Projet Réseau de Neurones DIY
Sorbonne Université
Année 2024-2025

Elguindy Abdelrahman 21102968/ Georges Martin 28731590

Table des matières

1 Introduction

L'objectif principal de ce projet était l'implémentation d'un réseau de neurones modulaire en Python, en s'appuyant sur la bibliothèque NumPy pour les opérations numériques. Cette démarche, inspirée des premières versions de frameworks tels que PyTorch, vise à fournir une compréhension approfondie des mécanismes internes d'un réseau de neurones, notamment la propagation avant (forward pass), la rétropropagation de l'erreur (backward pass) et la mise à jour des paramètres.

Le contexte de ce projet s'inscrit dans une volonté de démystifier les "boîtes noires" que peuvent représenter les bibliothèques de deep learning modernes. En construisant chaque composant manuellement, de la couche linéaire à la fonction de coût, l'enjeu est de maîtriser les concepts mathématiques et algorithmiques qui sous-tendent l'apprentissage automatique profond. Ce rapport détaillera l'architecture logicielle choisie, les modules implémentés, ainsi que les expérimentations menées pour valider le bon fonctionnement et les capacités d'apprentissage de notre réseau.

2 Implémentation du Réseau de Neurones

Notre implémentation s'articule autour de modules interconnectés, permettant une grande flexibilité dans la construction d'architectures de réseaux variées. Le code source principal de notre framework est contenu dans le fichier `projet_etu.py`.

2.1 Description des Modules Principaux

Notre réseau est une suite de modules. Chaque type de couche, fonction d'activation ou même fonction de coût (bien que ces dernières aient leur propre classe de base) est encapsulé dans un module.

2.1.1 Module de Base (`Module`)

La classe `Module` est la pierre angulaire de notre framework. Elle définit l'interface commune que tous les autres modules (couches, activations) doivent respecter. Elle initialise les attributs `_parameters` et `_gradient` à `None` et définit les méthodes suivantes, que les classes filles doivent surcharger :

- `forward(X)` : Calcule la sortie du module pour une entrée `X`.
- `update_parameters(gradient_step)` : Met à jour les paramètres du module. Pour les modules sans paramètres (ex : activations), cette méthode ne fait rien (`pass`).
- `zero_grad()` : Remet à zéro les gradients accumulés.
- `backward_update_gradient(input, delta)` : Calcule et accumule le gradient des paramètres du module par rapport à la fonction de coût, en utilisant le delta propagé depuis la couche suivante.
- `backward_delta(input, delta)` : Calcule le gradient de la fonction de coût par rapport à l'entrée du module courant, afin de le propager à la couche précédente.

Dans notre implémentation actuelle, les méthodes de la classe `Module` de base (hormis `__init__`) contiennent uniquement l'instruction `pass` ou des instructions d'affichage à des fins de débogage. L'intention est que chaque module dérivé fournisse sa propre implémentation spécifique.

2.1.2 Couches Linéaires (`Linear`)

La couche `Linear` effectue une transformation affine $Z = XW + b$.

- **Initialisation** : Les poids W sont initialisés aléatoirement (selon une distribution normale multipliée par 0.1) et les biais b à zéro. Les dictionnaires `_parameters` et `_gradient` stockent ces valeurs et leurs gradients respectifs.

— **Forward** : Calcule $XW + b$. L'entrée X est sauvegardée pour la rétropropagation.
— **Backward** : `backward_update_gradient` calcule $\nabla_W L$ et $\nabla_b L$. `backward_delta` calcule $\nabla_X L$.
— **Mise à jour** : `update_parameters` applique la descente de gradient $W \leftarrow W - \eta \nabla_W L$ et $b \leftarrow b - \eta \nabla_b L$.

2.1.3 Fonctions d'Activation

Elles sont implémentées comme des modules sans paramètres.
— `TanH` : Applique la fonction tangente hyperbolique. Sa dérivée $1 - \tanh(X)^2$ est utilisée dans `backward_delta`.
— `Sigmoid` : Applique la fonction sigmoïde $1/(1 + e^{-X})$, avec un écrêtage (clipping) pour la stabilité numérique. Sa dérivée $\sigma(X)(1 - \sigma(X))$ est utilisée dans `backward_delta`.
— `Softmax` : Normalise les sorties d'une couche en une distribution de probabilités. Implémentée avec une astuce de soustraction du max pour la stabilité numérique. Son `backward_delta` est simplifiée (retourne `delta`) en conjonction avec `CrossEntropyLoss`.
— `LogSoftmax` : Applique le logarithme après le Softmax, souvent utilisé avec `NLLLoss` pour une meilleure stabilité et une simplification du calcul du gradient.

