Master Informatique

Réseaux de neuronnes : Perceptron, MLP et Applications

Hocine LOUBAR et Bayane Aijjou

Encadrant : Nicholas Baskiotis

Semestre de printemps 2025

Contents

0.1 Introduction 2

1 Notions théoriques pour l'implémen 3

1.1 Perceptron et Module **Linear** 3

1.2 MLP (Multi-Layer Perceptron) 3

1.3 Fonctions d'activation 4

1.4 Backpropagation : calcul du gradient 5

1.5 Optimizer 6

1.6 Optimiseur et apprentissage par mini-batchs 6

2 Expérimentations et tests 7

2.1 Régression linéaire bruitée sur données synthétiques 7

2.2 Classification de données non linéaires : Perceptron vs MLP 10

2.2.1 Données Moons 10

2.2.2 Blobs : clusters - données regroupées par grappes 13

2.2.3 Classification sur un motif d'échiquier 16

2.3 Auto-encoder : apprentissage non supervisé sur USPS 23

2.3.1 Auto-encoder 1 23

2.3.2 Auto-encodeur 2 24

2.3.3 Auto-encodeur 3 25

2.3.4 Clustering des représentations latentes 27

2.3.5 Classification appliquée aux visages (YaleFaces) 30

0.1 Introduction

L'évolution des réseaux de neurones en intelligence artificielle commence historiquement par le perceptron. Le perceptron est un modèle trés simple apte a apprendre une frontière de décision linéaire entre deux classes entre **deux** classes. Concrètement, il fonctionne comme un neurone formel : il effectue une combinaison linéaire de ses entrées et applique une fonction de prédiction (ex. **Seuil**) pour prédire une classe.
Cependant le perceptron ne résout des problèmes de classification que si les données sont linéairement séparables, ce qui lui attribue une limite fondamentale. Dès lors que les données sont complexes (ex. Le problème du **XOR**) le perceptron échoue. Pour dépasser cette limite, le *Multi-Layer Perceptron* (MLP) a été introduit. Il consiste en une superposition de plusieurs couches de neurones, chacune composée de fonctions linéaires suivies de non-linéarités. Cette structure permet de modéliser des fonctions de décision complexes, non-linéaires, et a joué un rôle central dans le développement du *deep learning* moderne.
Dans ce projet nous étudions et implémentons pas à pas ces modèles fondamentaux — le perceptron multi-couches (**MLP**) et les réseaux de neuronnes convolutionnels (**CNN**) — dans une architecture logicielle contruite *from scratch*. Ces modèles seront entrainés sur des données réelles, notamment le dataset USPS (ou bien CIFAR10), a but d'observer leur capacité de généralisation et de classification.

1 Notions théoriques pour l'implémen

1.1 Perceptron et Module Linear

Un perceptron est un "neuronne" unique qui effectue une combinaison linéaire de ses entrées, suivi d'une fonction de décision, mthématiquemle processus est décrit par :

$$z = w^\top x + b \tag{1}$$

étant la fonction affine, et

$$\hat{y} = \text{step}(z) \tag{2}$$

La fonction de décision.
Cependant, pour une approche plus générale, on considère simplement :

$$\hat{y} = z = w^\top x + b$$

Ce qui permet a notre modèle de faire de la régression linéaire, qui est basée sur ce même principe. Ce module **Linear** est la première étape vers la construction de réseaux plus complexes. Le module *Linear* applique la transformation affine a l'entrée, etant paramétré par une matrice de poids $W \in \mathbb{R}^{d_{in} \times d_{out}}$ dont les dimensions dépendent de la taille des exemples (nombre de features) et du nombre de neuronnes dans la couche cachée (ou de sortie) et d'un vecteur de biais $b \in \mathbb{R}^{d_{out}}$ (un biais pour chaque neuronne).

```
class Linear(Module):
    def forward(self, X):
        return X @ self._parameters
```

La classe **Linear** hérite d'une classe abstraite **Module** dont l'architecture des MLPs dépends et est définie, c'est une représentation générique d'un module, qui est l'élément d'une séquence qui représentera le réseau.
Un module peut etre linéaire, non-linéaire tel une fonction d'activation, un réseau de neurones (lui même composé de modules) ou bien un transformation des sorties (telle la Softmax pour la classification multiclasse).
Chaque module contient les surcharges des méthodes de la classe Module, qui sont spécifiée dans celle-ci :

- Méthode forward pour calculer les sorties du module a partir des entées.
- méthode backward pour calculer les dérivées des sorties du module par rapport a ses entrés ainsi que ses paramètres nécessaires pour la mise à jour pendant l'apprentissage.
- autres méthodes tel que la **zerograd**, **updatedelta**..

1.2 MLP (Multi-Layer Perceptron)

Avec les limitations du perceptron simple a des problème linéairement séparables, il a été necessaire d'introduire des architectures plus robustes et puissantes : le *Multi-Layer Perceptron*.
Le MLP est une composition de plusieurs couches linéaires séparées par des fonctions d'activation non-linéaires. Cette structure permet d'approximer n'importe quelle fonction continue sur un domaine compacte (théorème d'universalité).

Le MLP est décrit comme une suite de modules, alternant entre linéaire et non-linéaire.
La propagation avant (forward) dans un MLP se décrit ainsi :

$$z^{[0]} = x \tag{3}$$

$$a^{[\ell]} = W^{[\ell]} z^{[\ell-1]} + b^{[\ell]}, \quad z^{[\ell]} = g(a^{[\ell]}) \tag{4}$$

où :

- $W^{[\ell]}$ est la matrice de poids de la couche ℓ,
- g est une fonction d'activation non-linéaire (par exemple `Tanh` ou `Sigmoid`),
- $z^{[\ell]}$ est l'activation de la couche ℓ.

Dans notre implémentation, chaque couche est représentée par un module (héritant de `Module`), et les couches sont combinées à l'aide du module `Sequential`, qui applique chaque transformation l'une après l'autre. Voici un exemple de définition de réseau MLP utilisé dans nos expérimentations :

```
net = Sequential(
    Linear(256, 64),
    Tanh(),
    Linear(64, 10),
    LogSoftmax()
)
```

Cette architecture permettra l'apprentissage d'une fonction de classification non-linéaire a partie de données images en entrée (ayant la forme d'un vecteur de taille 16x16 pixels). L'activation `Tanh` introduit de la non-linéarité afin de séparer des classes non séparables linéairement. Le résultat de l'entrainement du réseau sur le dataset USPS (chiffres manuscrits) sera vu dans la section `Expérimentations`.

1.3 Fonctions d'activation

On introduit de la non-linéarité dans les réseaux profonds a l'aide de fonctions d'activations, sans lesquels, le réseau multi-couches sera mathématiquement équivalent a un perceptron (avec plus de parametres a apprendre), car la conposition de fonction linéaires reste linéaire. En introduisant des fonctions d'activations non-linéaires entres les couches permet d'augmenter l'expressivité du modèle et permettra au réseau d'apprendre des représentations complexes.
Les fonctions d'activations implémentés dans ce projet sont :

- **Tanh** : fonction hyperbolique définie par $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Sa dérivée est :

$$\tanh'(z) = 1 - \tanh^2(z)$$

- **Sigmoid** : souvent utilisée en sortie pour des probabilités, définie par :

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

- **LogSoftmax** : utilisée en sortie pour la classification multi-classes. Elle transforme les scores en log-probabilités stables numériquement :

$$\text{logSoftmax}(z_i) = z_i - \log\left(\sum_j e^{z_j}\right)$$

Ces fonctions sont implémentées comme des modules héritant de `Module`, toutes avec une méthode `forward()` pour la passe avant, et `backward_delta()` pour calculer le gradient lors de la rétropropagation.

1.4 Backpropagation : calcul du gradient

L'appretissage d'un réseau de neuronne repose sur le calcul du gradient de la fonction de perte par rapport aux paramètres du modèle. Ce calcul se fait grâce à l'algorithme de rétropropagation qui utilise la règle de la chaîne a travers les couches du modèle.
Considérons une couche linéaire suivie d'une fonction d'activation :

$$z^{[\ell]} = W^{[\ell]}a^{[\ell-1]} + b^{[\ell]}, \quad a^{[\ell]} = g(z^{[\ell]})$$

On note $\mathcal{L}$ la fonction de perte, et on cherche à calculer :

$$\frac{\partial \mathcal{L}}{\partial W^{[\ell]}}, \quad \frac{\partial \mathcal{L}}{\partial b^{[\ell]}}, \quad \frac{\partial \mathcal{L}}{\partial z^{[\ell-1]}}$$

La rétropropagation commence par la dernière couche et calcule le **delta local** :

$$\delta^{[\ell]} = \frac{\partial \mathcal{L}}{\partial a^{[\ell]}} = \left(\frac{\partial \mathcal{L}}{\partial z^{[\ell]}}\right) \odot g'(a^{[\ell]})$$

où $\odot$ désigne le produit terme à terme (Hadamard).

Ensuite, on a :

$$\frac{\partial \mathcal{L}}{\partial W^{[\ell]}} = \delta^{[\ell]} \cdot (z^{[\ell-1]})^\top, \quad \frac{\partial \mathcal{L}}{\partial b^{[\ell]}} = \delta^{[\ell]}$$

et

$$\delta^{[\ell-1]} = (W^{[\ell]})^\top \cdot \delta^{[\ell]} \odot g'(a^{[\ell-1]})$$

Dans notre implémentation, chaque module possède une méthode `backward_delta()` qui renvoie le delta à propager à la couche précédente, et une méthode `backward_update_gradient()` qui calcule les gradients des poids et biais.

Par exemple, pour une couche `Linear`, on code :

```
def backward_update_gradient(self, input, delta):
    self._gradient += input.T @ delta
```

et

```
def backward_delta(self, input, delta):
    return delta @ self._parameters[:-1].T
```

Ces opérations sont valides batch par batch, et s'inscrivent dans un pipeline général via la classe `Sequential`, qui gère l'enchaînement des modules.

1.5 Optimizer

1.6 Optimiseur et apprentissage par mini-batchs

Une fois les gradients calculés par rétropropagation, ils sont utilisés pour mettre à jour les paramètres du réseau en suivant une règle d'optimisation. Dans notre projet, nous avons implémenté un optimiseur basé sur la descente de gradient stochastique (SGD).

Concrètement, l'entraînement se fait sur plusieurs époques (*epochs*), où chaque époque consiste en un passage complet sur toutes les données. Afin de rendre l'entraînement plus efficace et plus stable, les données sont découpées en mini-lots (*mini-batchs*). Chaque mini-batch contient un petit sous-ensemble des données d'entraînement (par exemple 32 exemples), sur lequel l'optimiseur effectue une mise à jour des poids.

L'entraînement suit le cycle suivant :

1. Les données sont mélangées aléatoirement à chaque époque.
2. Pour chaque mini-batch :
   - Le réseau effectue une passe avant pour obtenir les prédictions (`forward`).
   - La fonction de perte (`Loss`) calcule l'erreur entre la prédiction et la vérité terrain.
   - Le gradient de cette perte est calculé (`backward`).
   - Chaque module calcule ses gradients internes via `backward_update_gradient`.
   - Les poids sont mis à jour par l'optimiseur selon la formule :

     $$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}$$

     où η est le taux d'apprentissage (learning rate), ici noté `eps`.
3. La perte moyenne sur l'époque est enregistrée pour observer la convergence.

