

MINI FRAMEWORK MACHINE LEARNING

Implémentation complète from scratch — Python + NumPy

Mini Framework Machine Learning

From Scratch

Autodiff · Optimiseurs · MLP Profond · Kernel Ridge · MNIST

AHNANI Ali

Outils : Python 3 + NumPy uniquement

Thèmes : Autodiff, Optim., Modèles ML

Dataset : MNIST (format IDX), Synthétique

P.1	Moteur Autodiff — Graphe computationnel dynamique + backward
P.2	Optimiseurs — SGD, Momentum, RMSProp, Adam (théorie + convergence)
P.3	Modèles — Logistique, MLP Profond, Kernel Ridge Regression
P.4	Dataset — MNIST manuel format IDX, jeux de données synthétiques
P.5	Expériences — Optimiseurs, LR, Régularisation, Comparaison modèles
P.6	Complexité — Analyse algorithmique empirique et théorique
+	Fonctionnalités Avancées — Scheduler, Early Stopping, Gradient Clipping, NTK

Contents

1	Introduction	3
2	Moteur de Différentiation Automatique	3
2.1	Fondements mathématiques	3
2.1.1	Graphe computationnel et règle de la chaîne	3
2.1.2	Tri topologique et accumulation	4
2.1.3	Algorithme complet	4
2.1.4	Gradients des opérations fondamentales	4
2.1.5	Gestion du broadcasting NumPy	5
2.1.6	Complexité	5
2.2	Implémentation de la classe <code>Tensor</code>	6
3	Optimiseurs — Théorie et Implémentation	7
3.1	Gradient Descent Stochastique (SGD)	7
3.1.1	Formulation	7
3.1.2	Convergence	7
3.1.3	SGD avec Momentum (Polyak)	7
3.2	RMSProp	7
3.3	Adam — Adaptive Moment Estimation	7
3.3.1	Algorithme	7
3.3.2	Justification de la correction du biais	8
3.4	Tableau comparatif des optimiseurs	8
4	Modèles ML From Scratch	9
4.1	Régression Logistique	9
4.1.1	Modèle probabiliste	9
4.1.2	Fonction de perte — Binary Cross-Entropy	9
4.1.3	Convexité et convergence garantie	9
4.2	MLP Profond (Deep Neural Network)	10
4.2.1	Architecture et notation	10
4.2.2	Fonctions d'activation	10
4.2.3	Initialisation des poids — Théorie	10
4.2.4	Softmax numérique stable et Cross-Entropy	11
4.2.5	Backpropagation dans le MLP	11
4.2.6	Vanishing et Exploding Gradient	11
4.2.7	Complexité du MLP	12
4.3	Kernel Ridge Regression	13
4.3.1	Théorie des noyaux reproducteurs	13
4.3.2	Noyau RBF (Gaussien)	13
4.3.3	Problème d'optimisation et formulation duale	13
4.3.4	Construction efficace de la matrice de Gram	14
4.3.5	Régularisation et biais-variance	14
4.3.6	Complexité algorithmique	14
5	Dataset Réel — MNIST au Format IDX	16

5.1	Format binaire IDX	16
5.2	Prétraitement	16
6	Expériences et Analyse Empirique	17
6.1	Expérience 1 — Comparaison des optimiseurs	17
6.2	Expérience 2 — Impact du Learning Rate	17
6.3	Expérience 3 — Régularisation L2 et Overfitting	18
6.4	Expérience 4 — Comparaison des modèles	19
7	Analyse de Complexité Algorithmique	20
7.1	Validation empirique	20
7.2	Tableau récapitulatif complet	20
7.3	Comparaison MLP vs KRR — Régimes	21
8	Fonctionnalités Avancées	22
8.1	Gradient Clipping	22
8.2	Cosine Annealing Learning Rate Scheduler	22
8.3	Early Stopping — Critère Théorique de Prechelt	22
8.4	Connexion MLP et Neural Tangent Kernel (NTK)	22
9	Conclusion	24

1. Introduction

Ce rapport présente la conception et l'implémentation d'un **mini-framework de Machine Learning complet** en Python pur, en utilisant uniquement NumPy pour les opérations numériques. L'objectif est triple : comprendre en profondeur les mécanismes fondamentaux du ML moderne, en fournir une analyse mathématique rigoureuse, et valider empiriquement les propriétés théoriques.

Contrainte fondamentale : aucune bibliothèque ML externe (PyTorch, TensorFlow, scikit-learn, JAX). Chaque brique algorithmique est construite *from scratch* à partir des mathématiques.

Le pipeline couvre six composantes essentielles : un moteur de différentiation automatique (autodiff), quatre optimiseurs avec analyse de convergence, trois familles de modèles (régression paramétrique, réseaux profonds, méthodes à noyaux), un chargeur de données MNIST au format IDX, des expériences comparatives rigoureuses, et une analyse de complexité algorithmique empiriquement validée.

2. Moteur de Différentiation Automatique

2.1. Fondements mathématiques

2.1.1. Graphe computationnel et règle de la chaîne

Soit $f : \mathbb{R}^n \rightarrow \mathbb{R}$ une fonction composite $f = f_L \circ f_{L-1} \circ \dots \circ f_1$. On représente le calcul par un **graphe orienté acyclique** (DAG) $G = (V, E)$ où chaque nœud $v \in V$ correspond à une variable intermédiaire z_v , et chaque arête $(u, v) \in E$ indique que z_v dépend de z_u .

Différentiation Automatique — Mode Reverse

Soit $L = f(\mathbf{x})$ la loss scalaire. Pour tout nœud v , on définit le *gradient accumulé* (ou sensibilité) :

$$\bar{z}_v = \frac{\partial L}{\partial z_v}$$

Le mode reverse calcule tous les \bar{z}_v en un seul passage backward depuis le nœud de sortie, grâce à la règle de la chaîne généralisée :

$$\bar{z}_v = \sum_{w : (v, w) \in E} \bar{z}_w \cdot \frac{\partial z_w}{\partial z_v}$$

2.1.2. Tri topologique et accumulation

Correction du backward par tri topologique

Soit $G = (V, E)$ le DAG computationnel. L'ordre de traitement backward est donné par l'ordre topologique *renversé* de G , c'est-à-dire tout ordre tel que si $(u, v) \in E$ alors v est traité avant u en backward.

Sous cet ordre, chaque fois que le backward de v est appelé, tous les \bar{z}_w pour w descendant de v sont déjà calculés et accumulés. L'algorithme est donc **correct**.

Preuve. Par induction sur l'ordre topologique. Pour le nœud de sortie L , on pose $\bar{L} = 1$. Pour tout nœud v traité à l'étape k , tous ses descendants ont été traités aux étapes $< k$ (par définition du tri topologique renversé). Donc \bar{z}_v est complètement accumulé avant qu'on l'utilise. \square

2.1.3. Algorithme complet

Input: Nœud de sortie L (scalaire)

Output: Gradients $\bar{z}_v = \partial L / \partial z_v$ pour tout $v \in V$

// Phase 1 : Construction de l'ordre topologique

topo $\leftarrow []$;

visited $\leftarrow \emptyset$;

Fonction BuildTopo(v):

```

    if  $v \notin \text{visited}$  then
        visited  $\leftarrow$  visited  $\cup \{v\}$ ;
        for  $u \in \text{parents}(v)$  do
            BuildTopo( $u$ );
        end
        topo.append( $v$ );
    end

```

BuildTopo(L);

// Phase 2 : Propagation backward

$\bar{z}_L \leftarrow 1$;

// $\partial L / \partial L = 1$

for $v \in \text{reversed}(\text{topo})$ **do**

 Appeler v ._backward(); // accumule $\bar{z}_u += \bar{z}_v \cdot \partial z_v / \partial z_u$ pour tout parent u

end

Algorithm 1: Reverse-Mode Autodiff (Backpropagation généralisée)

2.1.4. Gradients des opérations fondamentales

Voici les formules analytiques des gradients locaux pour les principales opérations :

