Sorbonne Université
Sciences et ingénierie
Master Informatique

Réseau de neurones DIY

Antoine Théologien - 21400184

Parcours :	DAC
Cours :	Machine Learning
Date :	April 1, 2025

1 Introduction

Ce rapport est consacré au projet du cours Machine-Learning et consiste à réaliser une implémentation de réseau de neurones from scratch en Python, en essayant de reproduire au mieux le comportement de la très célèbre bibliothèque PyTorch. Dans celui-ci, les différents détails d'implémentation et les différentes expérimentations seront apportés. Le plan suivra celui de l'énoncé du projet, disponible dans l'archive. Le code source est également disponible dans cette dernière, ainsi que les notebooks contenant les différents tests réalisés.

2 Prémisses

Le projet se base ainsi sur une architecture modulaire, permettant d'assembler les différents éléments d'un réseau, facilitant la mise en place et le déploiement. On dispose ainsi de 2 classes principales qui serviront de classes mères pour la majorité des futures classes implémentées.

2.1 Classe Module

La première est la classe **Module**, qui, comme son nom l'indique, représente un module générique de notre réseau de neurones. Celle-ci contient toutes les méthodes permettant la bonne implémentation d'un réseau de neurones : forward, réinitialisation du gradient, backward, mise à jour des paramètres. Le diagramme de la classe est visible dans la figure 1.

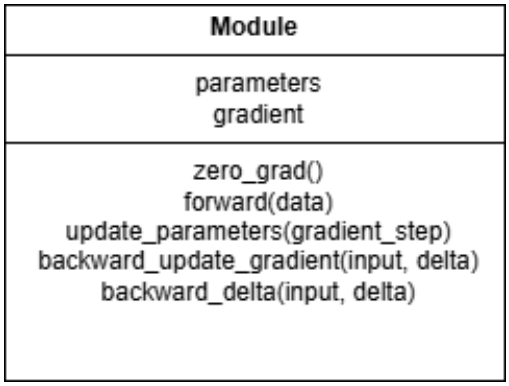

Figure 1: Diagramme de la classe Module

La majorité des fonctions de cette classe ne sont pour l'instant pas implémentées, à l'exception des méthodes *zero_grad* et *update_parameters*. Ainsi, dans la première, on commence par vérifier si la variable gradient n'est pas nulle, afin de s'assurer de pouvoir la manipuler. Si elle existe bien, on met à jour cette variable en mettant un zéro pour chaque paramètre contenu dans notre variable *parameters*. Dans $update_parameters$, on commence par vérifier cette fois-ci si les deux variables de notre classe sont bien définies, puis on met à jour la variable *parameters* en soustrayant le vecteur de la variable *gradient* à celui de nos paramètres, en veillant bien à multiplier celui-ci par le pas de gradient en paramètres. Le reste des méthodes seront implémentées dans les classes filles qui héritent de cette classe abstraite.

2.2 Classe Loss

Cette seconde classe abstraite nous permet donc de disposer des méthodes pour implémenter les différentes fonction de perte que nous souhaiterions utiliser dans notre réseau de neurones. Elle ne dispose que de deux méthodes : $forward$, qui prend en paramètre deux entrées et calcule le coût, et la méthode *backward*, qui va nous permettre de calculer le coup du gradient par rapport à une entrée. Il n'y a donc pas besoin de les implémenter pour l'instant : il faudra le faire spécifiquement en fonction des fonctions de coûts que nous souhaitons mettre en place.

3 Mon premier est ... linéaire !

3.1 Implémentation de la perte MSE

On commence donc par implémenter la fonction de perte Mean Squared Error (MSE), qui est la fonction la plus couramment utilisée pour les régressions linéaires.

$$\text{MSE}(y, \hat{y}) = (y_i - \hat{y}_i)^2$$

où y est le vecteur des valeurs réelles, $\hat{y}$ est le vecteur des prédictions, et n est le nombre d'échantillons. Pour ceci, on commence par déclarer la méthode **forward**, qui va nous permettre de calculer la perte entre les deux vecteurs d'entrée. On commence par vérifier que les deux vecteurs sont bien de la même taille, puis on calcule la somme des carrés des différences entre les deux vecteurs, on pourra ainsi calculer la moyenne dans notre descente de gradient. On récupère ainsi notre perte. On implémente ensuite la méthode backward, qui correspond au calcul de gradient de notre perte, s'exprimant ainsi :

$$\frac{\partial \text{MSE}}{\partial \hat{y}} = -2(y - \hat{y})$$

Cela va nous permettre d'obtenir les valeurs des gradients de notre perte et donc de pouvoir effectuer notre backpropagation.

La classe `MSELoss` hérite de la classe `Loss` et implémente les méthodes `forward` et `backward` pour calculer respectivement la perte et le gradient.

