
Sorbonne Université

# Rapport de projet Machine Learning: Réseaux de Neurones DIY

***Etudiants :***
Yacine Chettab
Louis Delsaer

***Professeur :***
Nicolas BASKIOTIS

2024-2025

# Introduction

Dans le cadre de ce projet, nous nous sommes donné pour objectif de concevoir et d'implémenter, en langage Python, une bibliothèque modulaire de réseaux de neurones artificiels, inspirée dans sa philosophie des premières versions de PyTorch. L'architecture de cette bibliothèque repose sur une abstraction rigoureuse des composants fondamentaux du calcul neuronal : chaque opération — qu'il s'agisse d'une transformation linéaire, d'une fonction d'activation ou d'un calcul de perte — est encapsulée dans un module autonome, capable de réaliser la passe avant, de calculer et de stocker les gradients lors de la rétro-propagation, puis de mettre à jour ses paramètres au moyen d'un optimiseur générique.

Le développement s'est structuré en plusieurs phases progressives. Nous avons commencé par l'implémentation d'un modèle de régression linéaire, fondé sur une couche linéaire (Linear) et la fonction de coût des moindres carrés (MSE). À cela ont été ajoutées des fonctions d'activation non linéaires telles que Tanh et Sigmoïde, ainsi qu'une structure séquentielle (Sequential) permettant de chaîner dynamiquement les modules. Par la suite, l'implémentation d'un optimiseur générique et l'introduction du cadre multi-classe — via la combinaison Softmax et entropie croisée — ont permis d'élargir le spectre des tâches d'apprentissage supervisé abordables par notre bibliothèque.

Enfin, une partie substantielle du projet a été consacrée à l'étude des auto-encodeurs, tant pour leur capacité à reconstruire les entrées que pour leur potentiel en matière de réduction de dimension et de structuration des représentations latentes. Ces expériences ont offert un terrain propice à l'analyse de diverses stratégies de régularisation, de visualisation des embeddings et d'exploration de leur réutilisation comme prétraitement dans des tâches de classification supervisée.

# Architecture logicielle et organisation du code

Le projet s'organise autour d'un répertoire principal `src`, structuré en plusieurs sous-dossiers dédiés respectivement à l'implémentation des modules, aux tests, à la sauvegarde des modèles et aux expérimentations via des notebooks Jupyter. Cette architecture favorise la modularité du code, la réplicabilité des expériences et une séparation claire des responsabilités (logique métier, validation, visualisation).

Table 1: Organisation des fichiers du projet

| **Fichier / Dossier** | **Description** |
|---|---|
| `src/modules/` | Implémentation des composants de base : couches linéaires, fonctions d'activation, fonctions de coût, structure séquentielle. |
| `src/test/` | Scripts de tests unitaires pour vérifier le bon fonctionnement des modules grâce à *torch*. |
| `src/models/` | Modèles entraînés (`.pkl`), erreur d'apprentissage et de teste (`.csv`) et des visualisations (`.png`). |
| `2_layers_NN.ipynb` | Réseau simple (2 couches linéaires, *Sigmoid* et $MSE$) testé sur un jeu synthétique (`dots`). |
| `encapsulage.ipynb` | Tests de la classe `Sequential` et de l'optimiseur. |
| `multi_class.ipynb` | Classification multi-classes sur les jeux de données: Iris, Fashion-MNIST et Kuzushiji-MNIST. |
| `autoencodeur.ipynb` | Entraînement et évaluation d'auto-encodeurs sur Fashion-MNIST. |
| `pretraitement.ipynb` | Comparaison entre classification brute et classification après reconstruction. |
| `visualisation.ipynb` | Clustering des représentations latentes produites par les auto-encodeurs. |

# Expériences et résultats

## Expérience 1 : Classification multi-classe

Cette première série d'expérimentations vise à évaluer les performances de notre bibliothèque sur des tâches de classification supervisée à classes multiples. Trois jeux de données ont été mobilisés : *Iris* (structure tabulaire), *Fashion-MNIST* et *Kuzushiji-MNIST* (images en niveaux de gris). L'objectif est d'évaluer différentes architectures, en variant notamment les fonctions d'activation et la fonction de la perte *Cross Entropie* suite à une activation $Softmax$.