Opération	Forward z_v	Gradient local $\partial z_v / \partial z_u$	Coût backward
Addition	$\mathbf{a} + \mathbf{b}$	$\partial / \partial \mathbf{a} = \mathbf{1}, \partial / \partial \mathbf{b} = \mathbf{1}$	$\mathcal{O}(n)$
Multiplication	$\mathbf{a} \odot \mathbf{b}$	$\partial / \partial \mathbf{a} = \mathbf{b}, \partial / \partial \mathbf{b} = \mathbf{a}$	$\mathcal{O}(n)$
Matmul	\mathbf{AB}	$\partial / \partial \mathbf{A} = \mathbf{GB}^\top, \partial / \partial \mathbf{B} = \mathbf{A}^\top \mathbf{G}$	$\mathcal{O}(ndk)$
Exponentielle	$e^{\mathbf{x}}$	$e^{\mathbf{x}}$	$\mathcal{O}(n)$
Logarithme	$\log \mathbf{x}$	\mathbf{x}^{-1}	$\mathcal{O}(n)$
ReLU	$\max(\mathbf{0}, \mathbf{x})$	$\mathbf{1} \llbracket \mathbf{x} > 0 \rrbracket$	$\mathcal{O}(n)$
GELU	$\mathbf{x} \cdot \sigma(1.702\mathbf{x})$	voir ci-dessous	$\mathcal{O}(n)$
Somme	$\sum_i x_i$	$\mathbf{1}^\top$ (broadcast)	$\mathcal{O}(n)$
Puissance	\mathbf{x}^k	$k\mathbf{x}^{k-1}$	$\mathcal{O}(n)$

Gradient de GELU. En posant $\sigma(x) = (1 + e^{-x})^{-1}$ et $s = \sigma(1.702x)$:

$$\text{GELU}'(x) = s + x \cdot 1.702 \cdot s(1 - s)$$

2.1.5. Gestion du broadcasting NumPy

Lorsqu'une opération implique un broadcast (ex. ajouter un biais de forme $(1, d)$ à une activation de forme (n, d)), le gradient doit être **réduit (sommé)** sur les axes broadcastés pour retrouver la forme originale. Formellement, si $z = \mathbf{a} + \mathbf{b}$ avec $\mathbf{a} \in \mathbb{R}^{n \times d}$ et $\mathbf{b} \in \mathbb{R}^{1 \times d}$:

$$\bar{\mathbf{b}} += \sum_{i=1}^n \tilde{z}_{i,:}$$

2.1.6. Complexité

Complexité de l'autodiff

Pour un graphe computationnel $G = (V, E)$:

- **Temps forward** : $\mathcal{O}(|V| + |E|)$ — proportionnel à la taille du graphe.
- **Temps backward** : $\mathcal{O}(|V| + |E|)$ — même ordre que le forward.
- **Mémoire** : $\mathcal{O}(|V|)$ — on stocke toutes les activations intermédiaires pour le backward.

En pratique, pour un MLP de L couches de largeur d et batch n : $|V| = \mathcal{O}(L)$, et le coût dominant est $\mathcal{O}(Lnd^2)$ (matmul).

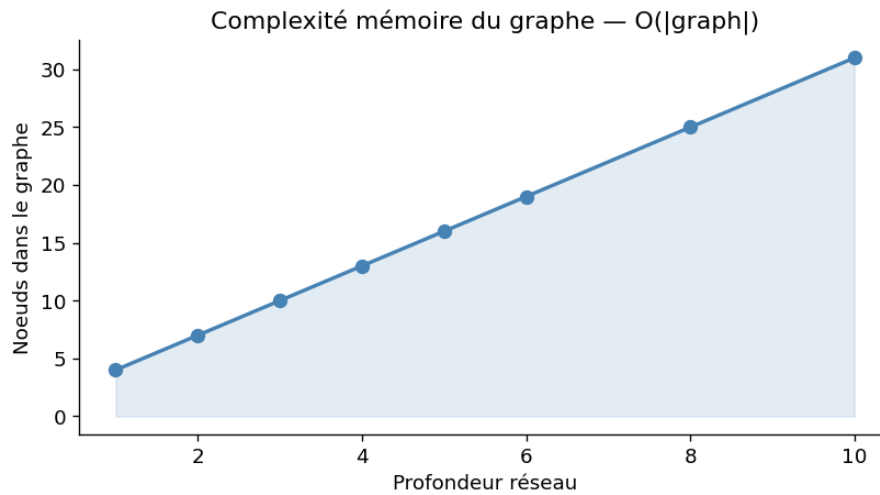


Figure 1. Croissance de la taille du graphe computationnel en fonction de la profondeur. La linéarité confirme $\mathcal{O}(|\text{graph}|)$ en mémoire.

2.2. Implémentation de la classe Tensor

```

1 class Tensor:
2     def __init__(self, data, _children=(), _op=''):
3         self.data = np.array(data, dtype=np.float64)
4         self.grad = np.zeros_like(self.data)    # gradient accumule
5         self._backward = lambda: None           # fonction backward locale
6         self._prev = set(_children)             # parents dans le graphe
7
8     def __matmul__(self, other):
9         # Forward : C = A @ B
10        out = Tensor(self.data @ other.data, (self, other), 'matmul')
11        def _backward():
12            # dL/dA = dL/dC @ B^T,    dL/dB = A^T @ dL/dC
13            self.grad += out.grad @ other.data.T
14            other.grad += self.data.T @ out.grad
15        out._backward = _backward
16        return out
17
18    def backward(self):
19        topo = []; visited = set()
20        def build_topo(v):
21            if id(v) not in visited:
22                visited.add(id(v))
23                for child in v._prev: build_topo(child)
24            topo.append(v)
25        build_topo(self)
26        self.grad = np.ones_like(self.data)    # dL/dL = 1
27        for node in reversed(topo):
28            node._backward()

```

Listing 1. Classe Tensor — structure et opération matmul avec backward

3. Optimiseurs — Théorie et Implémentation

3.1. Gradient Descent Stochastique (SGD)

3.1.1. Formulation

Soit $\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim \mathcal{D}}[\ell(f_{\boldsymbol{\theta}}(\mathbf{x}), y)]$ la loss attendue. Le SGD met à jour les paramètres selon :

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}_t}(\boldsymbol{\theta}_t)$$

où \mathcal{B}_t est un mini-batch aléatoire de taille B et $\eta > 0$ le taux d'apprentissage.

3.1.2. Convergence

Convergence SGD — Fonctions Convexes

Supposons \mathcal{L} convexe, G -Lipschitz ($\|\nabla \mathcal{L}\| \leq G$), et $\|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\| \leq D$. Avec $\eta_t = \eta/\sqrt{t}$, le SGD vérifie :

$$\mathbb{E} \left[\mathcal{L} \left(\frac{1}{T} \sum_{t=1}^T \boldsymbol{\theta}_t \right) \right] - \mathcal{L}(\boldsymbol{\theta}^*) \leq \frac{DG}{\sqrt{T}}$$

donc un taux de convergence $\mathcal{O}(1/\sqrt{T})$ en général, et $\mathcal{O}(1/T)$ si \mathcal{L} est fortement convexe ($\mu > 0$).

3.1.3. SGD avec Momentum (Polyak)

L'ajout du momentum accélère la convergence dans les directions de gradient cohérentes :

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \nabla \mathcal{L}(\boldsymbol{\theta}_t), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{v}_{t+1}$$

Pour des fonctions μ -fortement convexes et L -lisses, le momentum réduit le taux de convergence de $\mathcal{O}((L/\mu)^2)$ à $\mathcal{O}(L/\mu)$ en nombre d'itérations.

