage

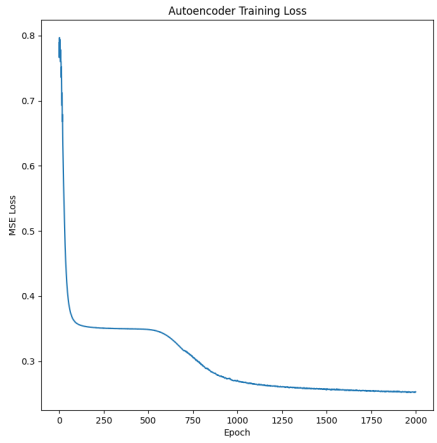

FIGURE 18 – Coût MSE de l'autoencodeur

fort (les entrées semblent correspondre avec du bruit), mais le réseau converge quand même vers des représentations bruités d'un basket ou d'un mixe pantalon-blouse.Avec un facteur d'échelle de 1, un plus grand nombre d'images reconstruites tendent à converger vers une représentation de basket, quelle que soit l'entrée, ce qui suggère une perte d'information significative dans la représentation latente.

On peut aussi analyser le coût MSE en fonction du bruit. Les résultats de cette comparaison est présenté sur la **Figure 20**. Lorsqu'on augmente le niveau du bruit par le facteur d'échelle, le coût MSE augmente.

5.4 Visualisation de l'espace latente

La dernière couche de notre réseau encodeur (responsable pour l'encodage de l'entrée de dimensions D vers une représentation de dimension d), est de dimensions d=2. Ce choix a été effectué afin de permettre la visualisation de l'espace latent et de mieux comprendre la manière dont les données y sont projetées. Nous avons décidé de regrouper les 9 classes en 3 groupes distincts, en fonction de leur similarité visuelle. Le détail de ce regroupement est présenté dans le **Tableau 4**.

Les représentations latentes sont illustrées dans la **Figure 21**. On y observe une séparation nette entre les chaussures et les hauts, bien que les chaussures présentent une

Classe	Articles	Regroupement
0	T-shirt, Blouse, Pull, Robe, Manteau	Survêtements / Hauts
1	Pantalon	Bas
2	Sandale, Basket, Bottine	Chaussures
3	Sac	Accessoires

TABLE 4 – Regroupement des classes pour la visualisation de l'espace latente

dispersion plus marquée. Entre ces deux groupes se situe le regroupement des sacs, caractérisé par une répartition relativement étendue des données. On remarque également un amas de points correspondant aux pantalons, intégré au cluster des hauts, ce qui est cohérent avec le fait que ces deux types d'articles partagent de nombreuses similarités visuelles, notamment au niveau des pixels. Cette représentation de l'espace latent met en évidence les dimensions les plus pertinentes pour la reconstruction, et nous donne des indications sur les similarités visuelles perçues par le réseau entre les différentes classes.

6 Discussion

Ce projet nous a permis de mieux comprendre les fondements du fonctionnement des réseaux de neurones, en particulier en codant manuellement les différents modules (*Linear*, *Sequential*, *TanH*, etc.) et fonctions de coût (*MSE*, *CE*). Cette implémentation "from scratch" nous a offert une perspective importante sur les mécanismes internes des bibliothèques telles que `PyTorch`, notamment la forward pass, la backward pass (ou rétropropagation), et la gestion explicite des gradients.

Nous avons également pu constater l'importance cruciale de la reproductibilité dans les expériences en apprentissage automatique. L'utilisation systématique d'un *seed* permet de garantir des résultats cohérents et comparables, surtout lors du débogage ou de l'évaluation des performances. Nous avons pu aussi réaliser l'importance dans le traitement des données (ou pas) avant de les utiliser dans un modèle quelconque.

Ce projet nous a aussi permis de réfléchir à la nature sémantique du problème d'apprentissage supervisé dans le cadre de la base de données MNIST et FashionMNIST dans l'analyse des résultats. Comprendre la structure des données, au-delà des étiquettes, est donc essentiel pour une analyse plus profonde du problème étudié.

Enfin, l'importance du choix des hyperparamètres (taux d'apprentissage, taille des mini-lots, nombre d'époques) et de l'architecture du réseau (nombre de couches, taille de la représentation latente) s'est révélée instructive. L'ajustement judicieux de ces paramètres est souvent la clé d'une convergence efficace et d'une bonne généralisation.