### Jeu de données *Iris*

Le jeu *Iris* comporte 150 échantillons répartis en 3 classes, chaque échantillon étant décrit par 4 variables numériques. Une visualisation tridimensionnelle permet d'anticiper une certaine séparabilité entre les classes. Le modèle utilisé est composé de trois couches linéaires séparées par des activations `Tanh` :

$$\texttt{modèle 1} : \texttt{Linear}(4, 10) \to \texttt{Tanh} \to \texttt{Linear}(10, 4) \to \texttt{Tanh} \to \texttt{Linear}(4, 3)$$

Les performances témoignent d'une bonne convergence de la fonction de coût, avec une classification précise, bien que potentiellement sujette au sur-apprentissage compte tenu de la petite taille du jeu.

### Jeu de données *Fashion-MNIST*

Le jeu *Fashion-MNIST* (60 000 échantillons d'apprentissage, 10 000 de test, répartis en 10 classes), deux architectures ont été expérimentées :

$$\texttt{modèle 2} : \texttt{Linear}(784, 176) \to \texttt{Tanh} \to \texttt{Linear}(176, 42) \to \texttt{Tanh} \to \texttt{Linear}(42, 10)$$
$$\texttt{modèle 3} : \texttt{Linear}(784, 176) \to \texttt{Tanh} \to \texttt{Linear}(176, 42) \to \texttt{ReLU} \to \texttt{Linear}(42, 10)$$

Les deux modèles convergent correctement, la version avec `ReLU` en profondeur offrant de légères améliorations en termes de précision et de stabilité d'apprentissage.

### Jeu de données *Kuzushiji-MNIST*

Ce jeu reprend la structure du *Fashion-MNIST* (images en $28 \times 28$, 10 classes), mais propose des caractères japonais manuscrits plus difficiles à distinguer. Le modèle adopté

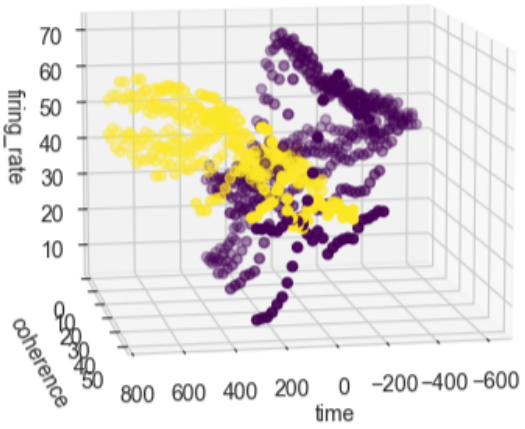



Figure 1: Visualisation en 3D du jeu de données Iris réparties en deux classes

ajoute une activation finale supplémentaire pour augmenter la non-linéarité :

`modèle 4` : $\texttt{Linear}(784, 176) \to \texttt{Tanh} \to \texttt{Linear}(176, 42) \to \texttt{ReLU} \to \texttt{Linear}(42, 10) \to \texttt{ReLU}$

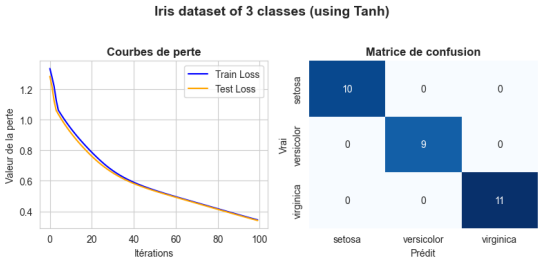



Figure 2: Résultats du modèle 1 (Iris)

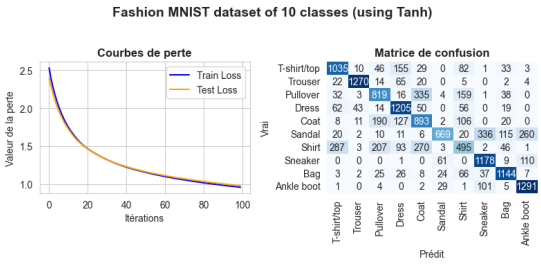



Figure 3: Résultats du modèle 2 (Fashion-MNIST)