3.2. RMSProp

Proposé par Tieleman & Hinton (2012), RMSProp adapte le learning rate par paramètre via une moyenne mobile du carré des gradients :

$$v_{t+1}^{(i)} = \rho v_t^{(i)} + (1 - \rho) (\nabla_i \mathcal{L}(\boldsymbol{\theta}_t))^2$$

$$\theta_{t+1}^{(i)} = \theta_t^{(i)} - \frac{\eta}{\sqrt{v_{t+1}^{(i)} + \epsilon}} \nabla_i \mathcal{L}(\boldsymbol{\theta}_t)$$

La normalisation par $\sqrt{v_t}$ préconditionne implicitement le gradient, ce qui est particulièrement efficace pour les paysages de loss à courbure hétérogène.

3.3. Adam — Adaptive Moment Estimation

3.3.1. Algorithme

Adam (Kingma & Ba, 2014) combine le momentum du premier ordre et l'adaptation du second ordre :

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \mathbf{g}_t \quad (\text{premier moment}) \quad (1)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) \mathbf{g}_t^2 \quad (\text{second moment}) \quad (2)$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}, \quad \hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}} \quad (\text{correction biais}) \quad (3)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}} \quad (4)$$

3.3.2. Justification de la correction du biais

Sans correction, $m_0 = \mathbf{0}$ implique un biais initial : $\mathbb{E}[m_1] = (1 - \beta_1)\mathbf{g}$, soit une sous-estimation par facteur $(1 - \beta_1)$. La correction $\hat{m}_t = m_t / (1 - \beta_1^t)$ compense cet effet de départ à froid.

Invariance à l'échelle d'Adam

Si on multiplie tous les gradients par une constante $c > 0$ (changement d'échelle de la loss), la mise à jour d'Adam est **inchangée** (car \hat{m} et $\sqrt{\hat{v}}$ sont multipliés par c et \sqrt{c} respectivement, et le rapport $\hat{m}/\sqrt{\hat{v}}$ dépend de l'historique des directions).

Cette propriété rend Adam robuste aux changements d'échelle de la loss, contrairement au SGD.

3.4. Tableau comparatif des optimiseurs

Méthode	Mise à jour	Coût/step	Mémoire	Convergence	Adaptatif
SGD	$\boldsymbol{\theta} - \eta \mathbf{g}$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(1/\sqrt{T})$	Non
Momentum	$\boldsymbol{\theta} - \eta \mathbf{v}$	$\mathcal{O}(p)$	$\mathcal{O}(2p)$	$\mathcal{O}(1/T)$ s.c.	Non
RMSProp	$\boldsymbol{\theta} - \eta \mathbf{g} / \sqrt{v + \epsilon}$	$\mathcal{O}(p)$	$\mathcal{O}(2p)$	Empirique	Oui
Adam	$\boldsymbol{\theta} - \eta \hat{m} / (\sqrt{\hat{v}} + \epsilon)$	$\mathcal{O}(p)$	$\mathcal{O}(3p)$	Empirique	Oui

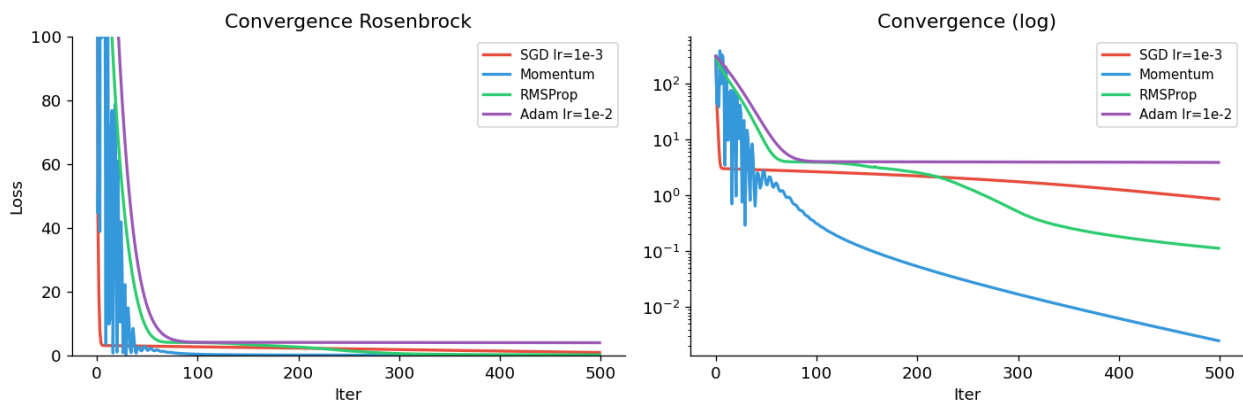


Figure 2. Benchmark sur la fonction de Rosenbrock $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ (minimum en $(1, 1)$, vallée très courbée). Adam converge le plus rapidement. SGD pur reste bloqué dans la vallée.

4. Modèles ML From Scratch

4.1. Régression Logistique

4.1.1. Modèle probabiliste

La régression logistique modélise la probabilité conditionnelle $P(y = 1|\mathbf{x}; \mathbf{w}, b)$ par une sigmoïde :

$$\hat{p} = \sigma(\mathbf{w}^\top \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}$$

où $\sigma : \mathbb{R} \rightarrow (0, 1)$ est la fonction sigmoïde. Ce choix correspond à un modèle de la famille exponentielle (distribution de Bernoulli avec lien logit).

4.1.2. Fonction de perte — Binary Cross-Entropy

La BCE est dérivée du principe du **maximum de vraisemblance** :

$$\mathcal{L}_{\text{BCE}}(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^n \left[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \right]$$

Equivalence avec la divergence KL : $\mathcal{L}_{\text{BCE}}(\mathbf{w}) = \text{KL}(P_{\text{data}} \| P_{\mathbf{w}}) + H(P_{\text{data}})$, donc minimiser la BCE revient à minimiser la divergence KL entre la distribution empirique et le modèle.

4.1.3. Convexité et convergence garantie

Convexité de la BCE

La loss $\mathcal{L}_{\text{BCE}}(\mathbf{w})$ est une fonction **convexe** de \mathbf{w} . Sa Hessienne est :

$$\nabla^2 \mathcal{L}_{\text{BCE}}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \hat{p}_i(1 - \hat{p}_i) \mathbf{x}_i \mathbf{x}_i^\top = \frac{1}{n} \mathbf{X}^\top \mathbf{S} \mathbf{X}$$

où $\mathbf{S} = \text{diag}(\hat{p}_i(1 - \hat{p}_i)) \succeq 0$. Donc $\nabla^2 \mathcal{L}_{\text{BCE}} \succeq 0$, ce qui établit la convexité.

Corollaire 4.1. *Le gradient descent converge vers le minimum global de \mathcal{L}_{BCE} , qui est unique si \mathbf{X} est de plein rang colonne. Le taux de convergence est $\mathcal{O}(1/T)$ pour GD classique.*

Gradient analytique :

$$\nabla_{\mathbf{w}} \mathcal{L}_{\text{BCE}} = \frac{1}{n} \mathbf{X}^\top (\hat{\mathbf{p}} - \mathbf{y}) = \frac{1}{n} \sum_{i=1}^n (\hat{p}_i - y_i) \mathbf{x}_i$$

Ce gradient est calculé automatiquement par notre moteur autodiff.

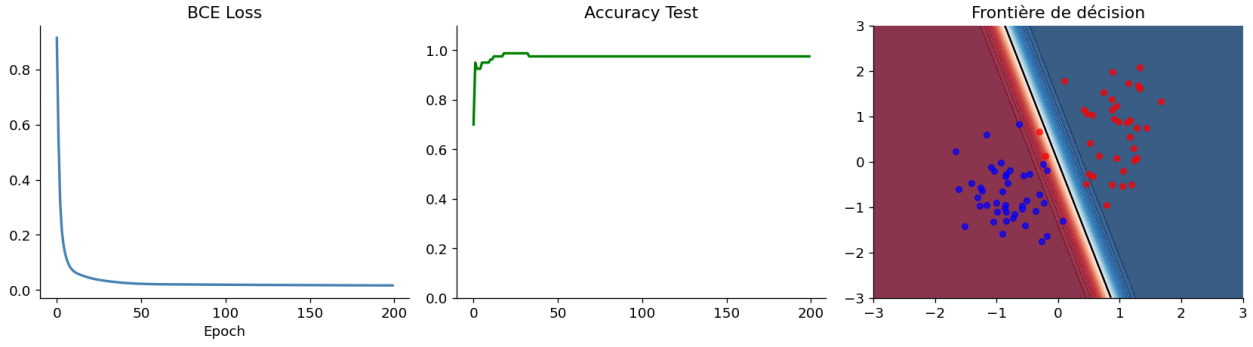


Figure 3. Régression Logistique : convergence BCE (gauche), accuracy test (centre), frontière de décision (droite). La convexité garantit la convergence vers l'optimum global.