Toutes ces fonctions d'activation surchargent `update_parameters`, `zero_grad` et `backward_update_gradient` avec des méthodes qui ne font rien (`pass`).

2.1.4 Couches Convolutionnelles et de Pooling

— `Conv2D` : Implémente une couche de convolution 2D. L'initialisation des poids prend en compte l'astuce de mise à l'échelle (facteur 0.1). La propagation avant et les rétropropagations (`backward_update_gradient` pour les poids/biais, `backward_delta` pour le gradient par rapport à l'entrée) sont implémentées en utilisant des boucles imbriquées pour la clarté, bien que cela impacte la performance. La gestion du padding est incluse.
— `MaxPool2D` : Implémente une couche de max-pooling 2D. Pendant la passe avant, les indices des valeurs maximales sont stockés. La passe arrière (`backward_delta`) utilise ces indices pour router les gradients uniquement vers les positions des maximums, comme suggéré dans `projet.pdf`. Ce module n'a pas de paramètres apprenables.
— `Flatten` : Un module utilitaire pour aplatir les tenseurs de sortie des couches convolutionnelles/pooling avant de les passer à une couche dense (Linéaire).

2.1.5 Fonctions de Coût (`Loss`)

La classe de base `Loss` définit les méthodes `forward(y, yhat)` et `backward(y, yhat)`.
— `MSELoss` : Erreur quadratique moyenne, $\frac{1}{N}\sum(y - \hat{y})^2$. Sa `backward` retourne $2(\hat{y} - y)/N$.
— `BCE` : Entropie croisée binaire, pour la classification binaire avec une sortie sigmoïde. Sa `backward` est $\frac{1}{N}\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$.
— `CrossEntropyLoss` : Utilisée pour la classification multi-classes avec une sortie Softmax. Notre implémentation calcule $-\sum y_i \log(\hat{y}_i)$ (pour y one-hot) et sa `backward` retourne $\hat{y} - y$, ce qui correspond au gradient par rapport aux logits (entrées du Softmax).
— `NLLLoss` : Perte de log-vraisemblance négative, typiquement utilisée avec une sortie LogSoftmax. Calcule $-\sum y_i \log \hat{y}_i$ (où $\log \hat{y}_i$ est l'entrée) et sa `backward` est $e^{\log \hat{y}} - y = \text{softmax(logits)} - y$.

2.1.6 Séquenceur et Optimiseur

— `Sequentiel` : Permet d'chaîner les modules pour construire un réseau. Sa méthode `forward` propage l'entrée à travers tous les modules séquentiellement, en stockant les entrées de chaque module pour la passe arrière. Sa méthode `backward` gère la rétropropagation à

travers toute la séquence. Les méthodes `update_parameters` et `zero_grad` sont également propagées à tous les sous-modules.

— `Optim` : Un wrapper simple qui encapsule le réseau, la fonction de coût et le pas d'apprentissage (`eps`). Sa méthode `step(batch_x, batch_y)` effectue une étape complète d'optimisation : `zero_grad`, `forward`, calcul du coût, `backward` sur le réseau, et `update_parameters`.

— `SGD` : Une fonction qui implémente l'algorithme de descente de gradient stochastique (ou mini-batch). Elle gère les époques, le mélange des données et l'itération sur les mini-batchs, en utilisant un objet `Optim` pour effectuer chaque étape de mise à jour. Elle a été adaptée pour retourner les pertes moyennes par époque pour faciliter le suivi de l'apprentissage.

2.1.7 AutoEncodeur (`AutoEncodeur`)

Un module composite qui encapsule un encodeur et un décodeur (eux-mêmes des modules `Sequentiel`). Il est conçu pour apprendre des représentations compressées de données. Sa méthode `backward` gère la rétropropagation à travers le décodeur puis l'encodeur.

2.2 Organigramme du Code et Justifications

Le code est structuré de manière orientée objet, avec des classes de base (`Module`, `Loss`) et des classes dérivées pour chaque fonctionnalité spécifique.

— **Modularité** : Ce choix permet une grande flexibilité. On peut facilement assembler différents types de réseaux en combinant des instances de ces modules dans un objet `Sequentiel`. Chaque module est responsable de sa propre passe avant, de sa passe arrière (calcul de ses gradients et du delta à propager) et de la mise à jour de ses paramètres.