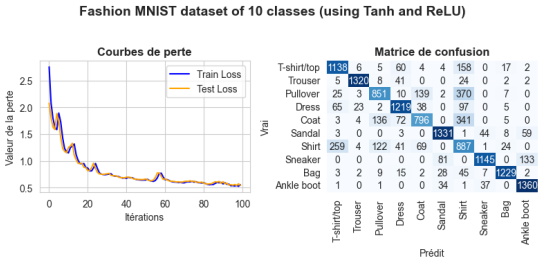



Figure 4: Résultats du modèle 3 (Fashion-MNIST)

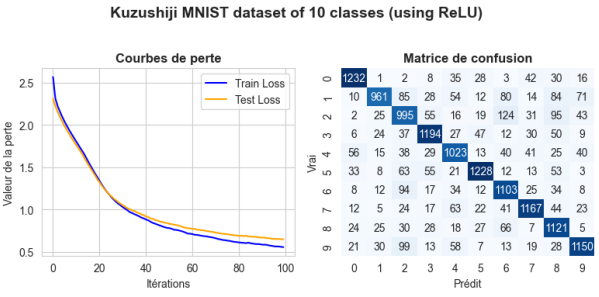



Figure 5: Résultats du modèle 4 (Kuzushiji-MNIST)

Les résultats restent satisfaisants et l'apprentissage se fait d'une manière plus lisse et rapide grâce à la dérivation simple de l'activation `ReLU`

## Résultats

Table 2: Comparaison des performances des modèles sur les différents jeux de données

| **Modèle** | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **Jeu de données** | Iris | Fashion-MNIST (Tanh) | Fashion-MNIST (ReLU) | Kuzushiji-MNIST |
| **Dimensions (entrée → sortie)** | $4 \to 3$ | $784 \to 10$ | $784 \to 10$ | $784 \to 10$ |
| **Nb d'exemples (train/test)** | 120/30 | 48 000/12 000 | 48 000/12 000 | 48 000/12 000 |
| **Nb d'itérations** | 1 000 | 100 | 100 | 100 |
| **Taux d'apprentissage** | 0,01 | 0,01 | 0,1 | 0,1 |
| **Train Loss** | 0,29 | 0,90 | 0,55 | 0,55 |
| **Test Loss** | 0,33 | 0,92 | 0,53 | 0,64 |
| **Accuracy** | 0,90 | 0,72 | 0,81 | 0,80 |
| **F1-score** | 0,90 | 0,72 | 0,81 | 0,80 |
| **Précision** | 0,90 | 0,71 | 0,82 | 0,80 |
| **Rappel** | 0,90 | 0,82 | 0,80 | 0,80 |

# Expérience 2 : Reconstruction d'images avec un auto-encodeur

Dans cette seconde série d'expérimentations, nous explorons les capacités de reconstruction d'images de plusieurs auto-encodeurs symétriques entraînés sur le jeu *Fashion-MNIST*, qui contient 60 000 images d'entraînement et 10 000 d'évaluation, réparties en 10 classes. Les images, en niveaux de gris, ont une résolution de $28 \times 28$ pixels, soit 784 dimensions d'entrée. L'objectif ici est de comprimer l'information via un encodage de plus faible dimension, puis de reconstruire les entrées originales à partir de cet encodage.

Les architectures testées partagent une forme symétrique et utilisent des poids transposés pour la phase de décodage. L'activation principale est la `LeakyReLU`($\alpha = 0.1$ ou $\alpha = 0.01$), avec une sortie sigmoïdale et une fonction de perte `BCELoss`. Voici un résumé des quatre modèles :

- **Modèle 1** : `784 → 256 → 64 → 256 → 784` ($\alpha = 0.1$)
- **Modèle 2** : `784 → 264 → 88 → 30 → 88 → 264 → 784` ($\alpha = 0.01$)
- **Modèle 3** : `784 → 424 → 128 → 32 → 128 → 424 → 784` ($\alpha = 0.01$)
- **Modèle 4** : `784 → 264 → 3 → 264 → 784` ($\alpha = 0.01$)

L'apprentissage a été effectué via un algorithme de descente de gradient avec mini-batchs (*batch size* = 128), sur 100 époques pour le modèle 1, 70 pour le modèle 3 et 50 époques pour les deux autres. Le taux d'apprentissage décroît exponentiellement selon l'époque (cf. code), permettant une stabilisation progressive de la descente sauf au premier modèle où le taux d'apprentsisage décroit par tranches.