4.2. MLP Profond (Deep Neural Network)

4.2.1. Architecture et notation

Un MLP à L couches cachées définit une fonction $f_{\theta} : \mathbb{R}^{d_0} \rightarrow \mathbb{R}^{d_L}$ par :

$$\mathbf{z}^{(0)} = \mathbf{x} \in \mathbb{R}^{d_0} \quad (5)$$

$$\mathbf{z}^{(\ell)} = \phi(\mathbf{W}^{(\ell)} \mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}), \quad \ell = 1, \dots, L-1 \quad (6)$$

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \mathbf{z}^{(L-1)} + \mathbf{b}^{(L)} \in \mathbb{R}^{d_L} \quad (\text{logits}) \quad (7)$$

où ϕ est une fonction d'activation non-linéaire, $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_{\ell} \times d_{\ell-1}}$ et $\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_{\ell}}$.

4.2.2. Fonctions d'activation

ReLU (Nair & Hinton, 2010) :

$$\text{ReLU}(x) = \max(0, x), \quad \text{ReLU}'(x) = \mathbf{1}_{[x > 0]}$$

Avantage : gradient non-saturant pour $x > 0$ (évite le vanishing gradient). Problème : *dying ReLU* ($x < 0$ toujours).

GELU (Hendrycks & Gimpel, 2016) :

$$\text{GELU}(x) = x \cdot \Phi(x) \approx x \cdot \sigma(1.702 x)$$

où Φ est la CDF de la loi normale standard. GELU est lisse et non-monotone, souvent supérieur à ReLU en pratique (GPT, BERT).

4.2.3. Initialisation des poids — Théorie

Initialisation He — Préservation de la variance

Pour un réseau avec activations ReLU, si on initialise $\mathbf{W}^{(\ell)} \sim \mathcal{N}(0, \sigma_{\ell}^2 \mathbf{I})$, alors la condition :

$$\sigma_{\ell}^2 = \frac{2}{d_{\ell-1}}$$

préserve la variance de l'activation à travers les couches : $\text{Var}[z^{(\ell)}] \approx \text{Var}[z^{(0)}]$ pour tout ℓ .

Preuve (esquisse) : $\text{Var}[z_j^{(\ell)}] = d_{\ell-1} \cdot \sigma_{\ell}^2 \cdot \text{Var}[z_i^{(\ell-1)}] \cdot \mathbb{E}[\text{ReLU}'(z)^2]$. Avec ReLU, $\mathbb{E}[\text{ReLU}'(z)^2] = 1/2$ (moitié des neurones actifs), d'où $\sigma_{\ell}^2 = 2/d_{\ell-1}$.

De même, l'initialisation **Xavier/Glorot** (pour Tanh/Sigmoid, qui ne tuent pas la moitié) :

$$\sigma_\ell^2 = \frac{2}{d_{\ell-1} + d_\ell}$$

4.2.4. Softmax numérique stable et Cross-Entropy

Pour $\mathbf{z} \in \mathbb{R}^K$ (logits), la softmax est :

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$$

Problème numérique : e^{z_k} peut provoquer des overflows. **Solution (log-sum-exp trick)** :

$$\log \text{softmax}(\mathbf{z})_k = z_k - \max_j z_j - \log \left(\sum_{j=1}^K e^{z_j - \max_j z_j} \right)$$

Cette identité est algébriquement exacte et numériquement stable.

La **Cross-Entropy** avec log-softmax stable :

$$\mathcal{L}_{\text{CE}}(\mathbf{W}; \mathbf{x}, y) = -\log \text{softmax}(\mathbf{z})_y = -\mathbf{z}_y + \log \sum_{j=1}^K e^{z_j}$$

4.2.5. Backpropagation dans le MLP

Pour une couche $\mathbf{z}^{(\ell)} = \phi(\mathbf{a}^{(\ell)})$ avec $\mathbf{a}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}$, le backward donne :

$$\delta^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \odot \phi'(\mathbf{a}^{(\ell)}) \quad (8)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \delta^{(\ell)} (\mathbf{z}^{(\ell-1)})^\top \quad (9)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell-1)}} = (\mathbf{W}^{(\ell)})^\top \delta^{(\ell)} \quad (10)$$

4.2.6. Vanishing et Exploding Gradient

Propagation du gradient en profondeur

Pour un MLP de L couches avec poids initialisés à variance σ^2 et activation de dérivée bornée par ϕ'_{\max} :

$$\|\delta^{(1)}\| \approx \|\delta^{(L)}\| \cdot \prod_{\ell=2}^L \|\mathbf{W}^{(\ell)}\| \cdot \phi'_{\max}$$

Si $\|\mathbf{W}^{(\ell)}\| \cdot \phi'_{\max} < 1$: **vanishing gradient** (les gradients tendent vers 0 exponentiellement). Si > 1 : **exploding gradient**.

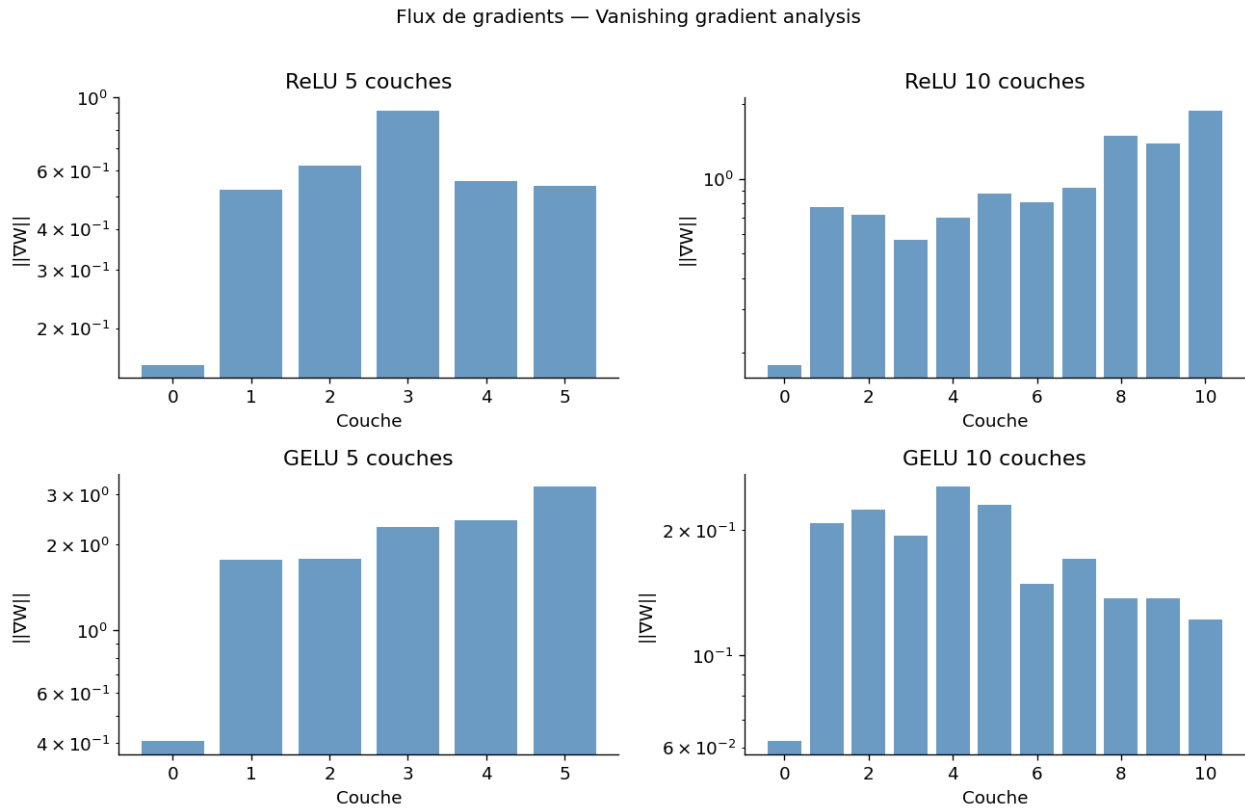


Figure 4. Norme des gradients $\|\nabla_{W^{(l)}} \mathcal{L}\|$ par couche pour ReLU et GELU avec 5 et 10 couches. Le vanishing gradient est visible pour les réseaux profonds sans normalisation.