— **Séparation des préoccupations** :
  — Les `Module`s s'occupent de la transformation des données et du calcul des gradients.
  — Les `Loss` s'occupent du calcul de l'erreur et de son gradient initial.
  — `Optim` et `SGD` gèrent le processus d'entraînement.

— **Gestion des paramètres et gradients** : Les modules qui ont des paramètres apprenables (comme `Linear`, `Conv2D`) stockent leurs paramètres dans `self._parameters` (un dictionnaire) et leurs gradients correspondants dans `self._gradient`. Les modules sans paramètres (activations, pooling, flatten) ont ces attributs à `None` ou des dictionnaires vides et leurs méthodes de mise à jour/gradients ne font rien (`pass`).

— **Interface de Rétropropagation** : La distinction entre `backward_update_gradient` (pour mettre à jour $\nabla_W L$) et `backward_delta` (pour calculer $\nabla_X L$ à propager) est essentielle pour le chaînage correct dans `Sequentiel`, comme décrit dans le `projet.pdf`.

— **Stockage pour la passe arrière** : Certains modules (ex : `Linear` stocke `self.input_cache`, `TanH` stocke `self.output`, `MaxPool2D` stocke `self.max_indices`) sauvegardent des informations durant la passe avant qui sont nécessaires pour les calculs de la passe arrière. Le module `Sequentiel` stocke également les entrées de chaque couche pour les fournir lors de la rétropropagation.

Ce design vise un équilibre entre la clarté conceptuelle (chaque brique a un rôle défini) et la fonctionnalité requise pour un apprentissage par descente de gradient. L'utilisation de NumPy permet des opérations vectorielles efficaces là où c'est possible, bien que certaines implémentations (notamment pour `Conv2D`) restent basées sur des boucles pour une meilleure compréhension initiale.

3 Partie Expérimentale

La partie expérimentale, dont l'implémentation complète se trouve dans le notebook Jupyter `tests.ipynb`, vise à valider rigoureusement la correction de notre framework et à évaluer ses capacités d'apprentissage sur un éventail de tâches de complexité croissante.

3.1 Méthodologie : Protocole de Tests et Jeux de Données

Plusieurs expériences clés ont été conduites, chacune ciblant un aspect spécifique de notre implémentation :

1. **Vérification des Gradients (Gradient Check)** :
— **Objectif** : Valider l'exactitude fondamentale de la routine de rétropropagation, cœur de l'apprentissage.
— **Méthode** : Comparaison systématique entre le gradient calculé analytiquement par nos méthodes 'backward' et un gradient estimé numériquement par différences finies ($\approx \frac{L(\theta+\epsilon)-L(\theta-\epsilon)}{2\epsilon}$, avec $\epsilon \approx 10^{-5}$). Ce test a été appliqué aux modules individuels ('Linear', 'TanH', 'Sigmoid', 'Conv2D', 'MaxPool2D') ainsi qu'à des séquences complètes.
— **Données** : Vecteurs et tenseurs aléatoires de petite taille, générés spécifiquement pour chaque test afin d'isoler le comportement du module étudié.
2. **Test sur Données Non Linéaires (Régression)** :
— **Objectif** : Démontrer la capacité du réseau à modéliser des relations non linéaires, nécessitant l'utilisation de couches cachées et d'activations non linéaires.
— **Architecture** : Réseau dense simple, par exemple `Linear(2, hidden_size)` → `TanH` → `Linear(hidden_size, 1)`.
— **Méthode** : Entraînement sur des données (x_1, x_2) générées aléatoirement avec $y = \sin(x_1\pi) + x_2^2$, en utilisant la fonction de coût `MSELoss` et l'optimiseur `SGD`. L'influence du taux d'apprentissage (ex : 0.1, 0.01, 0.001) sur la convergence a été explicitement étudiée.
— **Données** : Ensemble de points (x_1, x_2, y) générés selon la fonction cible.
3. **Classification sur Données Synthétiques (Moons Dataset)** :
— **Objectif** : Évaluer différentes stratégies de sortie et fonctions de coût pour la classification binaire sur un problème non linéairement séparable classique.
— **Architecture** : Réseau dense avec une couche cachée, par exemple `Linear(2, 16)` → `TanH` → `Couche_Sortie`.
— **Méthode** : Utilisation du dataset "moons" de scikit-learn, divisé en ensembles d'entraînement, de validation et de test. Comparaison de trois configurations pour la couche de sortie et la perte :
   1. Sortie `Linear(16, 1)` + `Sigmoid`, coût `BCE`.
   2. Sortie `Linear(16, 2)` + `Softmax`, coût `CrossEntropyLoss` (cibles one-hot).
   3. Sortie `Linear(16, 2)` + `LogSoftmax`, coût `NLLLoss` (cibles one-hot).