3.2 Implémentation du module linéaire

Le module linéaire, est une brique fondamentale de notre implémentation de réseau de neurones. Celle-ci hérite de la classe **Module**, et possède comme paramètre une matrice de poids de de dimension entrée/sortie. On ajoute une dimension de plus en entrée qui correspond à un vecteur de biais, facilitant l'implémentation de celui-ci. Pour la variable *gradient*, on crée une matrice remplie de zéro de la même taille.

Pour la méthode **forward**, on commence donc par ajouter un vecteur unitaire de biais dans nos données X, grâce à la fonction **hstack**. On effectue ensuite le produit matricielle grâce à l'opérateur @, qui est équivalent à la fonction **np.dot**.

On passe ensuite à la méthode **backward-update-gradient**, dans laquelle on commence par vérifier les bonnes dimensions de nos paramètres, mais aussi par ajouter un vecteur unitaire pour le biais. On ajoute ensuite notre gradient, obtenu par produit matriciel, dans la variable $self.gradient$.

3.3 Expérimentation : Régression linéaire

Avec ces deux éléments, nous pouvons maintenant implémenter une régression linéaire simple. Nous entraînerons un modèle pour prédire une sortie à partir d'une entrée donnée, en minimisant la perte MSE. Les premières expérimentations s'effectuent ainsi sur une simple fonction linéaire de la forme : $y = 3x+4$. Comme pour toutes les expérimentations dans ce projet, nous prenons soin de séparer les données d'entraînement et de test, afin de pouvoir évaluer la performance de notre modèle sur des données non vues. Nous utilisons la fonction **train_test_split** de la bibliothèque **sklearn** pour cela.

On dresse ainsi un graphique montrant la valeur de la perte en fonction du nombre d'itérations, ainsi qu'un second graphique montrant la valeur de la sortie prédite par rapport à la sortie réelle. On peut voir que la perte fini bien par converger, et que les valeurs prédites sont très proches des valeurs réelles. 2.

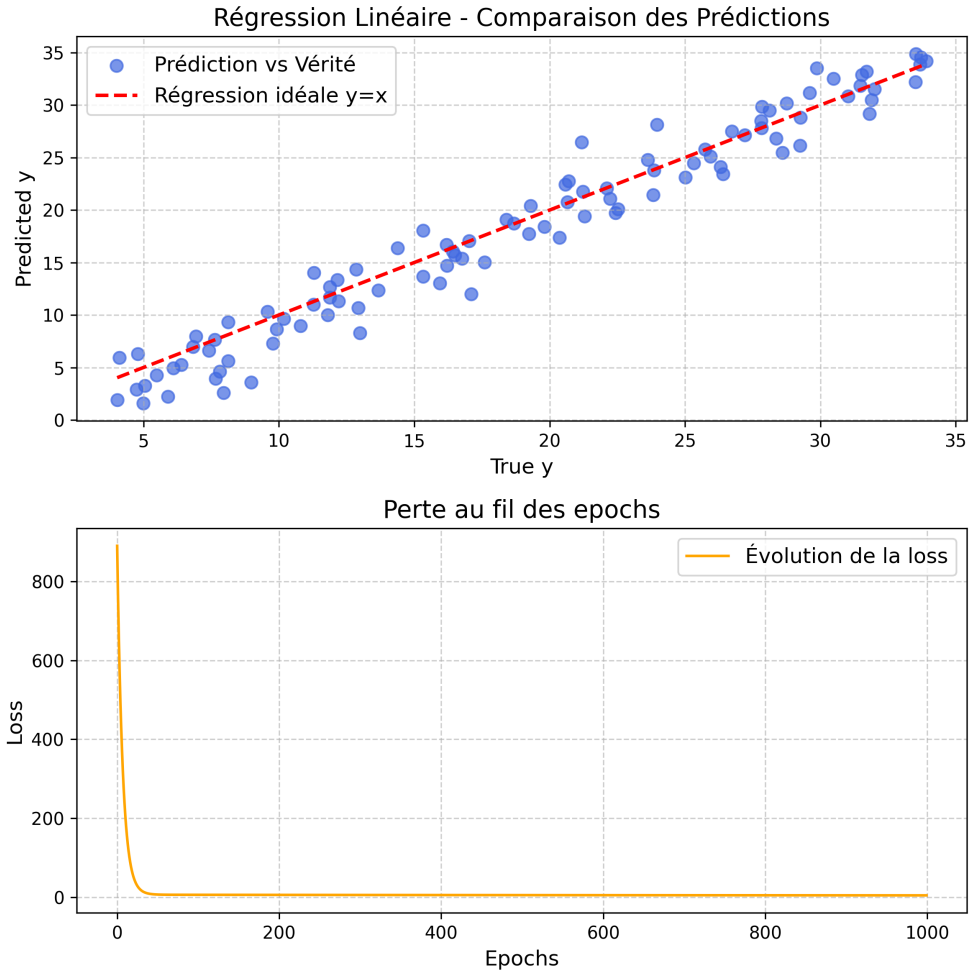

Figure 2: Loss et prédictions de la régression linéaire

On teste ensuite sur différents problèmes de classification, en utilisant la fonction **gen_arti** que nous avons déjà utilisé dans les différents TMEs de l'UE. On a donc trois jeux de données : le premier étant un mélange de deux gaussiennes, le deuxième un mélange de 4 gaussiennes (XOR), et enfin un échiquier. Les résultats de la classification sont visiles dans la figure 3.