Le cœur de ce processus est contenu dans notre classe `Optim`, qui encapsule un réseau (`Sequential`) et une fonction de perte (`Loss`). Elle applique l'entraînement batch par batch via la méthode `step` :

```
def step(self, batch_x, batch_y):
    yhat = self.net.forward(batch_x)
    loss_value = self.loss.forward(batch_y, yhat)

    delta = self.loss.backward(batch_y, yhat)
    self.net.zero_grad()
    self.net.backward_update_gradient(batch_x, delta)
    self.net.update_parameters(self.eps)

    return float(np.mean(loss_value))
```

L'ensemble du processus d'apprentissage est géré par la fonction `SGD()`, qui appelle `step()` pour chaque mini-batch. Elle prend en entrée les données, la taille des batchs, le nombre d'époques, et retourne l'historique des pertes.

Notre implémentation est générique et permet d'utiliser différentes fonctions de perte :

- `MSELoss` : perte quadratique moyenne, adaptée à la régression.

- `NLLLoss` : perte log-likelihood négative pour la classification multi-classes avec sorties en log-softmax.
- `BCELoss` : perte binaire pour des sorties comprises entre 0 et 1 (ex. classification binaire).

Ce système modulaire permet de facilement expérimenter avec différents types de données et de tâches d'apprentissage supervisé.

2 Expérimentations et tests

2.1 Régression linéaire bruitée sur données synthétiques

Afin de tester notre implémentation `from scratch`, nous avons d'abord expérimenté avec une tâche simple de régression linéaire. L'objectif est de vérifier que notre modèle peut apprendre une relation linéaire entre une variable x et une cible y, malgré la présence de bruit.

Données simulées. Nous générons un jeu de données synthétique de la forme :

$$y = 3x + 2 + \epsilon$$

où $\epsilon \sim \mathcal{N}(0, 0{,}8)$ est un bruit gaussien ajouté pour simuler des données imparfaites. 200 points sont générés sur l'intervalle $[-1, 1]$, puis séparés en un ensemble d'entraînement (80%) et un ensemble de test (20%).

Architecture. Le réseau utilisé ici est un perceptron simple constitué d'une seule couche linéaire :

```
net = Sequential(
    Linear(1, 1)
)
```

Perte et entraînement. Nous utilisons la fonction de perte quadratique moyenne (`MSELoss`), et entraînons le modèle à l'aide de l'algorithme SGD (descente de gradient stochastique) pendant 100 époques, avec un taux d'apprentissage $\varepsilon = 0{,}1$ et des mini-batchs de taille 10.

Résultats sur l'ensemble d'entraînement. Après l'apprentissage, nous observons que le modèle parvient à apprendre une droite de régression proche de la vraie tendance sous-jacente, malgré la présence de bruit dans les données.

Figure 1 : Nuage de points sur les données d'entraînement avec la prédiction du modèle (en rouge).

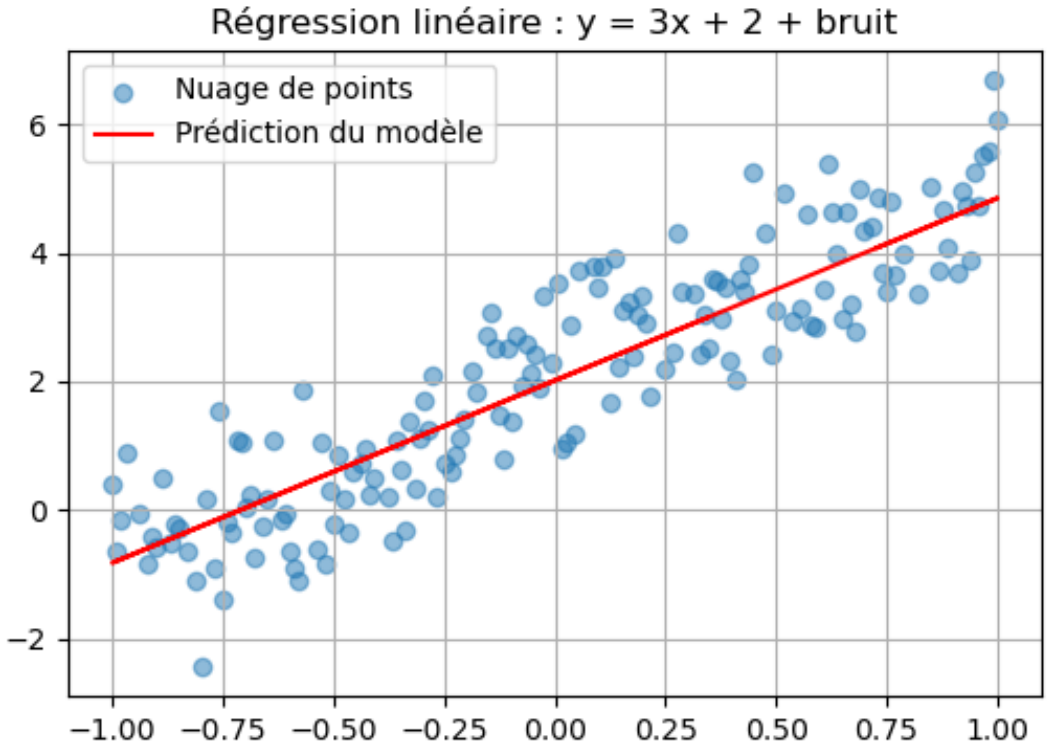
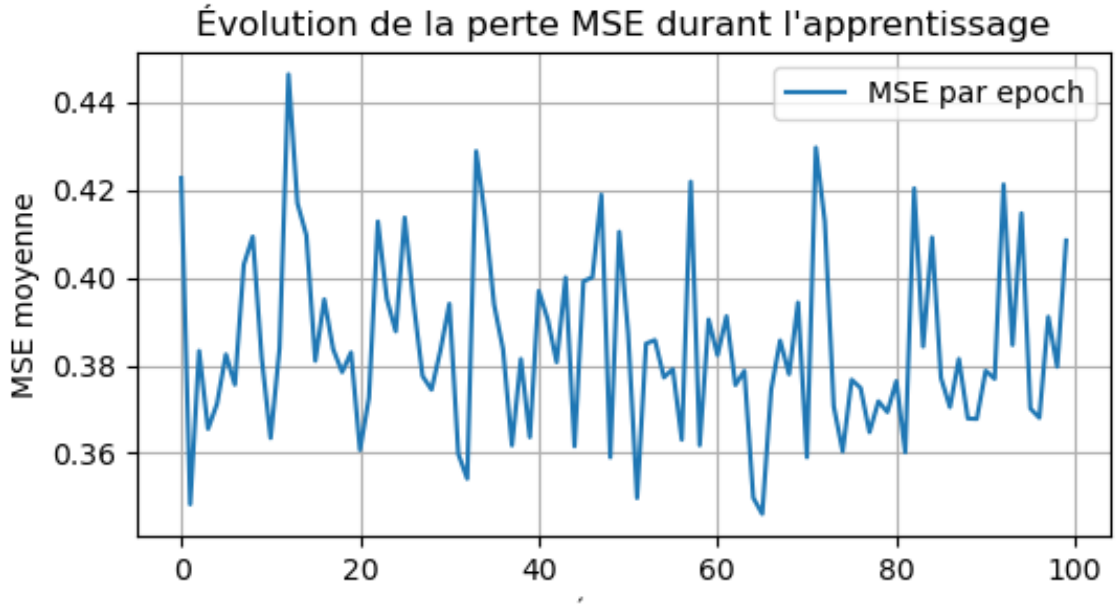


Figure 2 : Évolution de la perte MSE moyenne au cours de l'entraînement.

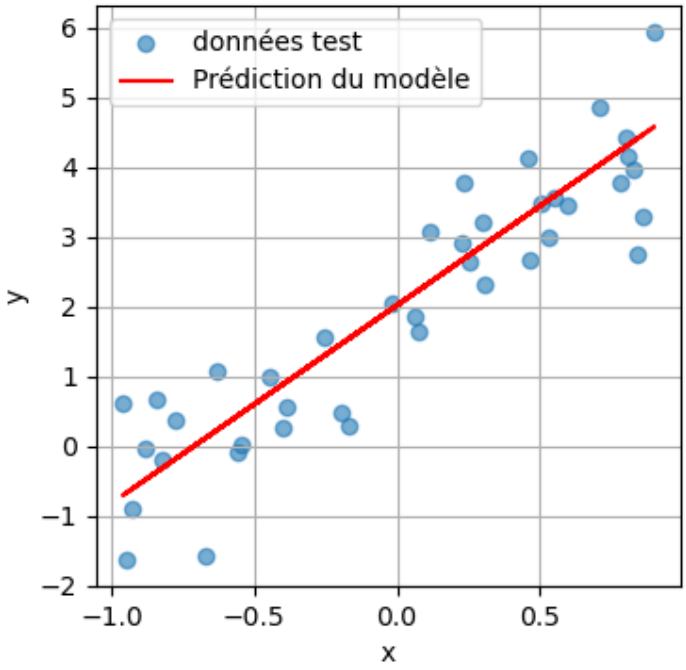


Résultats sur l'ensemble de test. Le modèle est ensuite évalué sur les données de test. Le MSE obtenu est :

$$\text{MSE}_{\text{test}} \approx 0{,}62$$

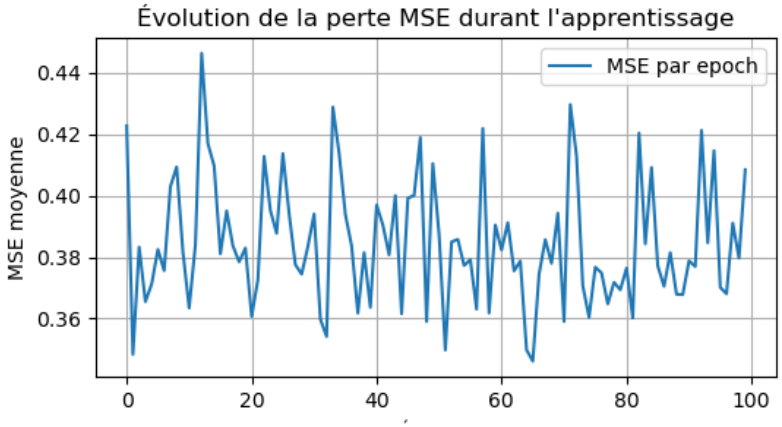
ce qui indique une bonne capacité de généralisation.

Figure 3 : Prédiction sur les données de test.



Comparaison à la vraie droite. Enfin, pour vérifier la fidélité du modèle, nous comparons la prédiction du réseau sur tous les points x aux valeurs exactes sans bruit données par $y = 3x + 2$. Cela permet d'estimer visuellement à quel point le modèle a appris la vraie fonction.

Figure 4 : Courbes de la vraie droite (sans bruit) et de la prédiction du modèle.



Conclusion. Ces tests valident le bon fonctionnement de notre module `Linear`, de la fonction de perte `MSELoss`, de la descente de gradient, et du pipeline d'entraînement `Sequential + Optim`. Le

réseau a bien appris à approximer une fonction linéaire bruitée, tant sur les données d'entraînement que sur les données de test.

2.2 Classification de données non linéaires : Perceptron vs MLP

2.2.1 Données Moons

Dans cette seconde expérimentation, nous abordons un problème de classification supervisée sur des données non séparables linéairement. L'objectif est d'observer la capacité de différents réseaux (perceptron simple puis MLPs) à apprendre des frontières de décision adaptées.

Problème. Nous générons un jeu de données synthétique en 2D composé de deux classes difficilement séparables par une frontière linéaire. Chaque échantillon appartient à une des deux classes, encodée sous forme one-hot. Les données sont divisées en ensembles d'entraînement et de test.

Évaluation. La performance est mesurée à l'aide de la précision (accuracy) sur les données de test. Nous visualisons également les frontières de décision des modèles, c'est-à-dire les zones de l'espace où le modèle prédit une classe ou l'autre.