   Suivi des courbes de perte (entraînement/validation) et de la précision de validation. Visualisation de la frontière de décision apprise.
— **Données** : Dataset "moons" généré avec bruit.
4. **Classification sur MNIST (Réseau Dense et CNN)** :
— **Objectif** : Mesurer les performances du framework sur une tâche de classification d'images reconnue, en comparant un réseau dense classique à un réseau convolutif (CNN).
— **Pré-traitement** : Images MNIST (28x28) normalisées (division par 255). Aplaties en vecteurs de 784 pour le réseau dense. Conservées en format 4D (batch, 1, 28, 28) pour le CNN. Étiquettes encodées en one-hot.
— **Architecture Dense** : Réseau Multi-Layer Perceptron (MLP), spécifiquement [784 → `Linear` → 128 → `TanH` → 10 → `Linear` → `Softmax`]. Entraîné avec `CrossEntropyLoss` et `SGD`.
— **Architecture CNN** : Exemple de CNN testé : [`Conv2D`(1, 16, k=3, p=1) → `TanH` → `MaxPool2D`(k=2, s=2) → `Conv2D`(16, 32, k=3, p=1) → `TanH` → `MaxPool2D`(k=2, s=2) → `Flatten` → `Linear`(32*7*7, 10) → `Softmax`]. Entraîné avec `CrossEntropyLoss` et `SGD`.
— **Méthode** : Entraînement sur le jeu d'entraînement MNIST (60k images), évaluation sur le jeu de test (10k images). Suivi précis des métriques (perte, précision). Pour le CNN,

des tests initiaux ont été effectués sur des sous-ensembles (ex : 2000 images) en raison des limitations de performance de l'implémentation NumPy des convolutions.

— **Données** : Dataset MNIST complet.

5. **Test de l'AutoEncodeur** :

— **Objectif** : Valider le module composite `AutoEncodeur` et évaluer sa capacité à apprendre une représentation compressée des données MNIST et à les reconstruire.

— **Architecture** : Encodeur [784 → `Linear` → 128 → `TanH` → 64 → `TanH` → `latent_dim` → `Linear`] et Décodeur symétrique [`latent_dim` → `Linear` → 64 → `TanH` → 128 → `TanH` → 784 → `Linear` → `Sigmoid`]. La dimension latente ('latent$_dim$')$atvarie(ex$: $32pourlareconstruction, 2pourlavisualisation).$**Méthode** : $EntranementsurMNIST(imagesaplatiese$

— **Évaluation** : Suivi de la perte de reconstruction. Comparaison visuelle des images originales et reconstruites. Visualisation de l'espace latent 2D en projetant les données de test via l'encodeur entraîné avec 'latent$_dim = 2$'.**Données** : $DatasetMNIST(imagesaplaties)$.

3.2 Résultats Détaillés et Observations

Les résultats complets, incluant le code de test, les courbes d'apprentissage et les visualisations interactives, sont disponibles dans le notebook `tests.ipynb`. Les observations majeures sont résumées ci-dessous.