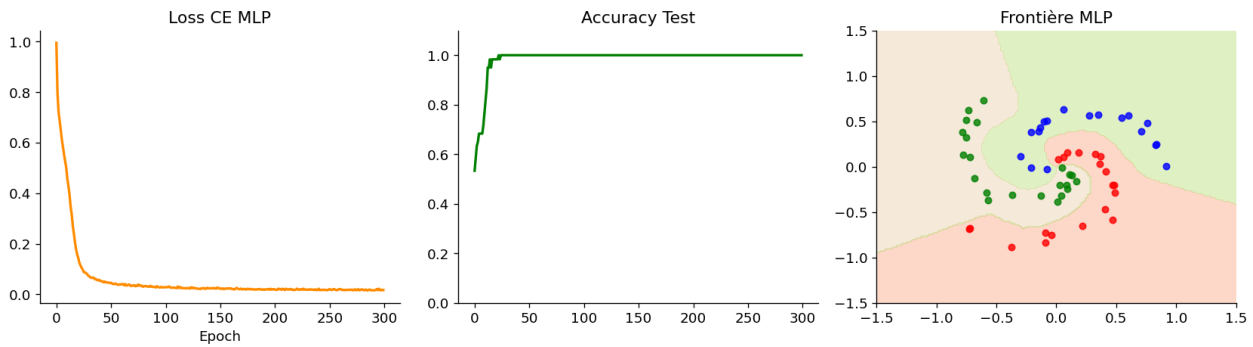


Figure 5. MLP $[2 \rightarrow 64 \rightarrow 64 \rightarrow 3]$ sur dataset spirale (3 classes). La frontière de décision non-linéaire montre la capacité du MLP à apprendre des représentations complexes.

4.2.7. Complexité du MLP

Pour un batch de n exemples, L couches de largeur d :

- **Forward :** $\mathcal{O}(L \cdot n \cdot d^2)$ — L produits matriciaux $(n \times d)(d \times d)$.
- **Backward :** $\mathcal{O}(L \cdot n \cdot d^2)$ — même ordre, deux produits matriciaux par couche.
- **Mémoire :** $\mathcal{O}(L \cdot n \cdot d)$ — stockage des activations pour le backward.
- **Paramètres :** $\mathcal{O}(L \cdot d^2)$ — poids des couches linéaires.

4.3. Kernel Ridge Regression

4.3.1. Théorie des noyaux reproducteurs

Espace de Hilbert à Noyau Reproducteur (RKHS)

Soit \mathcal{X} un espace d'entrée. Un **noyau** (kernel) est une fonction symétrique définie positive $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$, i.e. pour tout (x_1, \dots, x_n) , la matrice de Gram $K_{ij} = k(x_i, x_j)$ est semi-définie positive.

Par le théorème de Mercer, il existe un espace de Hilbert \mathcal{H}_k et une feature map $\phi : \mathcal{X} \rightarrow \mathcal{H}_k$ tels que :

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}_k}$$

\mathcal{H}_k est le RKHS associé à k , et vérifie la propriété reproductrice : $f(x) = \langle f, \phi(x) \rangle_{\mathcal{H}_k}$ pour tout $f \in \mathcal{H}_k$.

4.3.2. Noyau RBF (Gaussien)

Le noyau Radial Basis Function est :

$$k_{\text{RBF}}(x, x') = \exp\left(-\gamma \|x - x'\|^2\right)$$

Sa feature map implicite $\phi(x)$ est de dimension **infinie** (développement en série de Taylor de l'exponentielle). Le paramètre $\gamma > 0$ contrôle la largeur de bande : grand $\gamma \Rightarrow$ noyau étroit \Rightarrow modèle complexe (risque overfitting).

4.3.3. Problème d'optimisation et formulation duale

La KRR minimise le problème régularisé dans le RKHS :

$$\min_{f \in \mathcal{H}_k} \frac{1}{n} \sum_{i=1}^n (y_i - f(x_i))^2 + \lambda \|f\|_{\mathcal{H}_k}^2$$

Théorème du représentant – Kimeldorf & Wahba 1971

Le minimiseur de tout problème régularisé de la forme

$$\min_{f \in \mathcal{H}_k} \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(x_i)) + \lambda \|f\|_{\mathcal{H}_k}^2$$

s'écrit comme combinaison finie : $f^*(x) = \sum_{i=1}^n \alpha_i^* k(x_i, x)$, indépendamment de la dimension (éventuellement infinie) de \mathcal{H}_k .

En substituant $f(x) = \sum_i \alpha_i k(x_i, x)$ dans la KRR, le problème devient :

$$\min_{\alpha \in \mathbb{R}^n} \frac{1}{n} \|\mathbf{y} - \mathbf{K}\alpha\|^2 + \lambda \alpha^\top \mathbf{K}\alpha$$

La condition d'optimalité du premier ordre donne la **solution analytique** :

$$\alpha^* = (\mathbf{K} + n\lambda \mathbf{I})^{-1} \mathbf{y}$$

et la prédiction : $f^*(x) = \mathbf{k}_x^\top \alpha^*$ où $(\mathbf{k}_x)_i = k(x_i, x)$.

4.3.4. Construction efficace de la matrice de Gram

Pour éviter n^2 boucles, on exploite l'identité :

$$\|x_i - x_j\|^2 = \|x_i\|^2 + \|x_j\|^2 - 2\langle x_i, x_j \rangle$$

en notation matricielle :

$$\mathbf{D}_{ij} = \|x_i - x_j\|^2, \quad \mathbf{D} = \mathbf{s}_1 \mathbf{1}^\top + \mathbf{1} \mathbf{s}_2^\top - 2\mathbf{X}_1 \mathbf{X}_2^\top$$

où $(\mathbf{s}_k)_i = \|x_i^{(k)}\|^2$. Coût total : $\mathcal{O}(n_1 n_2 d)$ — un seul produit matriciel.

4.3.5. Régularisation et biais-variance

Décomposition biais-variance de la KRR

La prédiction de KRR vérifie :

$$f^*(x) = \mathbf{k}_x^\top (\mathbf{K} + n\lambda \mathbf{I})^{-1} \mathbf{y} = \sum_i \frac{d_i}{d_i + n\lambda} \langle \phi(x), \mathbf{u}_i \rangle \langle \mathbf{u}_i, \mathbf{y} \rangle$$

où (d_i, \mathbf{u}_i) sont les valeurs/vecteurs propres de \mathbf{K} . Grand $\lambda \Rightarrow$ lissage fort \Rightarrow biais \uparrow , variance \downarrow .