Les courbes illustrent une convergence progressive pour tous les modèles, avec une perte de test stable, signalant une bonne généralisation. Le modèle 3 converge le plus rapidement et atteint les pertes les plus faibles.

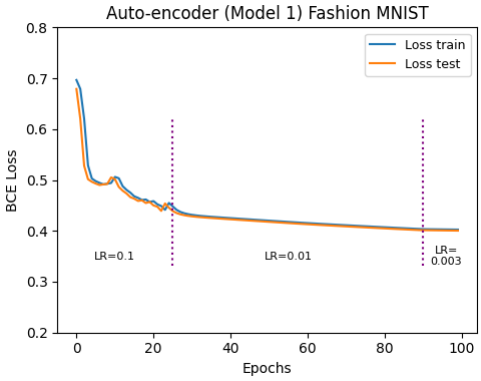



Figure 6: Courbe de loss du modèle 1

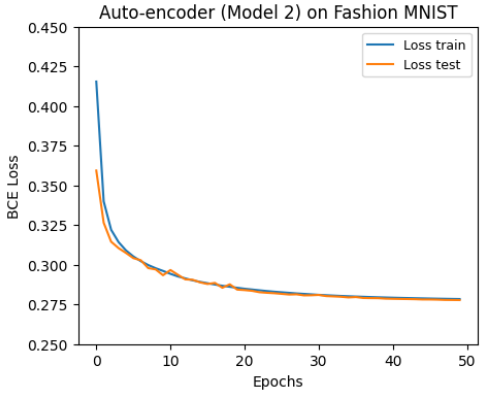



Figure 7: Courbe de loss du modèle 2

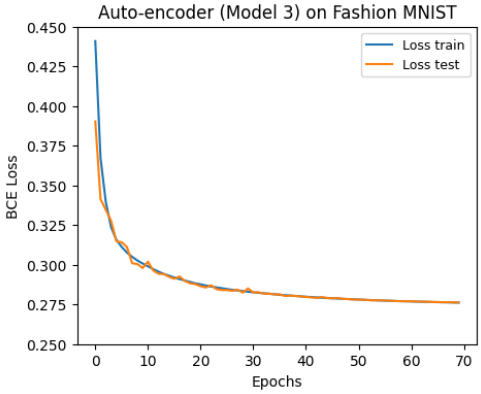



Figure 8: Courbe de loss du modèle 3

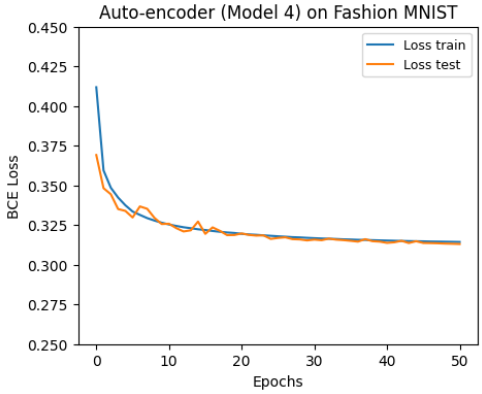



Figure 9: Courbe de loss du modèle 4

## Exemples de reconstruction

Les reconstructions obtenues sont visuellement satisfaisantes. On constate une perte d'information croissante lorsque la taille du goulot d'étranglement (bottleneck) diminue, notamment dans le modèle 4, qui projette les images dans un espace latent de dimension 3.

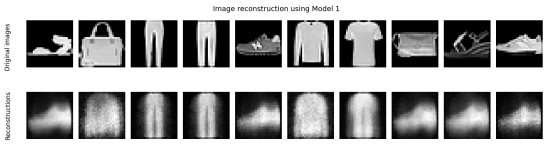



Figure 10: Reconstructions du Modèle 1 (bottleneck dim=64)

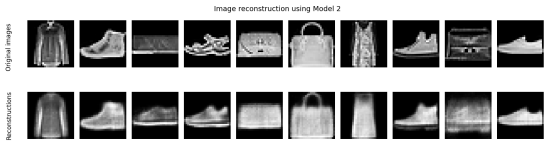



Figure 11: Reconstructions du Modèle 2 (bottleneck dim=30)