Perceptron simple. Nous commençons par entraîner un réseau ne comportant qu'une couche linéaire suivie d'une activation `LogSoftmax`. Ce modèle ne peut apprendre qu'une frontière linéaire. Il atteint :

$$\text{Accuracy train} = 81.25\%, \quad \text{Accuracy test} = 82.50\%$$

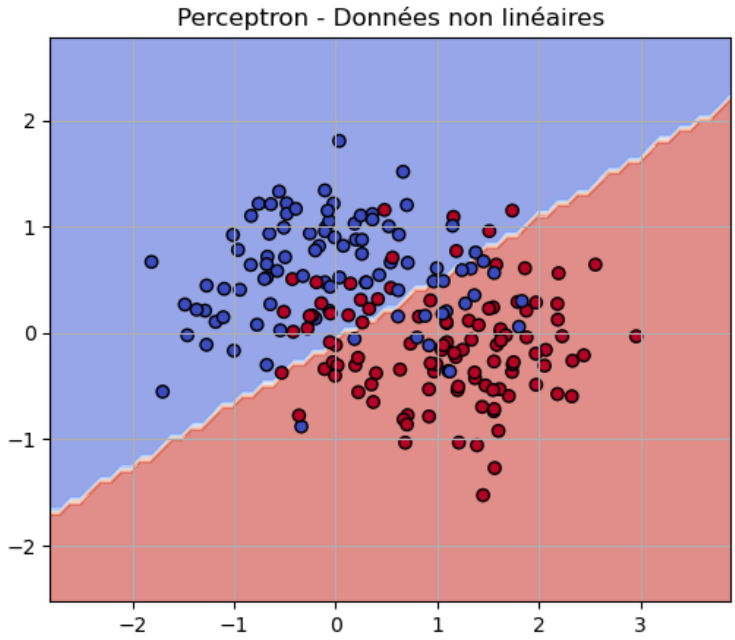


Figure 1 : Frontière de décision apprise par un perceptron simple.

MLP avec une couche cachée. Nous entraînons ensuite un MLP avec une seule couche cachée de 4 neurones et une activation `Tanh`. Cela permet au réseau d'apprendre des frontières de décision non linéaires, plus adaptées à la distribution des données. La précision sur le test augmente à 89.50%.

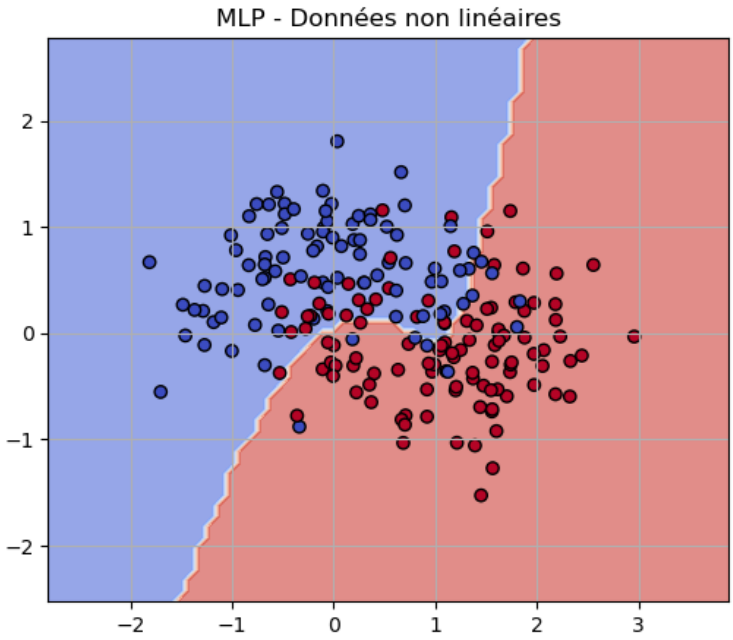


Figure 2 : Frontière de décision d'un MLP avec une couche cachée de 4 neurones.

Effet de l'architecture. Nous testons différentes tailles et profondeurs pour l'architecture du MLP. L'ajout de neurones dans la couche cachée affine légèrement la frontière, mais n'apporte pas de gain significatif sur la précision.

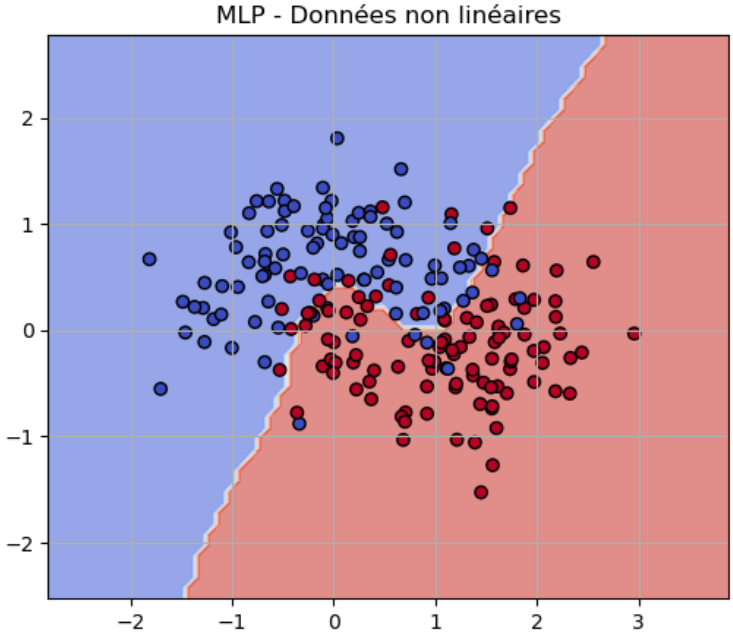


Figure 3 : MLP avec une couche de 6 neurones cachés.

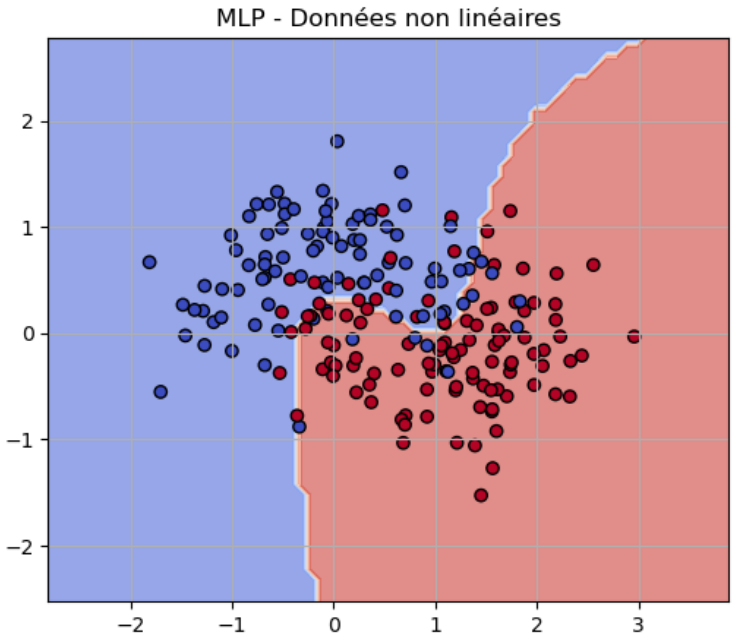


Figure 4 : MLP avec 8 neurones dans la couche cachée.

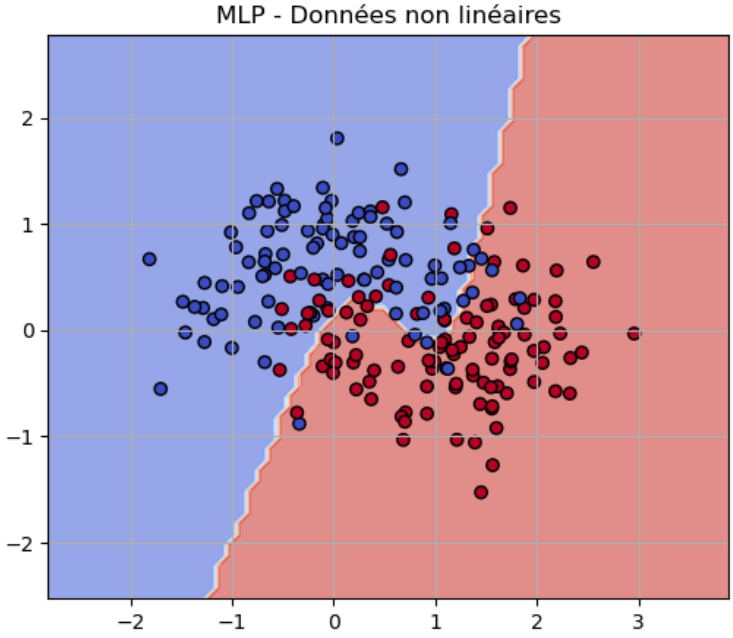


Figure 5 : MLP avec 10 neurones cachés.

Cas complexes. Enfin, nous avons testé nos modèles MLP sur des jeux de données plus complexes (multi-classes et distributions complexes). Le MLP parvient à approximer une séparation des classes mais montre ses limites lorsque la complexité de la distribution augmente (par exemple, clusters proches ou enchevêtrés).

2.2.2 Blobs : clusters - données regroupées par grappes

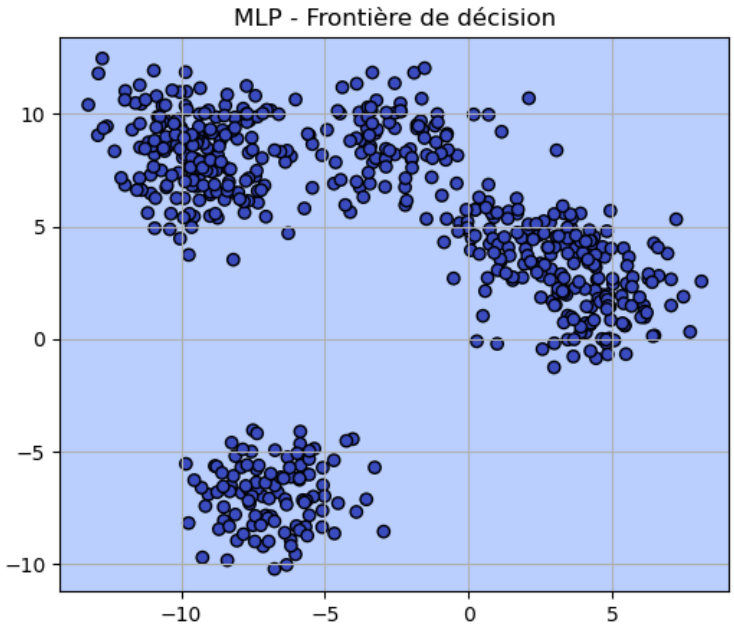


Figure 6 : Frontière apprise sur un jeu de données multi-classe relativement bien séparé.

Pas de sur apprentissage, ni de sous apprentissage, le résultat est bon , juste avec un modèle assez simple.

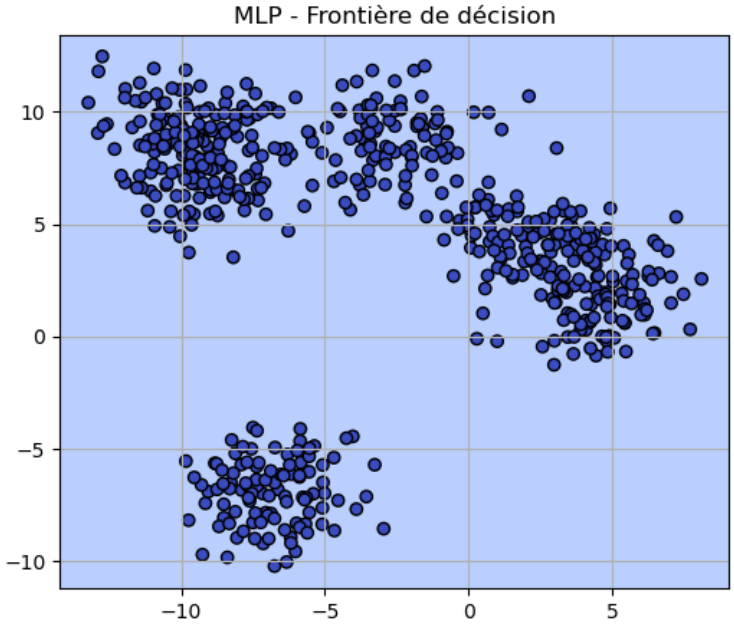


Figure 7 : Même jeu avec une autre initialisation ou architecture (frontière légèrement différente).