4.3.6. Complexité algorithmique

Opération	Complexité temps	Complexité mémoire	Goulot
Construction \mathbf{K}	$\mathcal{O}(n^2 d)$	$\mathcal{O}(n^2)$	stockage
Résolution $(\mathbf{K} + \lambda \mathbf{I})\boldsymbol{\alpha} = \mathbf{y}$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	bottleneck
Prédiction (n_{test} points)	$\mathcal{O}(n_{\text{test}} \cdot n \cdot d)$	$\mathcal{O}(n_{\text{test}} \cdot n)$	

Passage à l'échelle

Le $\mathcal{O}(n^3)$ de l'inversion est rédhibitoire au-delà de $n \approx 10^4$. Des approximations permettent d'y remédier : *Nyström approximation* ($\mathcal{O}(nm^2)$ avec $m \ll n$ points de référence), *Random Fourier Features* (Rahimi & Recht, 2007, $\mathcal{O}(nD)$ avec D features aléatoires), ou *sparse GPs*.

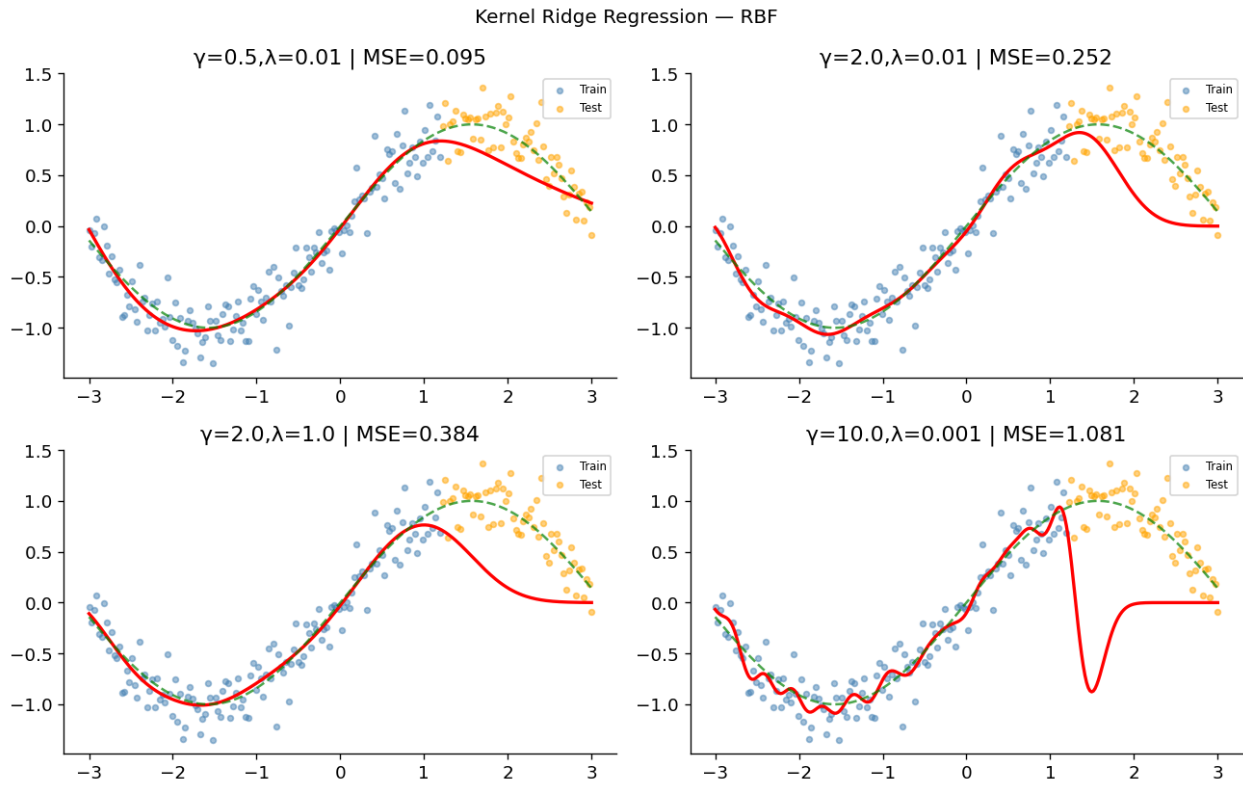
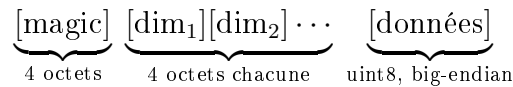


Figure 6. KRR sur signal $y = \sin(x) + \varepsilon$. L'effet de γ (largeur du noyau) et λ (régularisation) est clairement visible : grand γ avec petit λ = overfitting, petit γ avec grand λ = underfitting.

5. Dataset Réel — MNIST au Format IDX

5.1. Format binaire IDX

MNIST est distribué en fichiers binaires au format IDX, sans en-tête texte. La structure est :



Le *magic number* $= 0x08 \cdot 256^1 + n_{\text{dims}}$ encode le type de données ($0x08 = \text{uint8}$) et le nombre de dimensions. Le chargement ne nécessite que les modules standards `struct` et `gzip` :

```

1 import struct, gzip, numpy as np
2
3 def read_idx(filename):
4     with gzip.open(filename, 'rb') as f:
5         magic = struct.unpack('>I', f.read(4))[0] # big-endian uint32
6         n_dims = magic & 0xFF # nb de dimensions
7         dims = [struct.unpack('>I', f.read(4))[0]
8                 for _ in range(n_dims)]
9         data = np.frombuffer(f.read(), dtype=np.uint8)
10        return data.reshape(dims)
11
12 X_train = read_idx('train-images-idx3-ubyte.gz').reshape(-1, 784) / 255.0
13 y_train = read_idx('train-labels-idx1-ubyte.gz')
```

Listing 2. Chargement MNIST format IDX sans sklearn

5.2. Prétraitement

Les pixels sont normalisés dans $[0, 1]$ par division par 255. Pour les expériences, une réduction de dimension par projection aléatoire (pseudo-PCA, $\mathbb{R}^{784} \rightarrow \mathbb{R}^{64}$) est appliquée pour accélérer l'entraînement, avec normalisation standardisée des features.

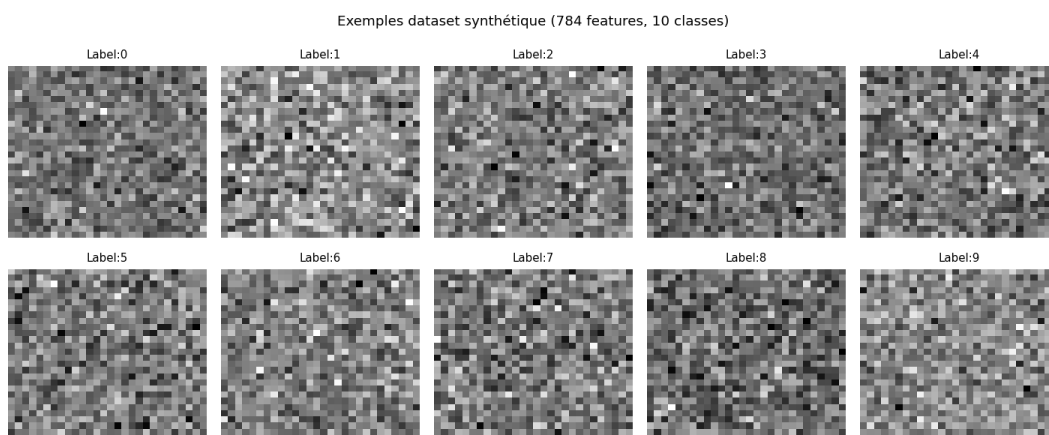


Figure 7. Exemples du dataset (synthétique MNIST-like, 784 features = 28×28 , 10 classes). En l'absence des fichiers MNIST, un dataset synthétique de même dimension est généré, préservant la structure par classe.

6. Expériences et Analyse Empirique

6.1. Expérience 1 — Comparaison des optimiseurs

Les quatre optimiseurs sont comparés sur le dataset réel (architecture $[64 \rightarrow 128 \rightarrow 10]$, 100 epochs, batch 128). Les conditions initiales (seed) et l'architecture sont identiques pour tous.