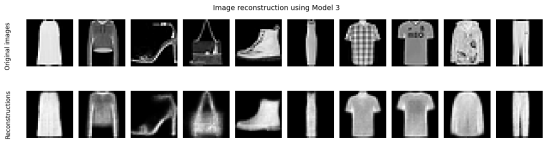



Figure 12: Reconstructions du Modèle 3 (bottleneck dim=32)

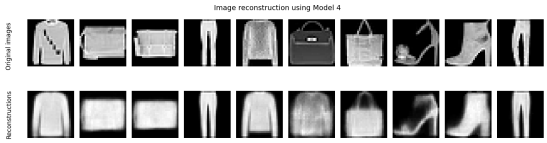



Figure 13: Reconstructions du Modèle 4 (bottleneck dim=3)

### Comparaison des performances

Table 3: Comparaison des performances des auto-encodeurs

| **Modèle** | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Profondeur | 4 | 6 | 6 | 4 |
| Dimension du bottleneck | 64 | 30 | 32 | 3 |
| Taux d'apprentissage initial | $10^{-1}$ | $10^{-1}$ | $10^{-1}$ | $10^{-1}$ |
| Stratégie de décroissance | heuristique | $0.95^t$ | $0.97^t$ | $0.97^t$ |
| Loss (train) | 0.40 | 0.278 | 0.27 | 0.314 |
| Loss (test) | 0.40 | 0.277 | 0.27 | 0.313 |
| Observations | Bonne reconstruction | Détails fins conservés | Reconstruction stable | Compression maximale |

On observe que les modèles disposant d'un encodage plus profond ou plus large (modèles 2 et 3) préservent mieux la structure des images. Le modèle 4, en compressant dans un espace de très faible dimension, perd des détails visuels mais peut être utilisé dans une optique de visualisation ou de clustering.

## Expérience 3 : Clustering des images multi-classes via leur représentation latente

Dans cette expérience, nous avons étudié la structure latente des images du jeu de données *Fashion-MNIST* en exploitant les représentations compressées issues des auto-encodeurs. L'objectif est de visualiser la distribution des embeddings dans des espaces à deux et trois dimensions, et d'évaluer dans quelle mesure ces représentations permettent une séparation des classes.

Nous avons principalement utilisé le **modèle 2**, en raison de sa structure plus légère et de performances comparables à celles du modèle 3. À titre d'illustration, les fichiers sauvegardés des deux modèles occupent respectivement environ 8 kB (modèle 2) et 13 kB (modèle 3), ce qui reflète une différence notable en nombre de paramètres.

**Nombre approximatif de paramètres.** Le modèle 2 est défini par les couches suivantes :

$$\texttt{Linear}(784, 176),\ \texttt{Tanh}(),\ \texttt{Linear}(176, 42),\ \texttt{Tanh}(),\ \texttt{Linear}(42, 10)$$

Le nombre de paramètres de la partie encodeur du modèle 2 est donné par :

$$(784 \times 264 + 264) + (264 \times 88 + 88) + (88 \times 30 + 30) = \mathbf{233\,230}$$

Le nombre de paramètres de la partie encodeur du modèle 3 est donné par :

$$(784 \times 424 + 424) + (424 \times 128 + 128) + (128 \times 32 + 32) = \mathbf{391\,368}$$

**Méthodologie.** Nous avons appliqué une passe avant (*forward pass*) sur l'ensemble d'apprentissage $X_{\text{train}}$, en ne considérant que la partie encodeur, afin d'extraire les représentations latentes (embeddings). Ces vecteurs ont été projetés dans un espace de dimension réduite à l'aide de l'algorithme `t-SNE` (*t-Distributed Stochastic Neighbor Embedding*) :

- **Projection en 2D** :
  `TSNE(n_components=2, perplexity=30, max_iter=3000)`

- **Projection en 3D** :
  `TSNE(n_components=3, perplexity=25, max_iter=3000, early_exaggeration=20)`

**Analyse des résultats.** Dans les deux cas, une séparation notable des classes est observée. Toutefois, certains regroupements ambigus apparaissent. En 2D, les classes *Shirt*, *Coat* et *Pullover* se confondent partiellement, ce qui s'explique par leur ressemblance morphologique. En 3D, une proximité similaire est observée entre les classes *T-Shirt*, *Shirt*, *Pullover* et, dans une moindre mesure, *Coat*.