- Les données blobs sont bien séparables, meme si ils ne sont pas linéaires, 32 neuronnes suffit largement.
- Pas trop de bruit dans les données, alors pas besoin d'un modèle puissant.

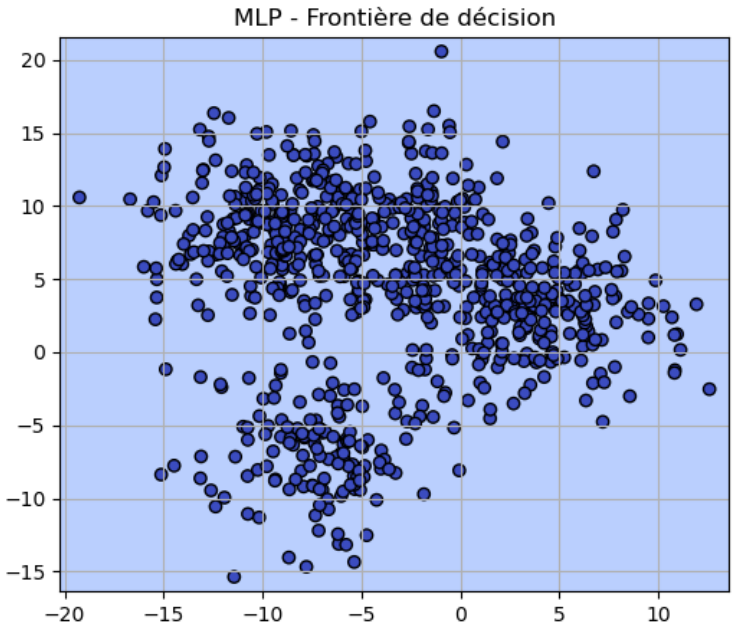


Figure 8 : Jeu de données avec clusters plus dispersés – complexité accrue pour le MLP.

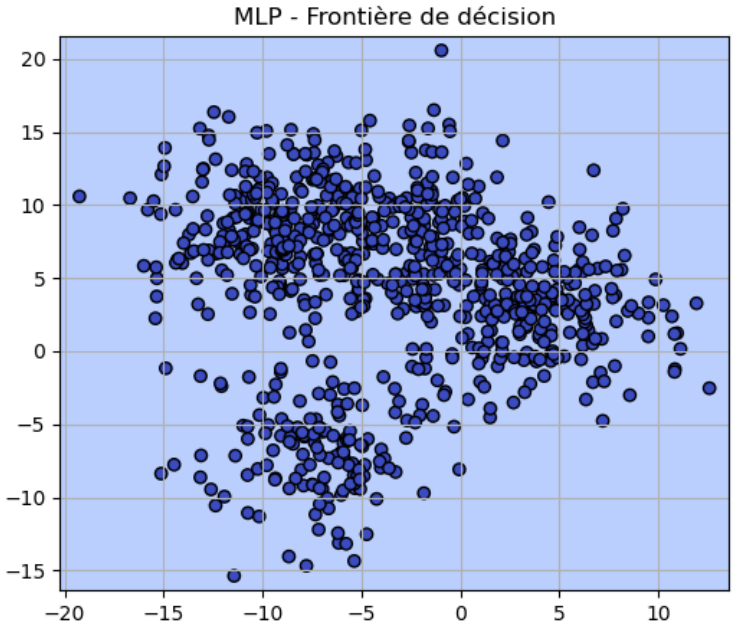


Figure 9 : Variante du dataset précédent : les classes se croisent fortement.

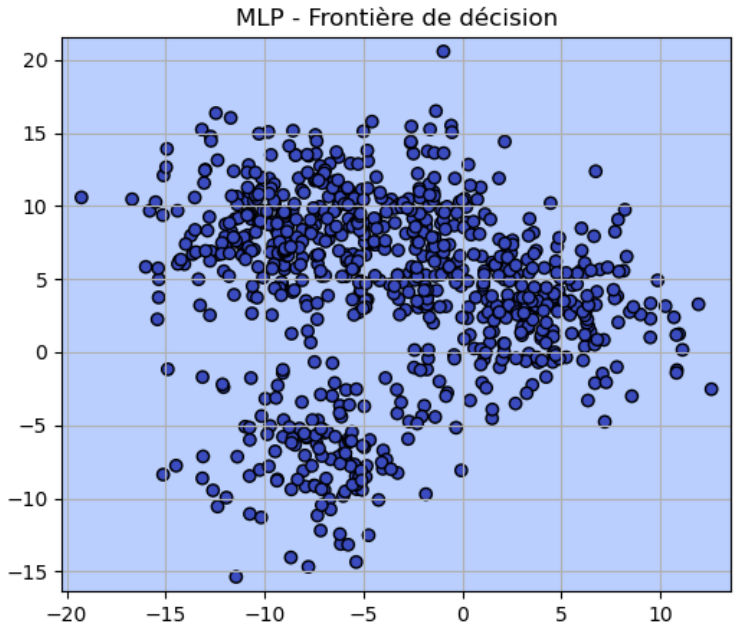


Figure 10 : Même dataset avec un MLP ne parvenant pas à séparer correctement les classes.

Cas supplémentaires : classification de blobs avec bruit. Enfin, nous avons testé une série d'architectures MLP sur un jeu de données synthétique avec plusieurs amas (blobs), certains bruités, afin d'étudier l'impact des choix d'architecture, d'initialisation et d'hyperparamètres sur la précision finale.

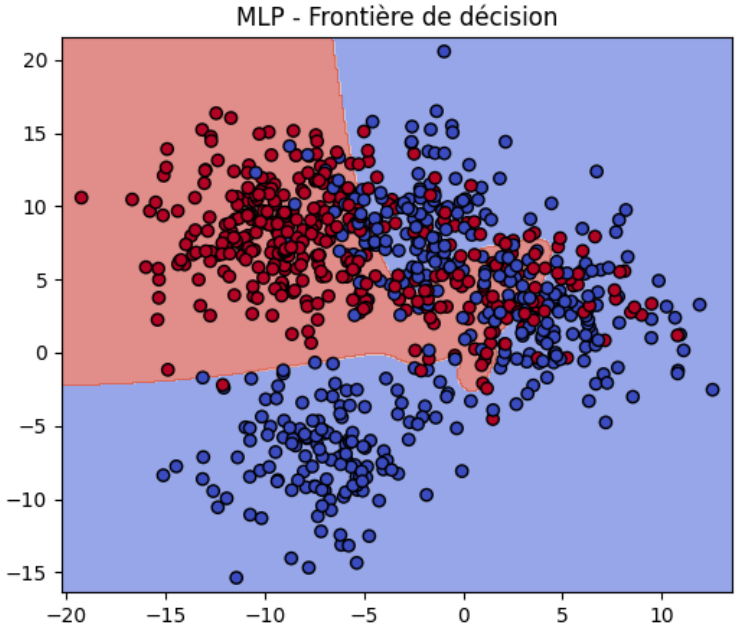


Figure 11 : MLP entraîné sur un jeu de blobs avec bruit : la frontière s'adapte aux structures non linéaires.

Les résultats sont synthétisés dans le tableau suivant :

Architecture	Init	Activation	LR	Batch	Epochs	Acc train	Acc test
2-32-1	1	Tanh + Sigmoïde	0.09	32	300	94.71%	93.50%
2-128-1	1	Tanh + Sigmoïde	0.09	16	300	94.54%	93.67%
2-128-1 + bruit	1	Tanh + Sigmoïde	0.09	16	300	79.72%	80.00%
2-64-128-64-1 + bruit	1	Tanh + Sigmoïde	0.01	16	300	79.75%	79.88%
2-64-128-64-1 + bruit	1	Tanh + Sigmoïde	0.10	16	300	80.00%	80.38%

Table 1: Performances de différentes architectures MLP sur un jeu de données non linéaire bruité.

2.2.3 Classification sur un motif d'échiquier

Celui-ci illustre une fonction non linéaire dont les classes sont organisées en damier. Le paramètre k régule la fréquence des cases : plus k est grand, plus les motifs sont petits et donc plus la tâche de classification devient difficile.

Ce problème est un excellent test pour évaluer la capacité d'un MLP à approximer des fonctions complexes et hautement non linéaires.

Résultat avec un perceptron simple. Sans surprise, le perceptron simple est incapable d'apprendre les motifs croisés du damier. Sa frontière de décision est purement linéaire.

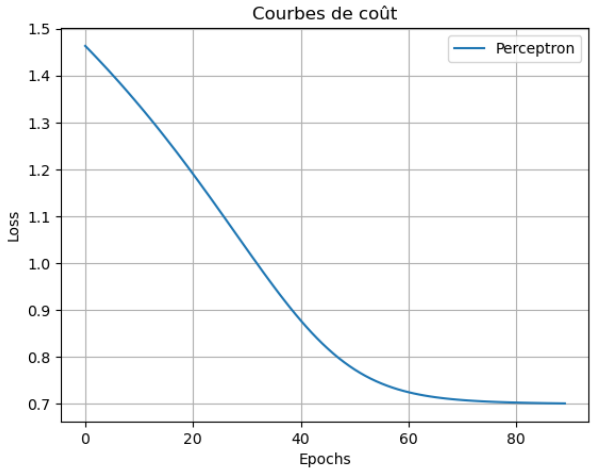


Figure 1: *

Figure 28 : Courbe de coût.

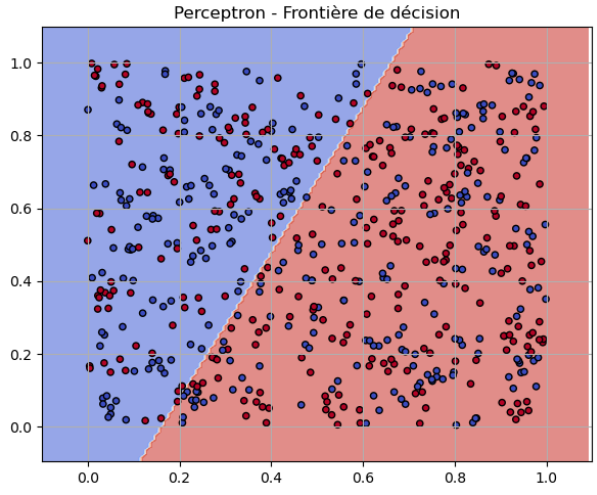


Figure 2: *

Figure 29 : Perceptron sur motif échiquier : échec total de séparation.

MLP de petite taille. Nous entraînons un MLP avec une petite architecture (par exemple : 2-32-1) pour observer s'il peut capturer partiellement le motif.