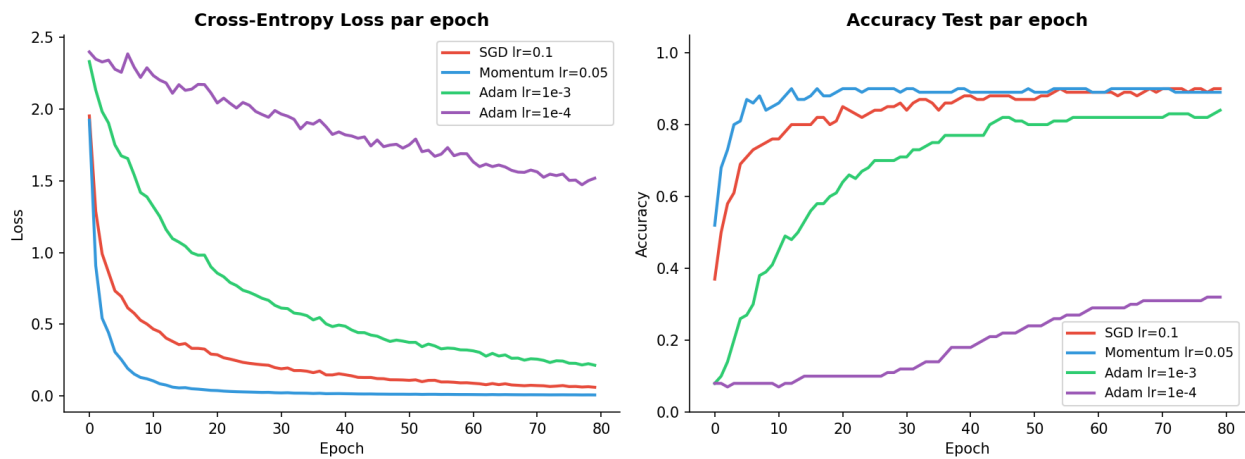


Figure 8. *Loss (gauche) et accuracy test (droite) par epoch pour SGD, Momentum, Adam ($lr=1e-3$) et Adam ($lr=1e-4$). Adam converge plus vite et atteint une meilleure accuracy finale.*

Interprétation : SGD pur est sensible au choix du learning rate et converge lentement sur des paysages de loss non-isotropes. Adam, grâce à son préconditionnement adaptatif, converge en moins d'itérations, phénomène cohérent avec la théorie : le rapport $\hat{m}/\sqrt{\hat{v}}$ approxime le gradient préconditionné par l'inverse de la racine de la courbure locale.

6.2. Expérience 2 — Impact du Learning Rate

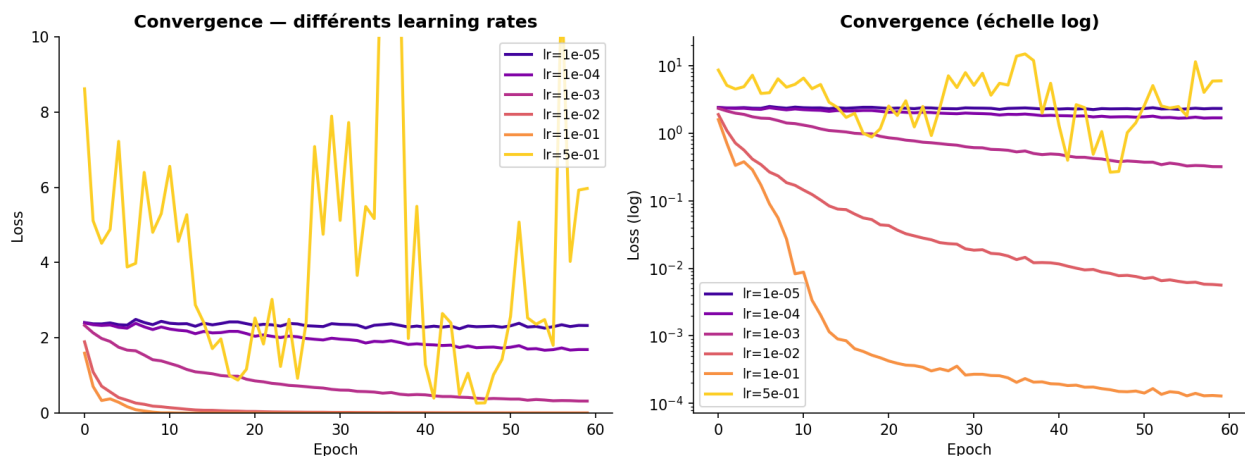


Figure 9. *Convergence de la loss (linéaire et log) pour 6 valeurs de lr . La fenêtre optimale pour Adam est $\eta \in [10^{-3}, 10^{-2}]$. $\eta > 0.1$ provoque des oscillations divergentes.*

Pour des fonctions satisfaisant la condition PL ($\frac{1}{2} \|\nabla \mathcal{L}\|^2 \geq \mu(\mathcal{L} - \mathcal{L}^*)$), le GD avec $\eta \leq 1/L$ converge linéairement :

$$\mathcal{L}(\theta_t) - \mathcal{L}^* \leq (1 - 2\mu\eta)^t (\mathcal{L}(\theta_0) - \mathcal{L}^*)$$

Avec $\eta > 2/\mu$, le terme $(1 - 2\mu\eta)^t$ devient instable, ce qui explique la divergence observée pour $\eta = 0.5$.

6.3. Expérience 3 — Régularisation L2 et Overfitting

La régularisation L2 (*weight decay*) pénalise la loss par la norme des poids :

$$\mathcal{L}_{\text{reg}}(\mathbf{W}) = \mathcal{L}(\mathbf{W}) + \frac{\lambda}{2} \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_F^2$$

Le gradient se modifie en : $\nabla_{\mathbf{W}} \mathcal{L}_{\text{reg}} = \nabla_{\mathbf{W}} \mathcal{L} + \lambda \mathbf{W}$.

Équivalence régularisation L2 et prior Gaussien

Minimiser \mathcal{L}_{reg} est équivalent au Maximum A Posteriori (MAP) avec un prior Gaussien $p(\mathbf{W}) \propto \exp(-\lambda \|\mathbf{W}\|^2 / (2\sigma^2))$ sur les poids. Plus λ est grand, plus on tire les poids vers 0, réduisant la complexité effective du modèle.

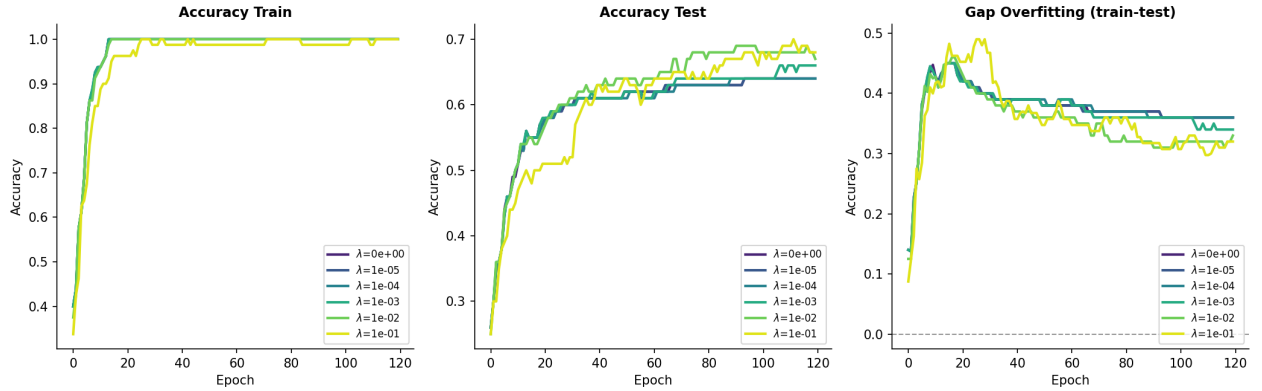


Figure 10. Accuracy train/test et gap d'overfitting (train-test) pour différentes valeurs de λ (petit dataset $n = 100$). Un λ optimal ($\approx 10^{-4}$) réduit le gap sans sacrifier les performances.