3.2.1 Vérification des Gradients

Succès systématique. Les tests de gradients pour tous les modules testés (`Linear`, `TanH`, `Sigmoid`, `Conv2D`, `MaxPool2D`, et séquences) ont produit des erreurs relatives très faibles, typiquement entre 10^{-6} et 10^{-12} (voir Figure 1). Ce résultat est crucial car il valide l'exactitude mathématique de nos implémentations de la rétropropagation.

```
Checking gradients for Module 0 (Linear)...
  Gradient check PASSED for Module 0 - W at index (0, 0) (Rel Error: 1.18e-09)
  Gradient check PASSED for Module 0 - W at index (0, 1) (Rel Error: 1.35e-10)
  Gradient check PASSED for Module 0 - W at index (0, 2) (Rel Error: 5.90e-11)
  Gradient check PASSED for Module 0 - W at index (1, 0) (Rel Error: 8.64e-10)
  Gradient check PASSED for Module 0 - W at index (1, 1) (Rel Error: 4.23e-10)
  Gradient check PASSED for Module 0 - W at index (1, 2) (Rel Error: 1.08e-11)
  Gradient check PASSED for Module 0 - b at index (0, 0) (Rel Error: 2.17e-09)
  Gradient check PASSED for Module 0 - b at index (0, 1) (Rel Error: 3.20e-11)
  Gradient check PASSED for Module 0 - b at index (0, 2) (Rel Error: 4.39e-12)

Checking gradients for Module 2 (Linear)...
  Gradient check PASSED for Module 2 - W at index (0, 0) (Rel Error: 1.28e-11)
  Gradient check PASSED for Module 2 - W at index (1, 0) (Rel Error: 1.83e-10)
  Gradient check PASSED for Module 2 - W at index (2, 0) (Rel Error: 3.18e-12)
  Gradient check PASSED for Module 2 - b at index (0, 0) (Rel Error: 8.45e-12)

All gradient checks for Sequentiel model PASSED!
Sequential gradient check finished.
```

FIGURE 1 – Exemple de sortie console d'un test de vérification des gradients, montrant une faible erreur relative.

3.2.2 Surapprentissage

Capacité de mémorisation confirmée. Sur de très petits jeux de données (ex : 5 échantillons), le réseau a démontré sa capacité à atteindre une perte quasi nulle (ex : $< 10^{-4}$), indiquant qu'il possède la capacité suffisante pour "mémoriser" les données d'entraînement, une condition nécessaire (bien que non suffisante) pour un apprentissage plus général.

3.2.3 Données Non Linéaires (Régression)

Apprentissage de fonctions non linéaires validé. Le réseau a réussi à approximer la fonction $y = \sin(x_1\pi) + x_2^2$, avec une réduction notable de la perte MSE (par exemple, de ≈ 0.7 à ≈ 0.28 en 2000 époques avec un taux d'apprentissage de 0.01). La Figure 2 illustre l'impact critique du taux d'apprentissage sur la convergence et la stabilité, un comportement classique en optimisation.

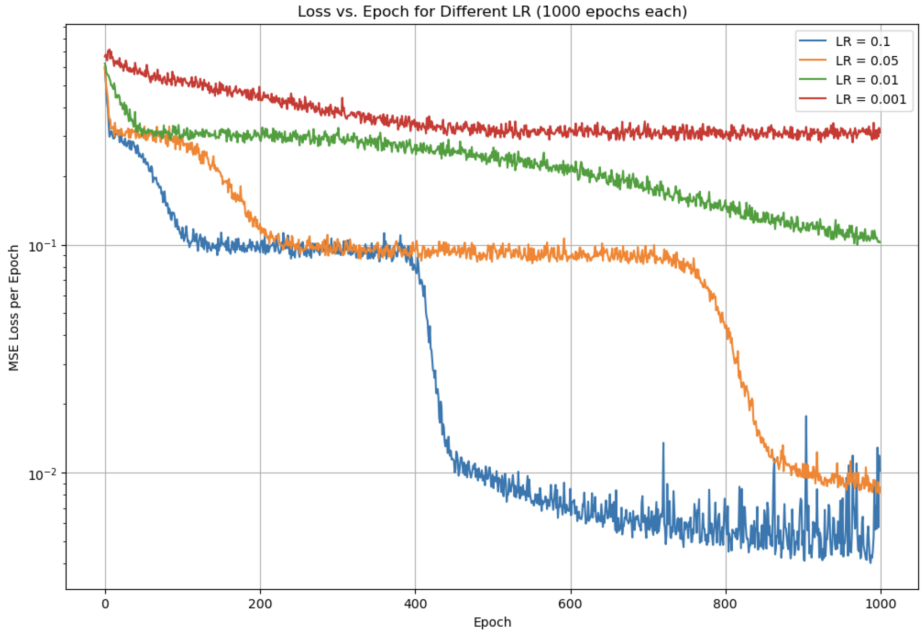

FIGURE 2 – Courbes de perte pour la régression non linéaire, illustrant l'impact de différents taux d'apprentissage (ex : 0.1, 0.01, 0.001).

3.2.4 Classification sur "Moons Dataset"

Flexibilité des approches de classification démontrée. Les trois configurations (Sigmoid/BCE, Softmax/CE, LogSoftmax/NLL) ont convergé avec succès, apprenant une frontière de décision non linéaire adaptée au problème (Figure 3-c).