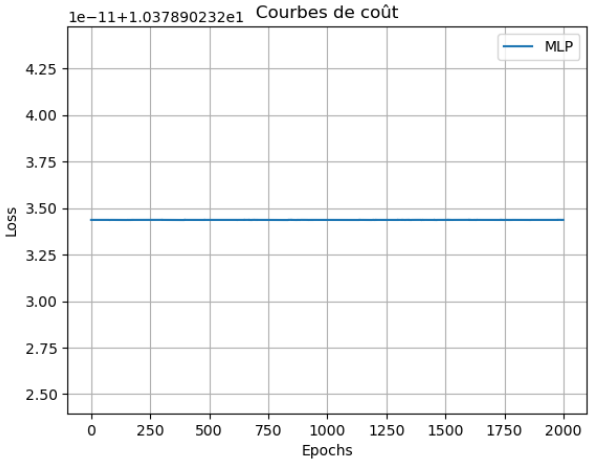


Figure 3: *

Figure 30 : Courbe de coût.

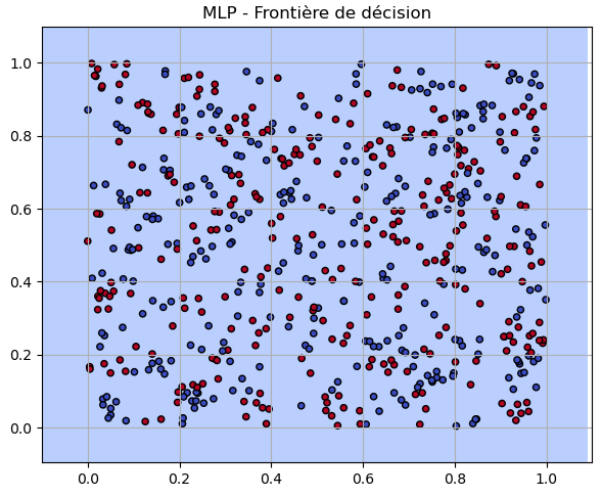


Figure 4: *

Figure 31 :MLP simple : quelques motifs sont capturés, mais pas les séparations fines.

MLP plus profond. L'augmentation du nombre de couches et de neurones permet au réseau de modéliser les ruptures de classe plus finement, mais cela augmente aussi le risque de surapprentissage.

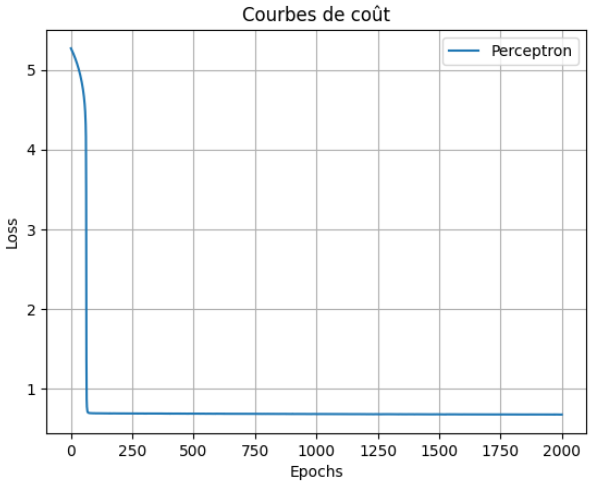


Figure 5: *

Figure 32 : Courbe de coût.

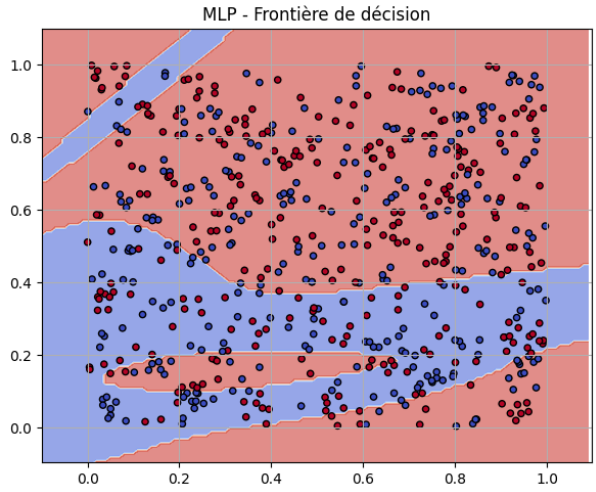


Figure 6: *

Figure 33 :Réseau plus profond : le damier commence à apparaître dans les zones de prédiction..

Architecture plus puissante. Avec une architecture plus large et un entraînement plus long, le modèle parvient à approximer la structure du damier. La séparation des zones reste imparfaite mais le motif global est capturé.

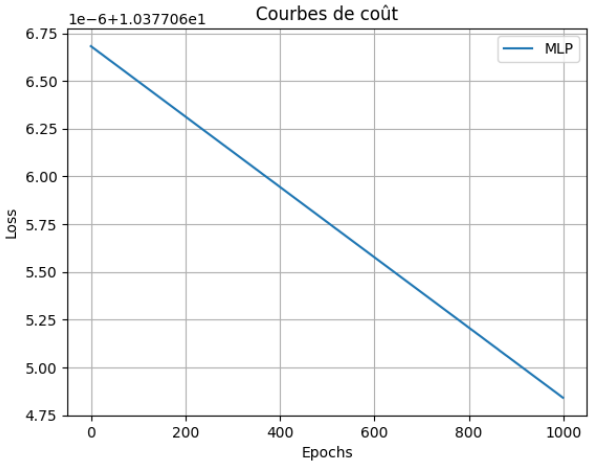


Figure 7: *

Figure 34 : Courbe de coût.

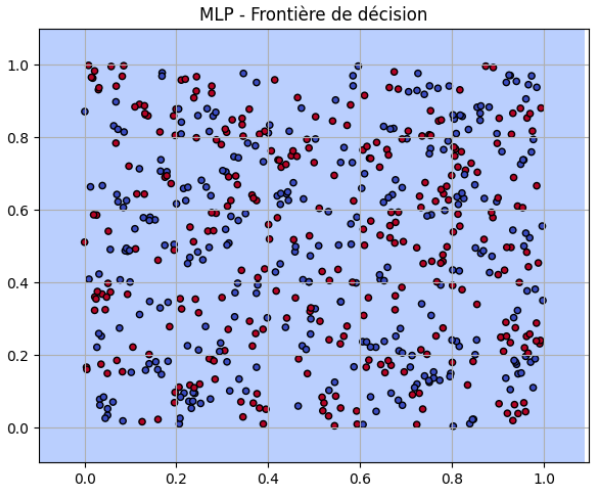


Figure 8: *

Figure 35 :Réseau plus profond : le damier commence à apparaître dans les zones de prédiction..

Cas extrême. En poussant encore davantage la capacité du réseau (ex. : plusieurs couches, plus de neurones, longue convergence), on observe que le modèle sursegmente l'espace d'entrée, produisant une prédiction visuellement très proche du motif original.

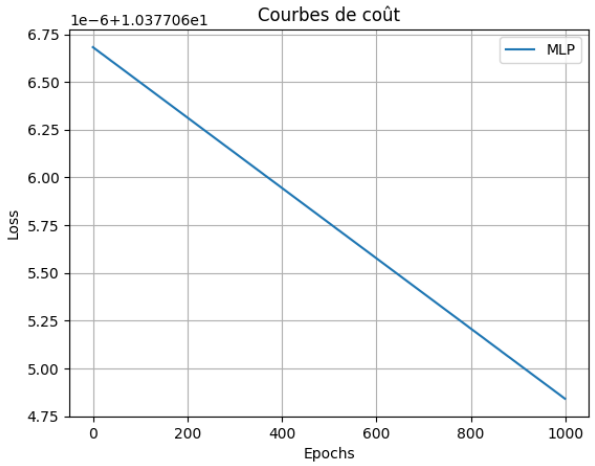


Figure 9: *

Figure 34 : Courbe de coût.

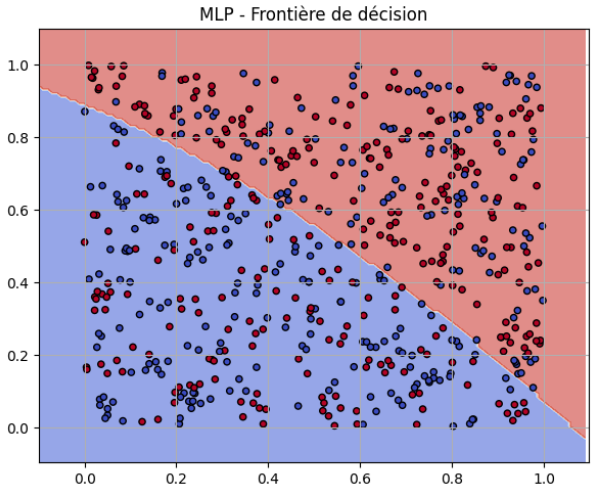


Figure 10: *

Figure 35 :Frontière très irrégulière — le MLP surapprend le motif d'échiquier.

Conclusion. Le motif d'échiquier représente un défi majeur pour les réseaux de neurones, surtout en faible dimension. Cette expérimentation démontre que :

- le perceptron échoue totalement sur ce type de structure,
- un MLP bien configuré peut approximer les motifs avec un taux d'erreur modéré,
- une capacité excessive conduit rapidement au surapprentissage.

Courbes de coût. Nous traçons également les courbes de perte durant l'apprentissage, afin d'évaluer la convergence des différents modèles MLP testés. On observe que malgré la complexité de la tâche, les modèles parviennent à faire décroître leur perte de manière stable.

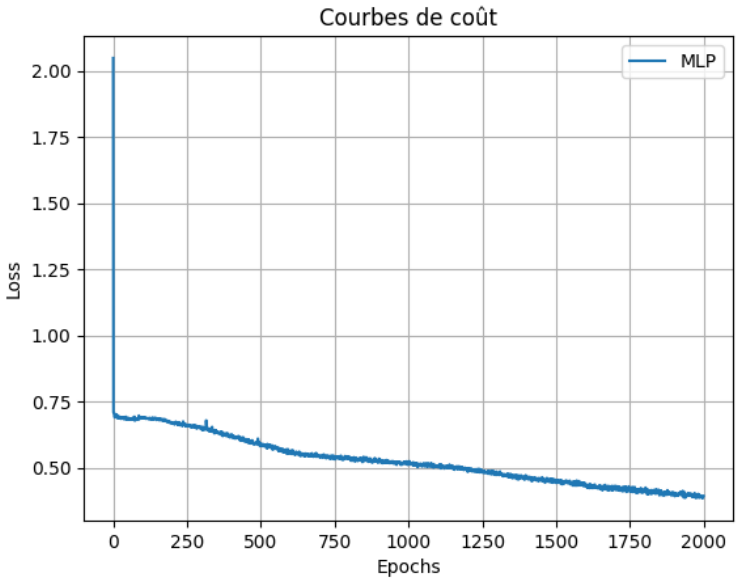


Figure 33 : MLP – perte qui décroît lentement mais régulièrement.

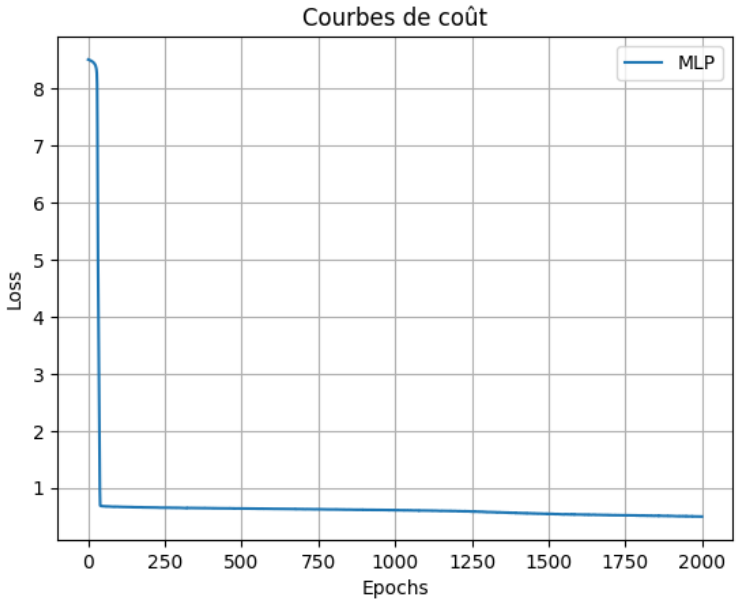


Figure 34 : MLP plus complexe – meilleure descente initiale de la loss.