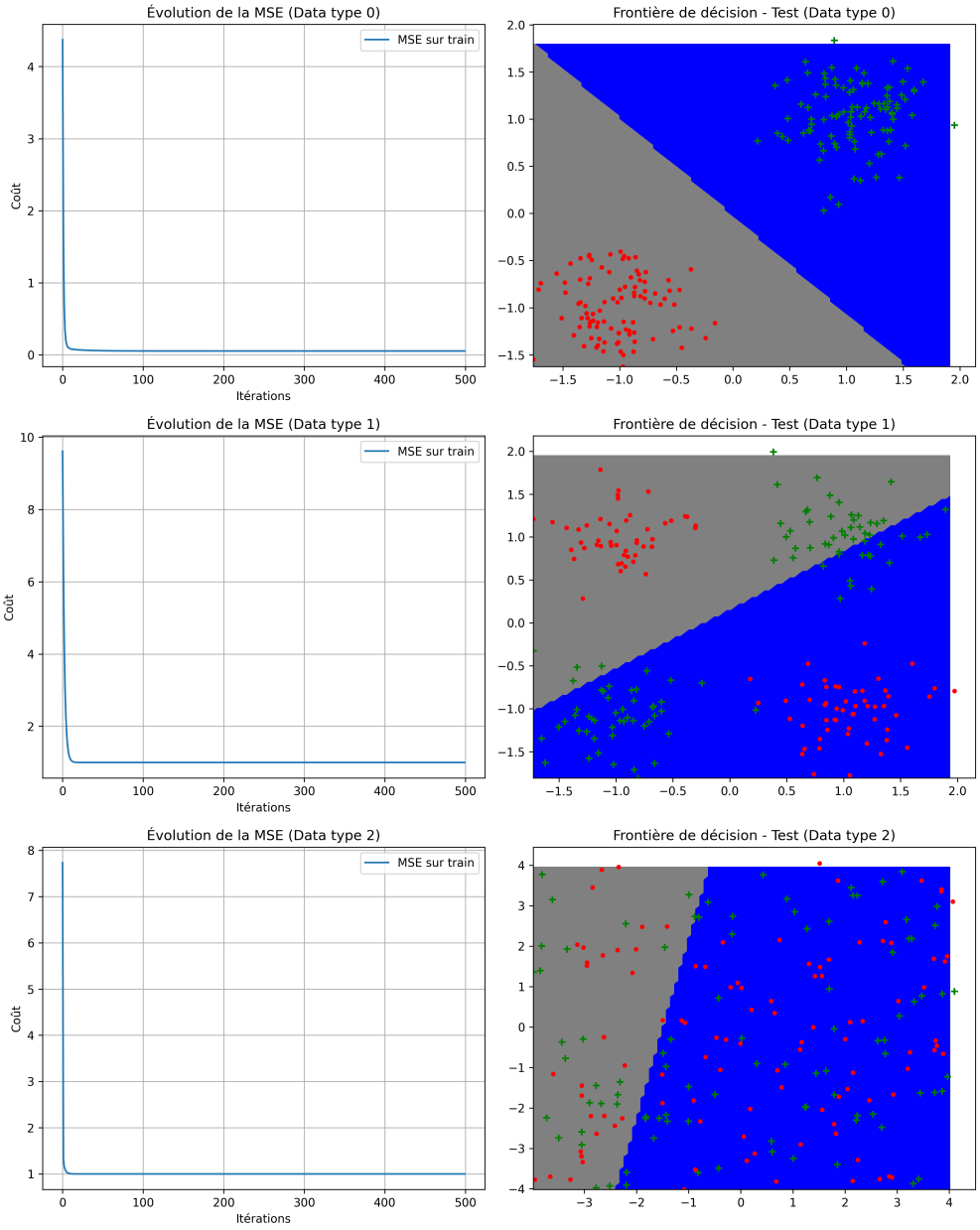

Figure 3: Loss et prédictions de la régression linéaire

On peut donc voir que la régression linéaire fonctionne bien sur la séparation des deux gaussiennes, ce qui est logique puisque la séparation est linéaire. Pour les 4 gaussiennes, la loss converge vers des valeurs beaucoup plus haute, une régression linéaire simple ne pouvant pas suffire à résoudre un tel problème, ce qui est encore plus visible sur notre problème d'échiquier.