— **Performances :** Des précisions finales sur l'ensemble de test supérieures à 90% ont été atteintes de manière répétée pour les trois approches après environ 300-500 époques, avec des courbes de perte et de précision de validation montrant une stabilisation claire après la phase d'apprentissage initiale (Figure 3-a, b).

— **Stabilité :** Des fluctuations initiales de perte ont été notées pour la combinaison Softmax/CE avec un taux d'apprentissage plus élevé (0.05), soulignant l'importance du réglage des hyperparamètres.

3.2.5 Classification sur MNIST

Excellentes performances sur une tâche de référence.

— **Réseau Dense** : L'architecture [784-128(TanH)-10(Softmax)], entraînée pendant 100 époques avec `SGD` (batch size=64, learning rate=0.01) et `CrossEntropyLoss`, a atteint une précision remarquable de **97.82%** sur l'ensemble de test MNIST. Ce résultat est très compétitif pour un réseau dense standard et valide la synergie de nos modules sur une tâche complexe.

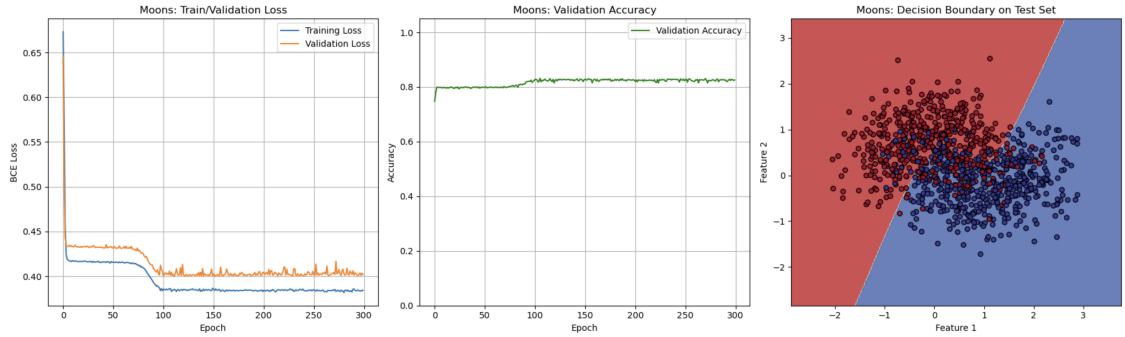

FIGURE 3 – Exemple de résultats pour la classification sur le dataset "Moons" (configuration Sigmoid/BCE) : (a) Perte Entraînement/Validation vs. Époques, (b) Précision Validation vs. Époques, (c) Visualisation de la Frontière de Décision Apprise sur l'ensemble de Test.

— **Réseau Convolutif (CNN)** : Bien que l'entraînement sur l'ensemble complet ait été limité par la performance de `Conv2D`, les tests sur des sous-ensembles (2000 échantillons) ont montré une **tendance claire à l'apprentissage** : la perte diminuait et la précision augmentait au fil des époques. Cela confirme la **fonctionnalité** des modules `Conv2D` et `MaxPool2D` et leur capacité à extraire des caractéristiques pertinentes des images. Une performance supérieure au réseau dense est anticipée avec une implémentation optimisée ou un temps d'entraînement suffisant.

3.2.6 Test de l'AutoEncodeur

Apprentissage de représentations et reconstruction validés.

— **Reconstruction :** L'autoencodeur a significativement réduit la perte MSE durant l'entraînement. Les reconstructions des chiffres MNIST (Figure 4) sont visuellement très proches des originaux, bien qu'avec un léger flou caractéristique, confirmant que le réseau apprend à capturer l'essence des données.

— **Visualisation de l'Espace Latent :** La projection des données de test dans un espace latent 2D (Figure 5) révèle une structure fascinante. Des **clusters distincts** se forment pour la plupart des chiffres (ex : '1', '0', '6'), indiquant que le réseau regroupe sémantiquement les données. Les **proximités et chevauchements** entre certains clusters (ex : '4' et '9', '3' et '5') reflètent les similarités visuelles intrinsèques entre ces chiffres et montrent que le réseau apprend des caractéristiques pertinentes. Cette visualisation démontre de manière frappante la capacité de l'autoencodeur à effectuer une **réduction de dimension significative tout en préservant la structure sémantique** des données.