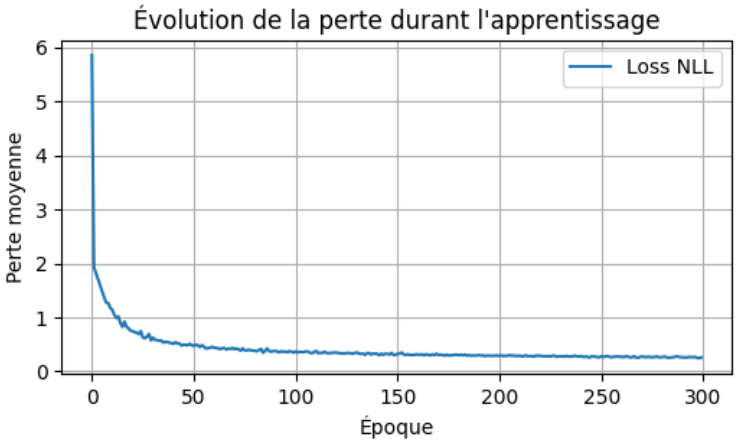


Figure 35 : Autre modèle MLP – stabilité après une perte initiale élevée.

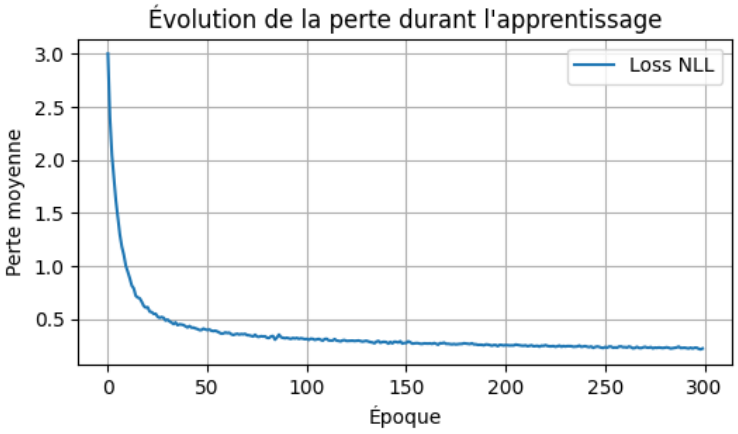


Figure 36 : Exemple de convergence lente mais régulière sur 2000 époques.

Autres visualisations des frontières de décision. Enfin, les figures suivantes illustrent la manière dont la complexité de l'architecture du MLP affecte la finesse de la frontière apprise.

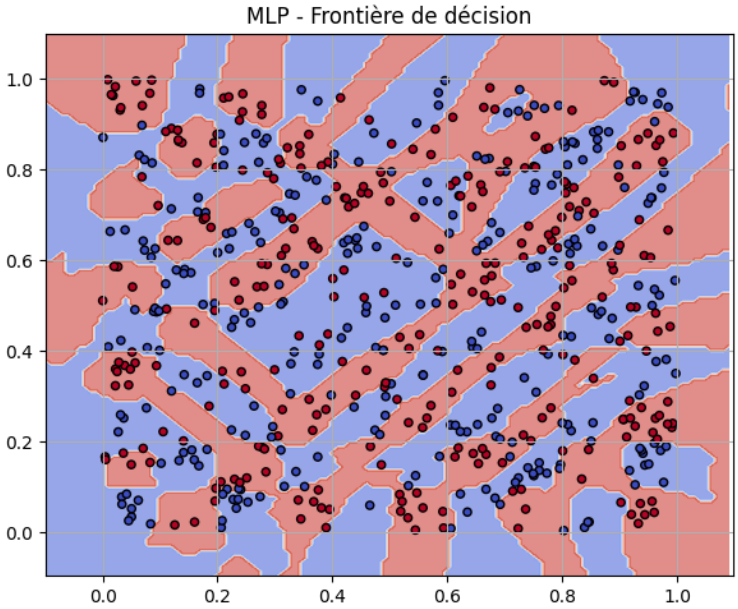


Figure 37 : Frontière dense et segmentée – le réseau s'adapte finement à chaque case du damier.

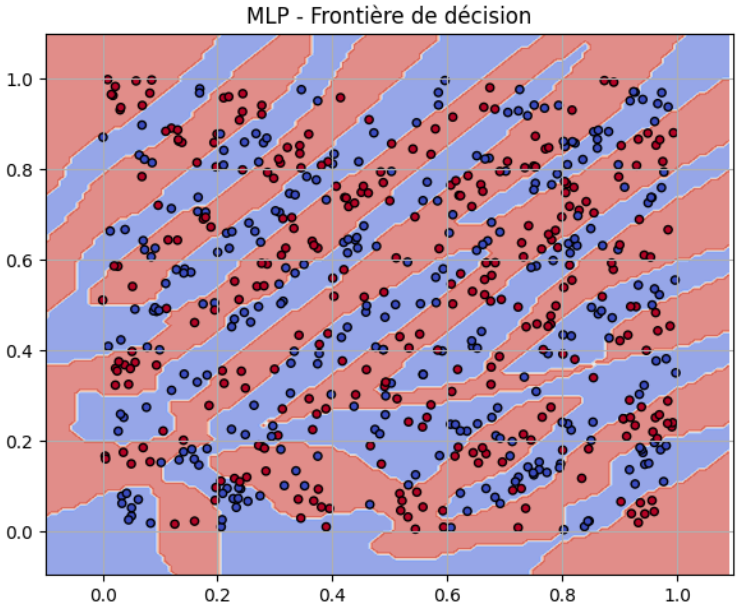


Figure 38 : Un autre exemple d'architecture qui sur-segmente l'espace.

Bilan. Ces expérimentations montrent que :

- Un simple perceptron est incapable de capturer une fonction non linéaire comme celle du motif échiquier.
- Un MLP, même modeste, peut capturer partiellement ce type de structure.

- Plus l'architecture est complexe (profondeur, largeur), plus la frontière apprise est fine, mais au risque de surapprendre.
- Les courbes de coût confirment la capacité d'optimisation du réseau même sur une tâche aussi difficile.

2.3 Auto-encoder : apprentissage non supervisé sur USPS

Les auto-encodeurs sont des réseaux de neurones conçus pour apprendre une représentation compacte des données d'entrée (codage), en les reconstruisant à travers un passage par une couche cachée de dimension inférieure. Ils sont utilisés pour la réduction de dimension, la compression ou encore la détection d'anomalies.

2.3.1 Auto-encoder 1

Dans cette expérimentation, nous construisons un auto-encodeur simple entraîné sur le dataset **USPS** contenant des chiffres manuscrits en niveaux de gris (taille 16x16).

Architecture. L'auto-encodeur est composé de deux parties symétriques :

- un encodeur (`Linear(256, 64)` puis `Tanh()`) qui projette les données dans un espace latent de dimension 64,
- un décodeur (`Linear(64, 256)` puis `Sigmoid()`) qui reconstruit l'image d'entrée.

La fonction de perte utilisée est la **moyenne des erreurs quadratiques (MSE)** entre l'image originale et l'image reconstruite.

Résultats. Le modèle est entraîné durant 200 époques sur les données USPS. On observe une nette baisse de la perte au début de l'entraînement, suivie d'une stabilisation autour d'une valeur constante.

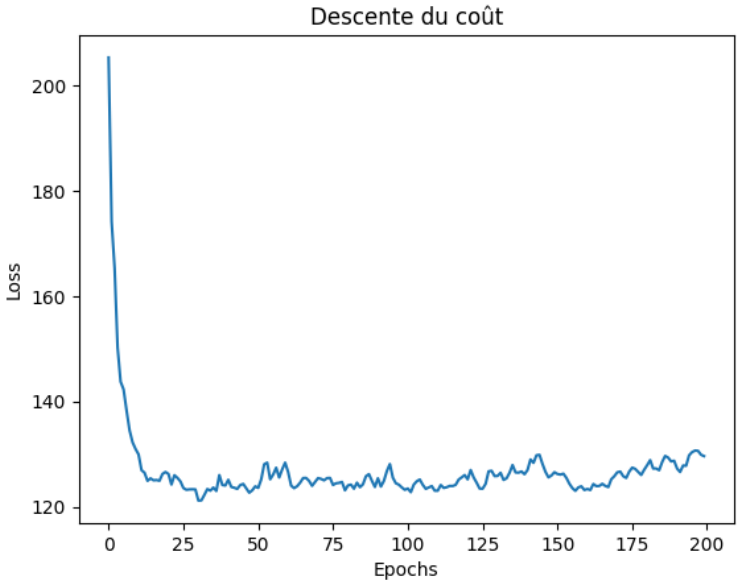


Figure 11: **Figure 39** : Descente du coût (MSE) pendant l'entraînement de l'auto-encodeur.

Qualité des reconstructions. La figure ci-dessous montre 10 chiffres originaux (ligne du haut) et leurs reconstructions générées par l'auto-encodeur (ligne du bas). On observe que les formes globales sont bien reproduites, bien que les contours soient légèrement flous. Cela montre que le réseau a appris une représentation latente fidèle de l'information visuelle principale.

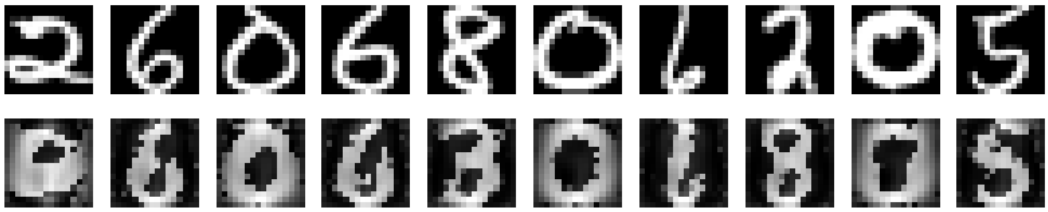


Figure 12: **Figure 40** : Ligne 1 : chiffres originaux. Ligne 2 : reconstruction par l'auto-encodeur.

2.3.2 Auto-encodeur 2

Nous testons ici une seconde version de l'auto-encodeur, avec une architecture similaire mais un entraînement plus poussé (400 époques, meilleur taux d'apprentissage).

Courbe de perte. La convergence est bien plus stable : la perte atteint une valeur très faible (≈ 0.01) sans sursauts.

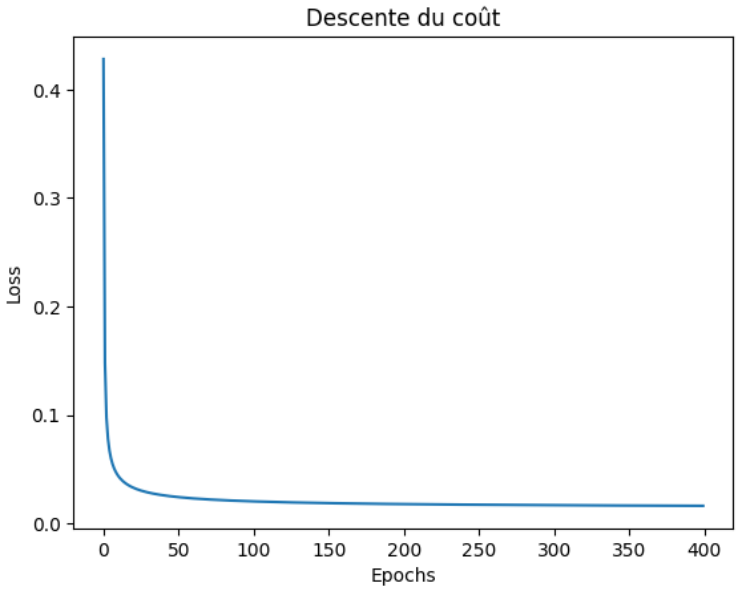


Figure 13: **Figure 41** : Courbe de perte du deuxième auto-encodeur.