Nous avons ensuite répété l'expérience avec le **modèle 4**, en projetant ses embeddings en 2D et en 3D selon les mêmes paramètres. Cette comparaison permet d'évaluer l'effet d'une architecture plus profonde (avec activations `ReLU`) sur la qualité de la séparation des représentations latentes. Entre 2D et 3D, la séparation est légèrement meilleure en 3D en utilisant le modèle 4 mais le contraire avec le modèle 1.

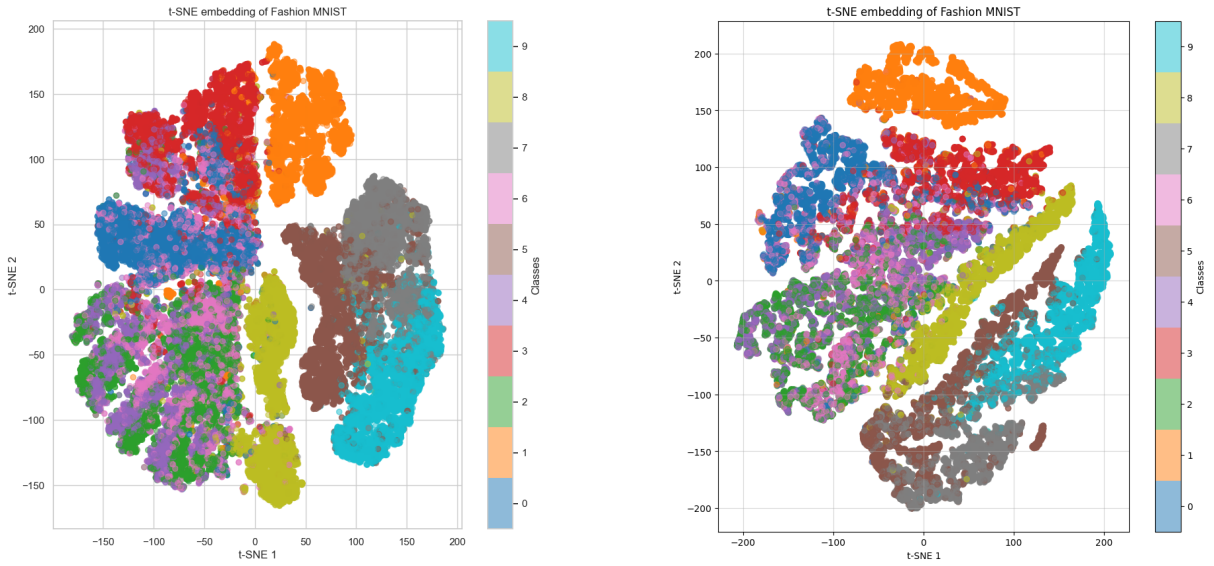
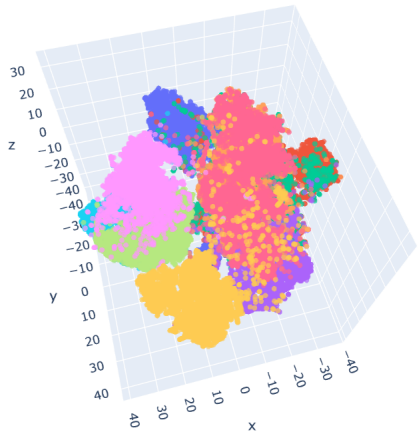
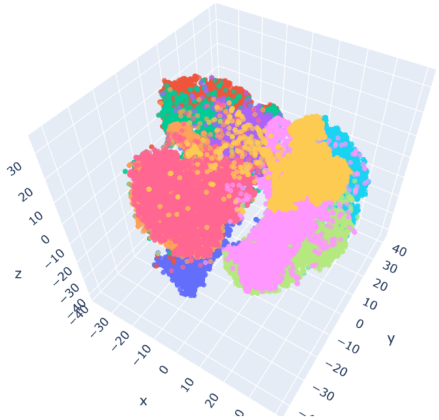