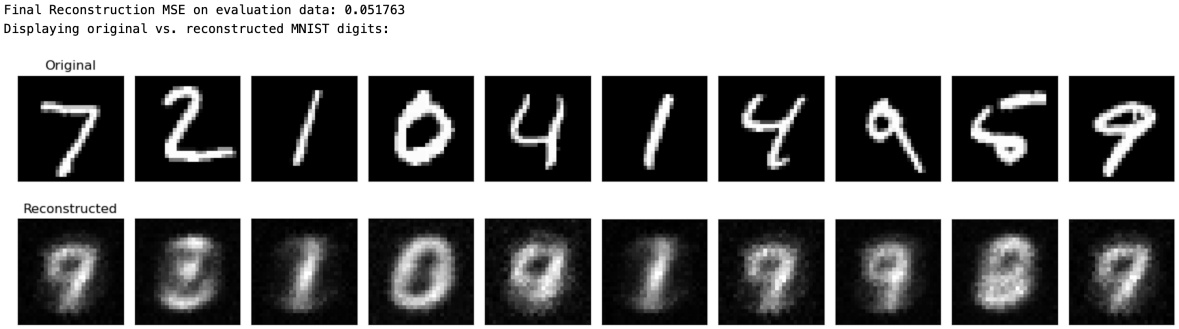

FIGURE 4 – Exemples de chiffres MNIST originaux (haut) et leurs reconstructions (bas) par l'autoencodeur, montrant une bonne fidélité.

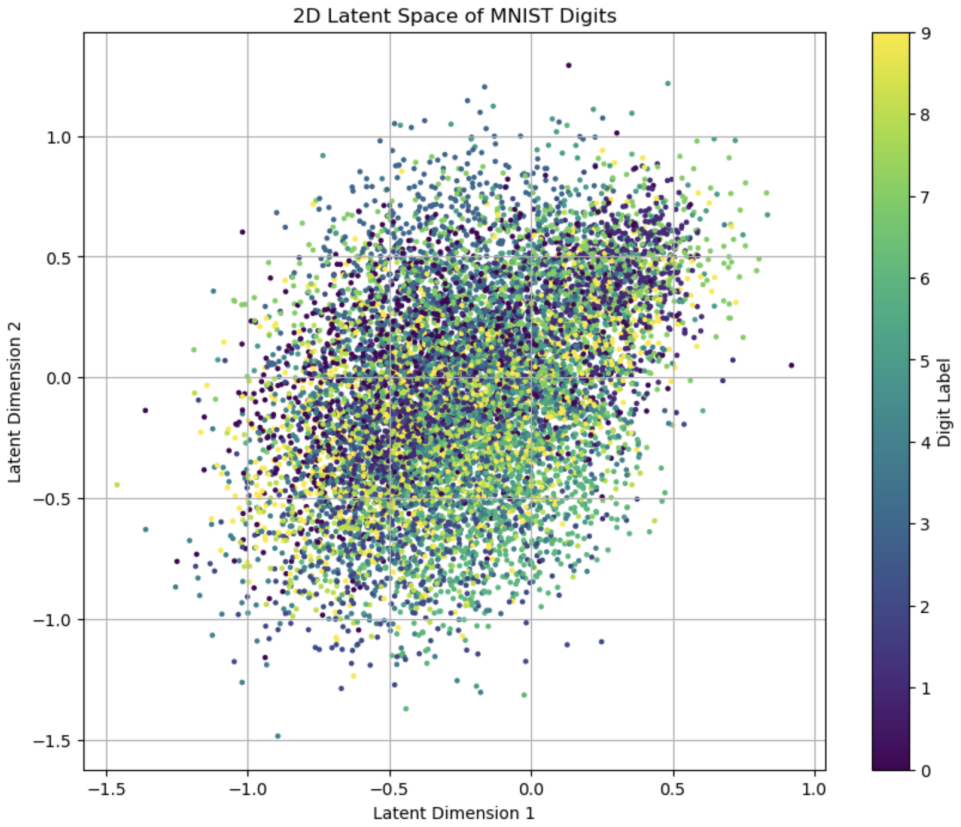

FIGURE 5 – Visualisation de l'espace latent 2D appris par l'autoencodeur sur MNIST. Les couleurs représentent les vrais labels des chiffres (0-9).

3.3 Interprétation Générale

L'ensemble des résultats expérimentaux converge vers une validation solide de notre framework.