Reconstructions. Le modèle reconstruit globalement les contours des chiffres, mais l'effet de flou reste très présent. L'espace latent semble avoir convergé vers une représentation minimale.

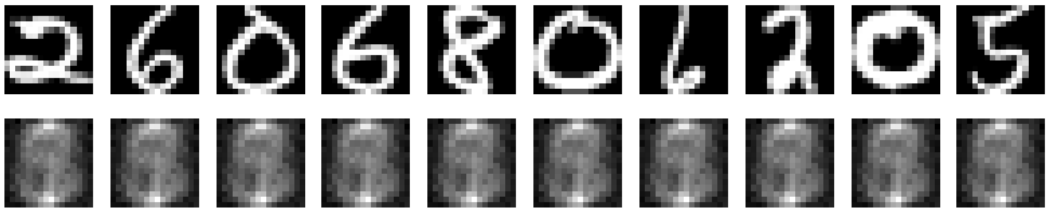


Figure 14: **Figure 42** : Ligne 1 : chiffres originaux. Ligne 2 : reconstructions (auto-encodeur 2).

2.3.3 Auto-encodeur 3

Dans cette dernière expérimentation, nous évaluons différentes combinaisons d'architecture, normalisation, fonction d'activation et de perte, afin de tester la robustesse de notre auto-encodeur.

Architecture. Tous les modèles utilisent la même structure symétrique :

$$\text{Encodeur} : 256 \to 128 \to 64 \to 48 \qquad \text{Decodeur} : 48 \to 64 \to 128 \to 256$$

Exemple 1. Données centrées avec standardisation, fonction d'activation `Tanh` et `MSELoss`.

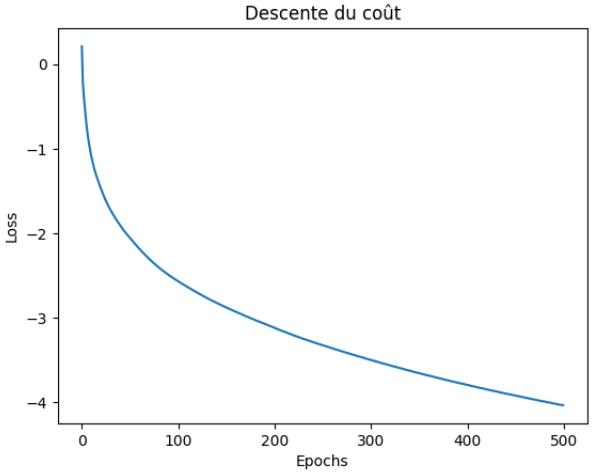


Figure 15: *

Figure 43 : Courbe de coût (centrées + BCELoss).

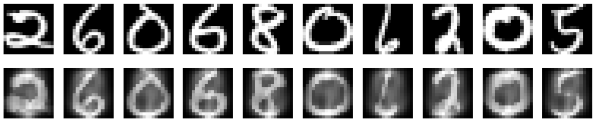


Figure 16: *

Figure 44 : Reconstructions correspondantes.

Exemple 2. Données prétraitées par MinMaxScaler, fonction `BCEELoss`. Ce test permet d'observer l'impact de la normalisation et de la loss sur les résultats visuels.

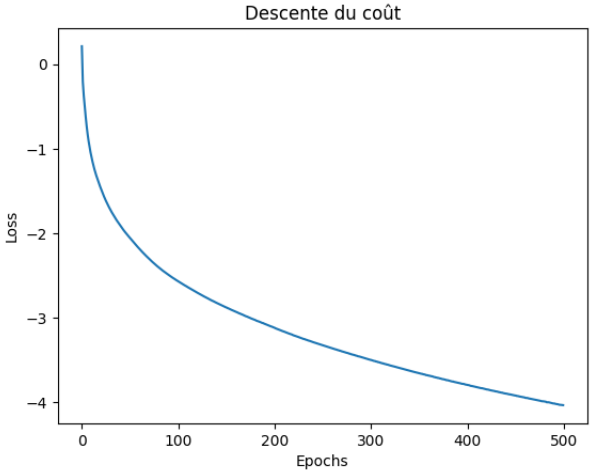


Figure 17: *
Figure 45 : Courbe de coût (MinMax + BCEELoss).

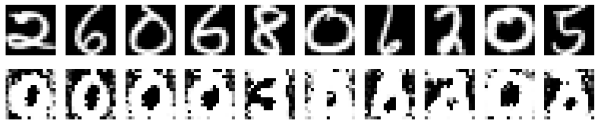


Figure 18: *
Figure 46 : Reconstructions dégradées (saturation, sur-entraînement).

Synthèse. Le tableau ci-dessous résume les différentes configurations testées. On observe que les modèles avec données centrées et `MSELoss` tendent à produire des reconstructions plus douces, tandis que l'utilisation de `BCEELoss` sans une normalisation adaptée peut entraîner des artefacts.

Encoder	Decoder	Données	Init	Activation	Loss	LR	Batch	E
256-128-64-48	48-64-128-256	centrées (standard)	1	Tanh	MSELoss	0.010	64	
256-128-64-48	48-64-128-256	[0, 1]	1	Tanh + Sigmoide	BCELoss	0.005	64	
256-128-64-48	48-64-128-256	centrées (standard)	1	Tanh	BCEELoss	0.010	64	
256-128-64-48	48-64-128-256	MinMax Scaler	1	Tanh	BCEELoss	0.010	64	

Table 2: Configurations testées pour l'entraînement des auto-encodeurs.

2.3.4 Clustering des représentations latentes

Une application classique des auto-encodeurs est l'extraction de représentations compressées des données, exploitables pour des tâches non supervisées telles que le clustering.

Dans cette expérience, nous utilisons les **codes latents** (vecteurs de 48 dimensions issus de la couche bottleneck de l'auto-encodeur) comme **base pour un clustering KMeans**. Les résultats sont projetés en 2D à l'aide de **PCA** (analyse en composantes principales).

Clustering sur les codes. Le premier ensemble de figures montre les clusters prédits par KMeans (gauche) et les vraies classes (droite) sur les représentations issues du premier auto-encodeur.

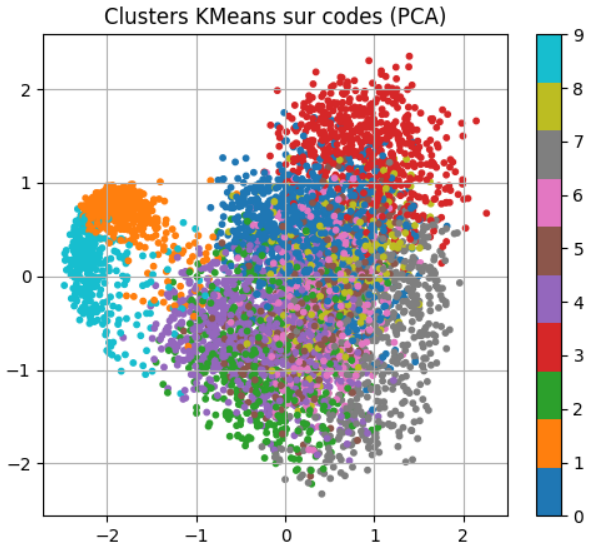


Figure 19: *

Figure 47 : Clusters KMeans sur les codes.

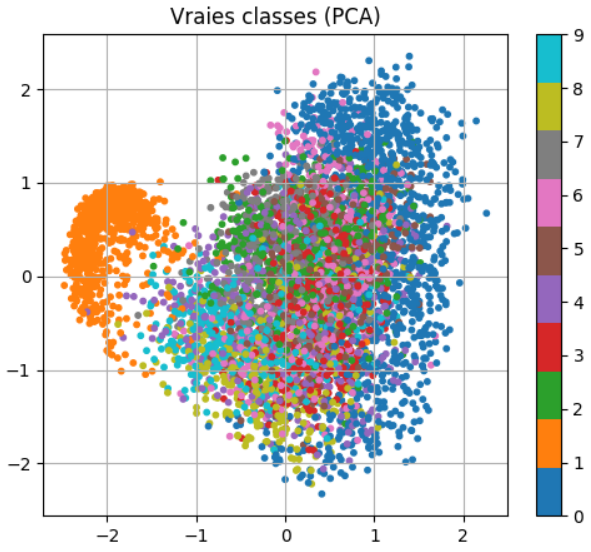


Figure 20: *

Figure 48 : Vraies classes correspondantes.

Reconstructions associées. Les codes ayant permis ce clustering donnent lieu à des reconstructions relativement fidèles.

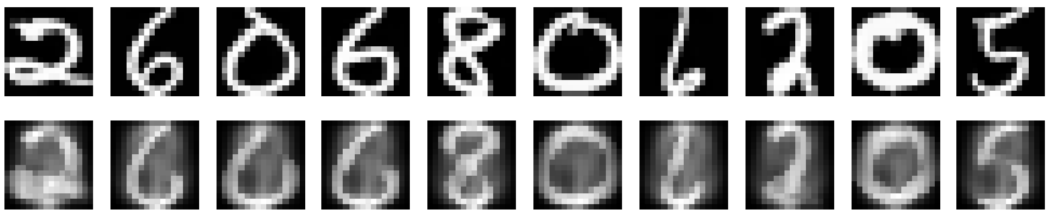


Figure 49 : Reconstructions correspondantes aux codes utilisés.

Comparaison avec d'autres configurations. Les figures suivantes présentent deux autres tests de clustering, avec des codes obtenus via différents traitements (normalisation, architectures. . .).

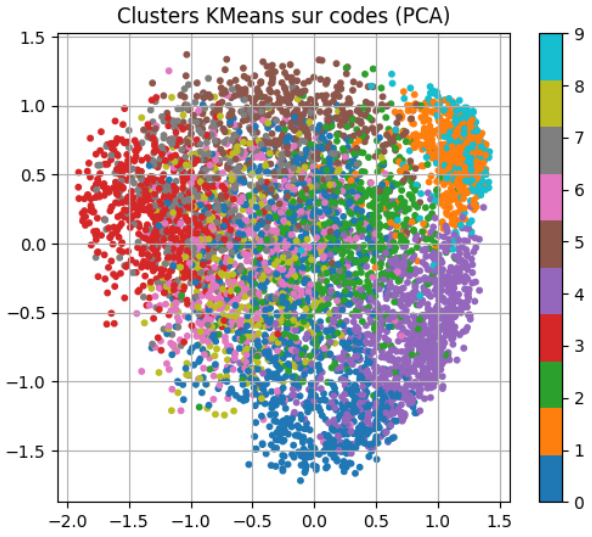


Figure 21: *

Figure 50 : Clustering sur codes (test 2).