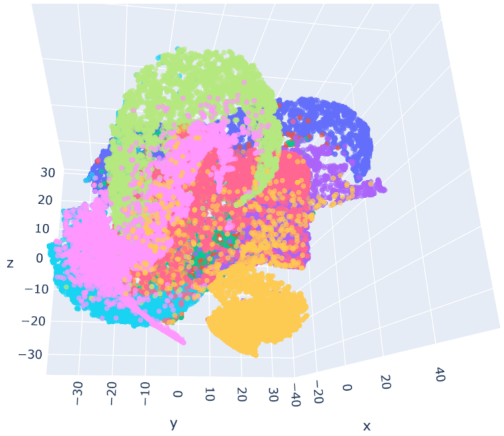

Figure 14: Visualisation 2D des représentations latentes du modèle 2 (gauche) et du modèle 4 (droite) des modèles de reconstruction d'images



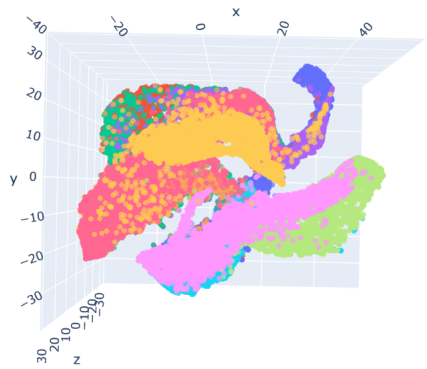
(a) Modèle 2 — vue 1



(b) Modèle 2 — vue 2



(c) Modèle 4 — vue 1



(d) Modèle 4 — vue 2

Figure 15: Visualisation 3D des représentations latentes extraites via t-SNE selon deux angles de vue pour les modèles 2 et 4

| **Modèle** | **Projection** | **KL divergence (après 3000 itérations)** |
|---|---|---|
| Modèle 2 | 2D | **1.201** |
| | 3D | 2.016 |
| Modèle 4 | 2D | 1.341 |
| | 3D | **1.210** |

Table 4: Comparaison de la divergence KL lors de l'application de t-SNE sur les représentations latentes des modèles 2 et 4

# Expérience 4 : Classification après prétraitement par auto-encodeur

Dans cette expérience, nous explorons le potentiel des auto-encodeurs en tant qu'outil de prétraitement pour des tâches de classification. L'intuition sous-jacente repose sur la capacité des auto-encodeurs à filtrer le bruit et à extraire des caractéristiques saillantes à partir des données brutes. Nous avons ainsi appliqué les auto-encodeurs **modèles 2, 3 et 4** de l'*Expérience 2* afin de reconstruire les images des jeux de données *Fashion-MNIST* et *Kuzushiji-MNIST*. Les classifieurs **modèles 3 et 4** (cf. *Expérience 1*) ont ensuite été réentraînés, non pas sur les images originales, mais sur leurs réprensations dans l'espace latent.

Cette approche permet de tester deux hypothèses :

- Les représentations dans l'espace latent peuvent préserver l'information discriminante tout en supprimant certains détails inutiles ou bruités.
- Un auto-encodeur bien entraîné peut agir comme un filtre informationnel, facilitant ainsi la tâche de classification en aval.

## Protocoles expérimentaux

Pour chaque auto-encodeur (modèles 2, 3 et 4), nous avons généré les embeddings en espace latent de différentes dimensions chacun, des images d'entrée. Ces sorties ont ensuite été utilisées comme données d'apprentissage et de test pour les classifieurs suivants :

- **Modèle 3** : `Linear(784,176) → Tanh → Linear(176,42) → ReLU → Linear(42,10)`
- **Modèle 4** : `Linear(784,176) → Tanh → Linear(176,42) → ReLU → Linear(42,10) → ReLU`

Les hyperparamètres d'apprentissage (nombre d'itérations, taux d'apprentissage) ont été maintenus constants, identiques à ceux de l'expérience 1, afin de garantir une comparaison équitable.