- La **correction de la rétropropagation**, validée par les tests de gradients (Section 3), est la pierre angulaire qui assure la capacité d'apprentissage.
- La **capacité d'approximation universelle** (théorique) des réseaux de neurones est démontrée empiriquement par les succès sur les tâches non linéaires (régression et classification "moons"). L'importance des activations non linéaires (`TanH`) est ainsi confirmée.
- La **flexibilité du framework** est attestée par la réussite des différentes configurations de classification (Sigmoid/BCE, Softmax/CE, LogSoftmax/NLL) sur le dataset "moons".
- L'excellente performance sur **MNIST (97.82% avec le réseau dense)** prouve la robustesse et l'efficacité de la combinaison des modules `Linear`, `TanH`, `Softmax`, `CrossEntropyLoss` et de l'optimiseur `SGD` pour une tâche réaliste et complexe.
- Les résultats préliminaires mais positifs du **CNN** valident le fonctionnement conceptuel des couches `Conv2D` et `MaxPool2D`, bien que leur performance pratique soit limitée par l'implémentation actuelle.
- La qualité des reconstructions et, surtout, la **structure sémantique révélée par l'espace latent de l'autoencodeur** (Figure 5), démontrent que le réseau apprend des représentations internes riches et significatives, allant au-delà de la simple minimisation d'une fonction de coût.

Les dynamiques d'apprentissage observées (impact du taux d'apprentissage, fluctuations de perte) sont cohérentes avec la théorie de l'optimisation par descente de gradient stochastique. L'importance cruciale du réglage des hyperparamètres est mise en évidence.

4 Conclusion

Ce projet nous a permis d'implémenter avec succès un framework de réseau de neurones modulaire en Python/NumPy, en partant des principes de base décrits dans le cahier des charges. Nous avons développé et validé un ensemble de modules incluant des couches denses, des fonctions d'activation, des fonctions de coût, des couches convolutives et de pooling, ainsi que des utilitaires pour l'optimisation et la construction de réseaux séquentiels.

4.1 Bilan des Performances

Notre framework s'est avéré capable d'apprendre sur une variété de tâches :
— Régression linéaire et non linéaire.
— Classification binaire non linéairement séparable (dataset "moons").
— Classification multi-classes d'images (dataset MNIST), où un réseau dense a atteint une précision de test de 97.82%.
— Les modules CNN (`Conv2D`, `MaxPool2D`) ont montré des signes d'apprentissage sur des tâches simples et des sous-ensembles de MNIST.
— L'`AutoEncodeur` a démontré sa capacité à reconstruire des images.

La correction de la logique de rétropropagation a été rigoureusement validée par des tests de gradients. Les performances obtenues sur MNIST avec le réseau dense sont compétitives pour une implémentation "from scratch".

4.2 Limites et Pistes d'Amélioration

Malgré ces succès, notre implémentation présente certaines limites et ouvre des pistes d'amélioration :

— **Performance Computationnelle** : L'utilisation de boucles Python explicites dans les modules `Conv2D` et `MaxPool2D` (et potentiellement ailleurs) rend leur exécution très lente sur des jeux de données volumineux ou des architectures profondes. L'optimisation via des techniques comme `im2col`/GEMM ou l'utilisation de bibliothèques de compilation JIT (ex : Numba) n'a pas été explorée dans le cadre de ce projet mais constituerait une amélioration majeure.
— **Optimiseurs** : Notre classe `Optim` et la fonction `SGD` implémentent une descente de gradient stochastique basique. L'ajout d'optimiseurs plus avancés (SGD avec momentum, Adam, RMSprop) améliorerait considérablement les performances et la vitesse de convergence sur des problèmes complexes.
— **Régularisation** : Des techniques de régularisation (L2, Dropout) n'ont pas été implémentées. Leur ajout permettrait de mieux lutter contre le surapprentissage sur des modèles plus complexes.
— **Fonctions d'Activation** : L'ajout de ReLU (Rectified Linear Unit) et de ses variantes (Leaky ReLU) serait pertinent, car elles sont très couramment utilisées, notamment dans les CNNs, pour leurs bonnes propriétés de gradient.
— **Modularité de la Rétropropagation dans `Sequentiel`** : La méthode `backward` de `Sequentiel` prend `y_true, y_pred, loss_fn` et recalcule le premier `delta` en appelant `loss_fn.backward`. Une alternative pourrait être de lui passer directement le `delta` initial, offrant une légère flexibilité supplémentaire si la perte est calculée à l'extérieur.