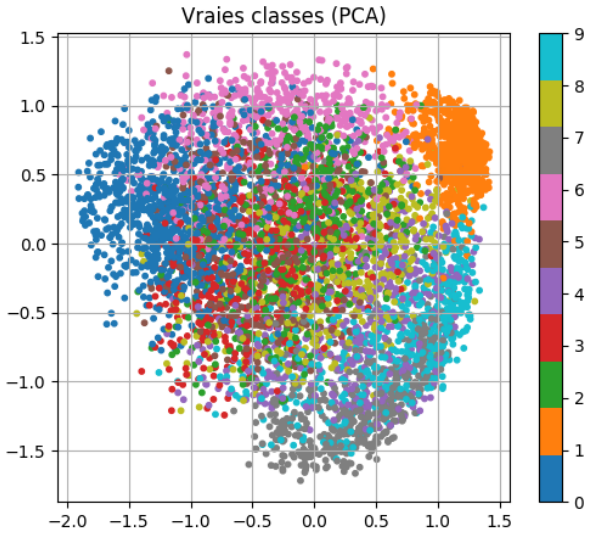


Figure 22: *

Figure 51 : Vraies classes associées (test 2).

Conclusion. Le clustering KMeans sur les représentations latentes offre des résultats prometteurs : bien que les clusters ne correspondent pas parfaitement aux vraies classes, on observe une organisation claire des points dans l'espace projeté. Cela montre que l'auto-encodeur capture une structure informative des chiffres manuscrits, utile en aval pour des tâches non supervisées.

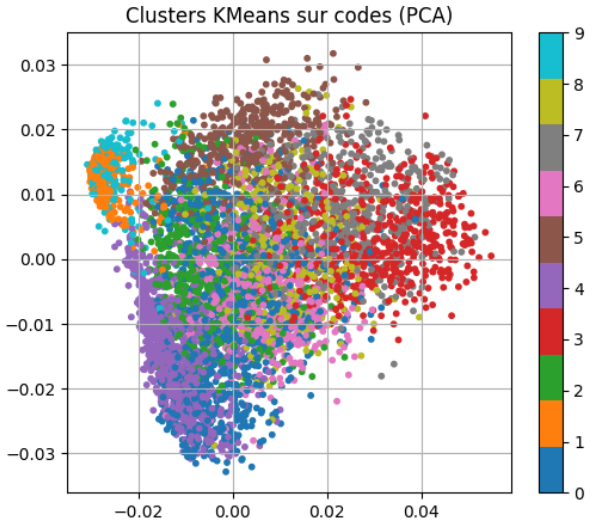


Figure 23: *

Figure 52 : Clustering sur codes (test 3).

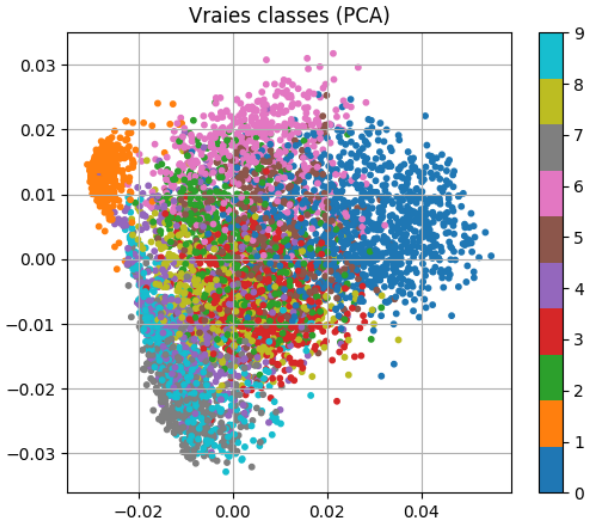


Figure 24: *

Figure 53 : Vraies classes associées (test 3).

2.3.5 Classification appliquée aux visages (YaleFaces)

Nous avons entraîné un auto-encodeur convolutionnel sur le dataset Yale Faces, afin de tester la capacité du réseau à capturer l'information visuelle sur des images en niveaux de gris de visages humains.

Apprentissage. Deux types de pertes ont été testés : la perte MSE et la perte binaire croisée (BCE). Les courbes suivantes illustrent la descente du coût au cours de l'entraînement :

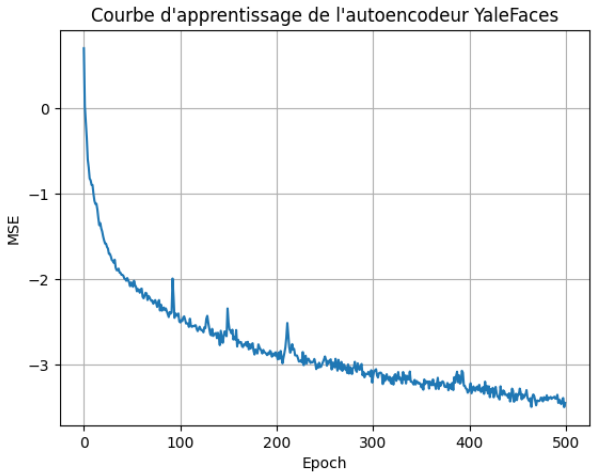


Figure 25: *

Figure 54 : Descente du coût (MSE).

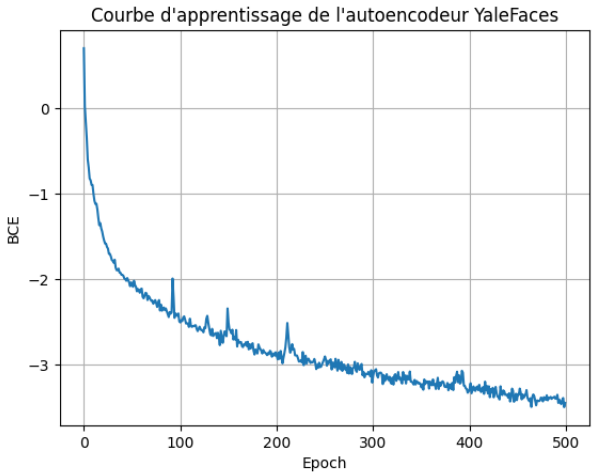


Figure 26: *

Figure 55 : Descente du coût (BCE).

Résultats visuels. Ci-dessous sont présentées les reconstructions d'images après passage par l'auto-encodeur. On remarque que la perte BCE a tendance à donner des sorties binarisées trop agressivement.

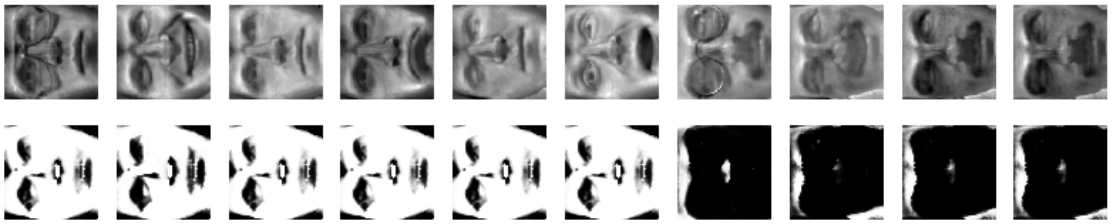


Figure 27: *

Figure 56 : Reconstructions avec MSELoss (en bas).

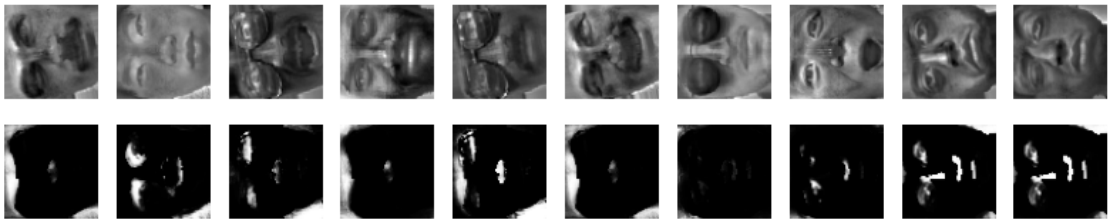


Figure 28: *

Figure 57 : Reconstructions avec BCELoss (en bas).

Early stopping. Nous avons également introduit une stratégie d'**early stopping**, afin d'éviter le surapprentissage. L'apprentissage s'arrête automatiquement lorsque la perte de validation cesse de diminuer.

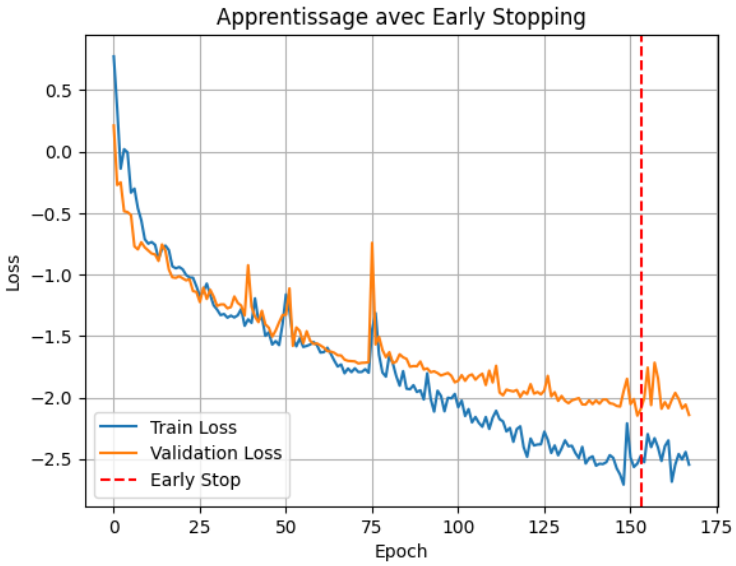


Figure 58 : Apprentissage avec early stopping sur YaleFaces.

Conclusion. Sur des données comme YaleFaces, les auto-encodeurs permettent une compression efficace, mais une bonne normalisation (standardisation autour de 0) est cruciale. Le choix de la fonction de perte a également un fort impact sur la qualité des reconstructions (MSE donne de meilleurs résultats visuels ici que BCE).

Conclusion générale

Dans ce projet, nous avons implémenté et exploré en profondeur plusieurs architectures de réseaux de neurones entièrement *from scratch*. Partant du perceptron simple jusqu'aux MLPs plus profonds et enfin aux auto-encodeurs, nous avons progressivement renforcé les capacités de modélisation de nos modèles.

Nos expérimentations ont mis en lumière plusieurs points clés :

- Le **perceptron simple**, bien que historiquement fondamental, ne permet de résoudre que des problèmes linéaires, ce qui limite son usage sur des tâches réelles.
- Les **MLPs** offrent une expressivité bien plus grande grâce à la superposition de couches et l'introduction de non-linéarités. Ils parviennent à approximer des fonctions complexes, à condition que l'architecture soit bien calibrée.
- Sur des jeux de données synthétiques (moons, blobs, échiquiers), nous avons pu observer comment la profondeur, le choix des activations et les hyperparamètres influencent la précision et la forme de la frontière de décision.
- Les **auto-encodeurs** ont démontré leur capacité à apprendre des représentations compressées utiles, avec une efficacité qui dépend fortement de la normalisation des données, du type de perte et du décodage.
- L'ajout d'outils comme **early stopping** ou l'analyse des **clusters latents** (via PCA + KMeans) montre comment les représentations apprises peuvent être exploitées au-delà de la reconstruction.

D'un point de vue pédagogique, cette approche *from scratch* nous a permis de mieux comprendre les rouages internes des réseaux de neurones, et les mécanismes d'apprentissage profond. Elle constitue une base solide pour aborder des modèles plus complexes (CNN, RNN, transformers...) ou pour déployer des solutions dans des contextes réels.

Pistes d'amélioration et perspectives :

- Implémenter des régularisations (dropout, L2) pour limiter le surapprentissage.
- Ajouter des métriques plus fines pour l'évaluation (f1-score, confusion matrix...).
- Étendre les auto-encodeurs à des modèles variationnels (VAE) pour la génération.
- Appliquer les représentations latentes à des tâches de classification supervisée.

Ce projet marque une étape importante dans notre compréhension pratique de l'apprentissage profond.