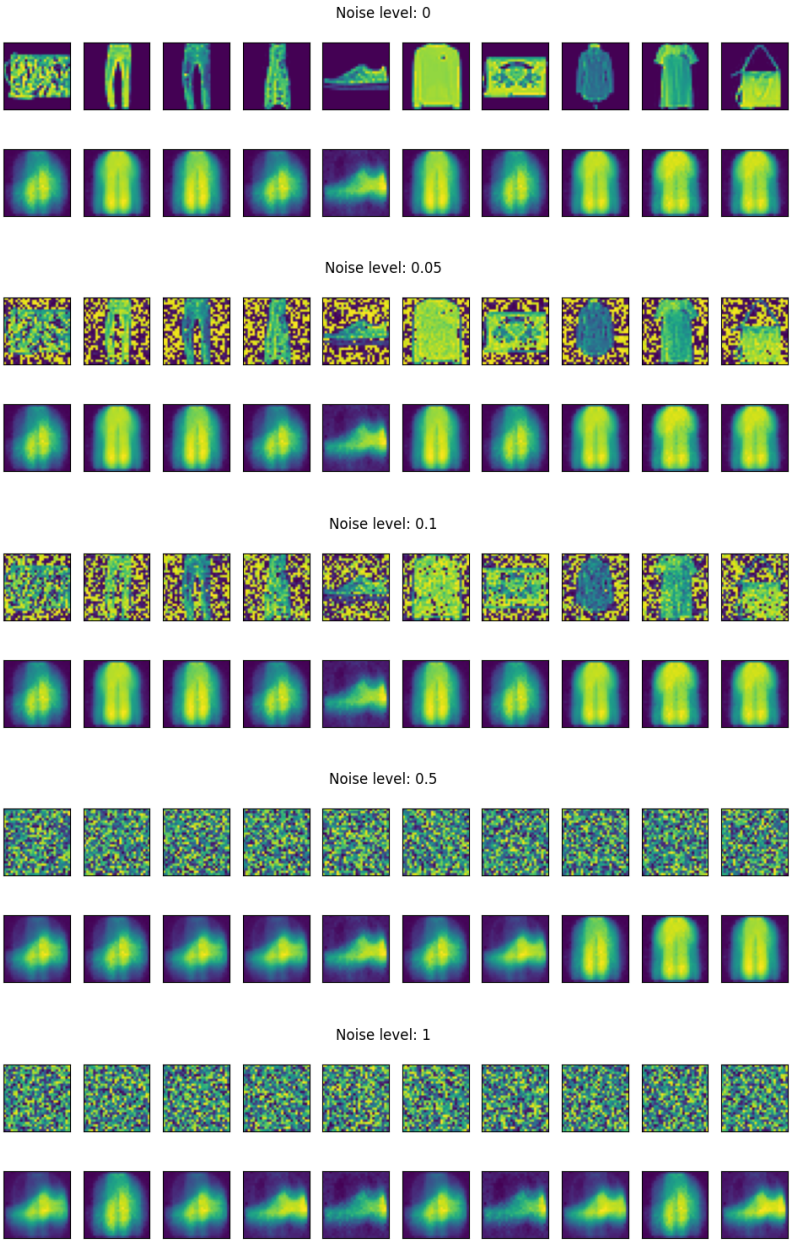

FIGURE 19 – Effet du bruit

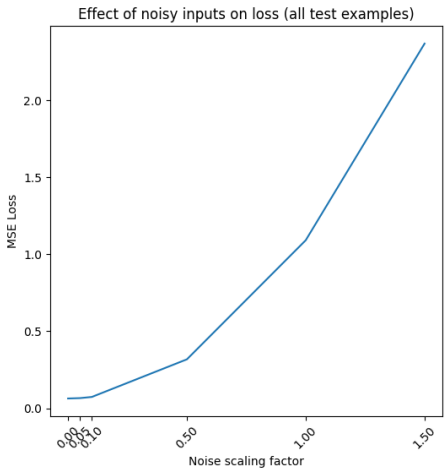

FIGURE 20 – Coût MSE en fonction du facteur d'échelle du bruit

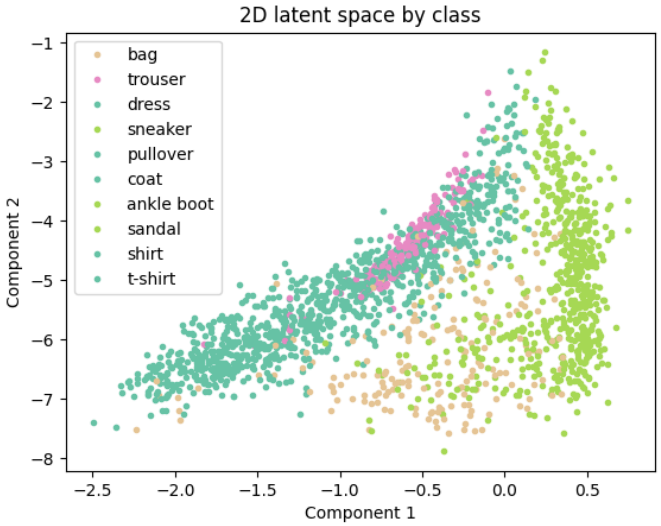

FIGURE 21 – Représentation de l'espace latente des images