## Résultats

Table 5: Performances de classification avant et après le prétraitement par auto-encodeur

| **Classifieur Auto-Encodeur** | **Fashion-MNIST (3)** | | | | **Kuzushiji-MNIST (4)** | | | |
|---|---|---|---|---|---|---|---|---|
| | **-** | **AE 2** | **AE 3** | **AE 4** | **-** | **AE 2** | **AE 3** | **AE 4** |
| Train Loss | 0,55 | 0,95 | **0,89** | 1,77 | 0,55 | **0,97** | 0,99 | 1,89 |
| Test Loss | 0,53 | 0,95 | **0,89** | 1,78 | 0,64 | **0,99** | 1,05 | 1,90 |
| Accuracy | 0,81 | 0,69 | **0,70** | 0,34 | 0,80 | **0,67** | 0,64 | 0,33 |
| F1-score | 0,81 | 0,68 | **0,70** | 0,31 | 0,80 | **0,67** | 0,57 | 0,30 |
| Precision | 0,82 | 0,70 | **0,71** | 0,34 | 0,80 | **0,69** | 0,54 | 0,31 |
| Rappel | 0,80 | 0,69 | **0,70** | 0,34 | 0,80 | **0,69** | 0,64 | 0,33 |

Les résultats du tableau **??** indiquent que l'utilisation des images reconstruites par les auto-encodeurs comme données d'entrée pour la classification **entraîne généralement une dégradation des performances** sur les deux jeux de données considérés.

**Cas de *Fashion-MNIST* :**

- La **meilleure performance** est atteinte sans reconstruction (baseline), avec une accuracy de **0,81**.
- L'utilisation des auto-encodeurs AE2 et AE3 conduit à des performances inférieures. Toutefois, AE3 semble légèrement mieux préserver les métriques (F1-score et précision à environ **0,70–0,71**).
- AE4 dégrade sévèrement la qualité : l'accuracy chute à **0,34**, ce qui équivaut pratiquement à un comportement aléatoire pour une classification en 10 classes.

**Cas de *Kuzushiji-MNIST* :**

- Une tendance similaire est observée : la baseline fournit une accuracy de **0,80**.
- Les auto-encodeurs AE2 et AE3 aboutissent à des résultats amoindris (respectivement **0,67** et **0,64**).
- Le modèle AE4 échoue à capturer l'information discriminante : l'accuracy tombe à **0,33**, illustrant un échec quasi complet de la reconstruction utile pour la classification.

Ces observations suggèrent que les auto-encodeurs utilisés ne parviennent pas à reconstruire les images de manière suffisamment fidèle pour préserver les structures discriminantes nécessaires à la classification. Plusieurs hypothèses peuvent être avancées :

- Le **goulot d'étranglement** (*bottleneck*) du réseau, correspondant au faible nombre de dimensions latentes, pourrait être trop contraint pour encoder les détails pertinents.
- L'entraînement des auto-encodeurs pourrait ne pas avoir été suffisamment poussé ou correctement optimisé (choix du taux d'apprentissage, absence de régularisation, etc.).
- Les auto-encodeurs ont été entraînés pour minimiser une fonction de reconstruction *pixel-wise*, qui n'incorpore aucune contrainte sur la **sémantique** ou sur la **structure discriminante** des classes.

# Conclusion et Perspectives

Ce projet a démontré la faisabilité d'une bibliothèque de réseaux de neurones développée «from scratch», en validant l'apprentissage différentiel via des MLP simples et des auto-encodeurs, tout en révélant leurs limites sur des tâches visuelles. L'expérience de prétraitement par auto-encodeur (Expérience 4) a montré que la reconstruction pixel-wise n'assure pas le maintien de l'information discriminante, notamment lorsque le goulot d'étranglement est trop restreint.

Pour pallier ces insuffisances, il conviendrait de :

- Réviser le **bottleneck** (plus de dimensions ou régularisations structurées) et enrichir la **fonction de reconstruction** (perceptual loss, loss conjointe reconstruction/classification).
- Adopter des architectures spécialisées pour l'image: **CNN** et **auto-encodeurs convolutifs** pour capter les motifs spatiaux.
- Explorer des approches avancées: apprentissage **auto-supervisé** (SimCLR, BYOL), pipelines **end-to-end**, techniques de **régularisation** et d'**AutoML** pour optimiser hyperparamètres et architectures.

Ces pistes ouvrent un champ d'améliorations prometteur pour rapprocher notre solution «DIY» des standards actuels en deep learning, alliant clarté algorithmique et robustesse expérimentale.