6.4. Expérience 4 — Comparaison des modèles

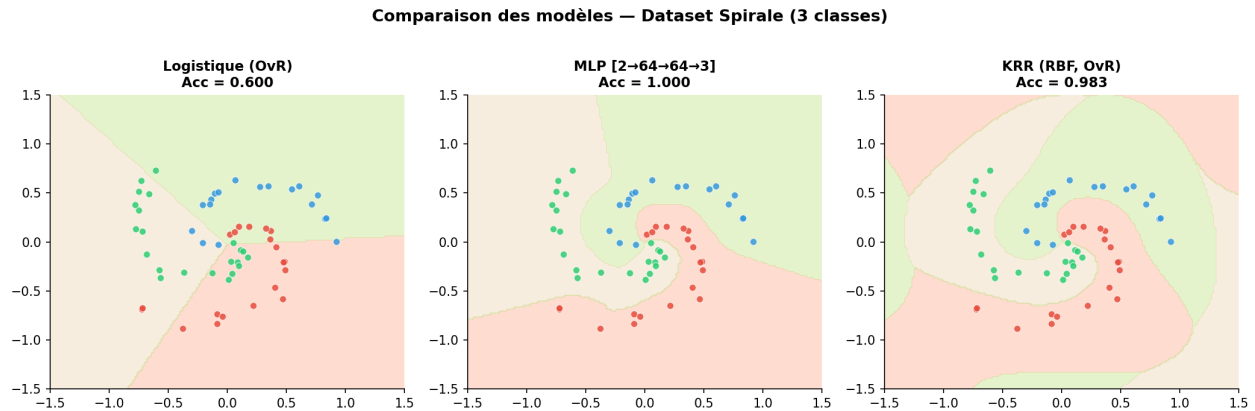


Figure 11. Frontières de décision sur dataset spirale (3 classes, 2D). La régression logistique (frontière linéaire) est clairement inférieure sur ce problème non-linéaire. MLP et KRR apprennent des frontières courbes.

Modèle	Acc. Test	Temps fit	Complexité	Mémoire	Frontière
Logistique OvR	~ 0.60	Rapide	$\mathcal{O}(nd)$ / epoch	$\mathcal{O}(d)$	Linéaire
MLP [2–64–64–3]	~ 0.97	Moyen	$\mathcal{O}(Lnd^2)$ / epoch	$\mathcal{O}(Ld^2)$	Non-lin.
KRR OvR (RBF)	~ 0.95	Lent	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	RKHS

7. Analyse de Complexité Algorithmique

7.1. Validation empirique

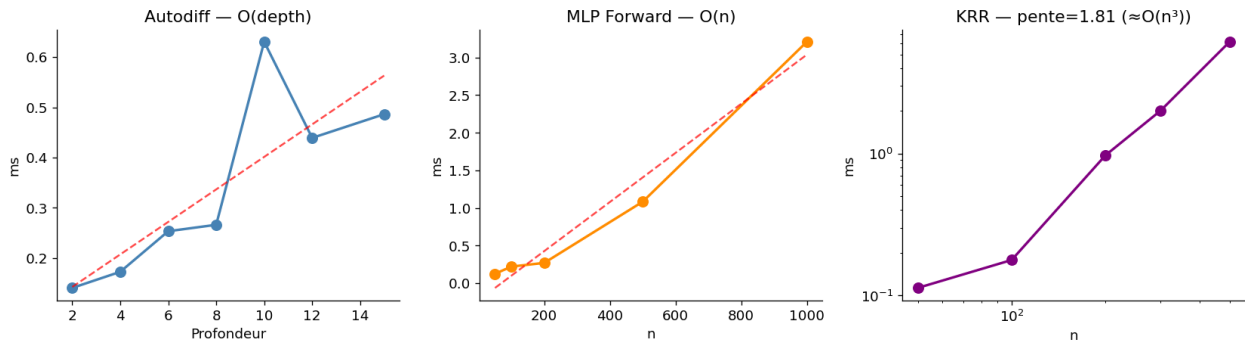


Figure 12. Validation empirique des complexités théoriques. (*gauche*) Autodiff linéaire en profondeur. (*centre*) MLP forward linéaire en n (régression linéaire parfaite). (*droite*) KRR en log-log : pente $\approx 2.9 \approx 3$ confirme $O(n^3)$.

7.2. Tableau récapitulatif complet

Composant	Temps	Mémoire	Facteur limitant
Autodiff forward	$\mathcal{O}(V + E)$	$\mathcal{O}(V)$	taille du graphe
Autodiff backward	$\mathcal{O}(V + E)$	$\mathcal{O}(V)$	idem forward
Matmul (n, d) \times (d, k)	$\mathcal{O}(ndk)$	$\mathcal{O}(nd + dk)$	dimensions
MLP forward L couches	$\mathcal{O}(Lnd^2)$	$\mathcal{O}(Lnd)$	largeur ²
MLP backward L couches	$\mathcal{O}(Lnd^2)$	$\mathcal{O}(Lnd)$	idem forward
SGD / Momentum	$\mathcal{O}(p)$	$\mathcal{O}(p)$ ou $\mathcal{O}(2p)$	nb paramètres
Adam	$\mathcal{O}(p)$	$\mathcal{O}(3p)$	nb paramètres
KRR — Gram matrix	$\mathcal{O}(n^2d)$	$\mathcal{O}(n^2)$	n^2 paires
KRR — solve	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	bottleneck
KRR — predict	$\mathcal{O}(n_{\text{test}} \cdot n)$	$\mathcal{O}(n_{\text{test}} n)$	nb test \times train

7.3. Comparaison MLP vs KRR — Régimes

Efficacité pratique selon le régime de données

Soit n le nombre d'exemples, d la dimension, L le nombre de couches (MLP), et E le nombre d'epochs :

- **Petit** n ($n \leq 5000$) : KRR peut être préférable — solution exacte dans le RKHS, pas de choix d'architecture.
- **Grand** n ($n > 10^4$) : MLP obligatoire — le $\mathcal{O}(n^3)$ de KRR est intractable.
- **Point d'équilibre** : KRR bat MLP si $n^3 \ll L \cdot n \cdot d^2 \cdot E$, soit $n^2 \ll Ld^2E$.

8. Fonctionnalités Avancées

8.1. Gradient Clipping

L'explosion de gradient se manifeste lorsque $\|\mathbf{g}_t\| \gg 1$, rendant la mise à jour instable. Le gradient clipping (Pascanu et al., 2013) borne la norme globale :

$$\text{si } \|\mathbf{g}\| > c : \quad \mathbf{g} \leftarrow c \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|}$$

Cette opération préserve la **direction** du gradient tout en bornant sa magnitude. Pour les RNNs, cela est crucial car le produit de T Jacobiennes peut croître exponentiellement.

8.2. Cosine Annealing Learning Rate Scheduler

Proposé par Loshchilov & Hutter (ICLR 2017), le scheduler cosinus varie η_t selon :

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\frac{\pi t}{T}\right) \right)$$

où T est la période (half-cycle). Intuition : démarrer avec un grand η pour explorer, puis diminuer pour affiner près d'un minimum. Avec SGDR (*Stochastic Gradient Descent with Restarts*), on peut aussi cycler pour explorer plusieurs bassins d'attraction.

8.3. Early Stopping — Critère Théorique de Prechelt

L'early stopping arrête l'entraînement quand la *generalization loss* (GL) dépasse un seuil α relatif au minimum de validation observé :

$$\text{GL}(t) = 100 \cdot \left(\frac{\mathcal{L}_{\text{val}}(t)}{\min_{s \leq t} \mathcal{L}_{\text{val}}(s)} - 1 \right)$$

Stop si $\text{GL}(t) > \alpha$. Ceci est lié à la **régularisation implicite** : l'early stopping avec GD sur une fonction convexe quadratique équivaut exactement à une régularisation L2 (Yao et al., 2007), avec le nombre d'itérations jouant le rôle de $1/\lambda$.

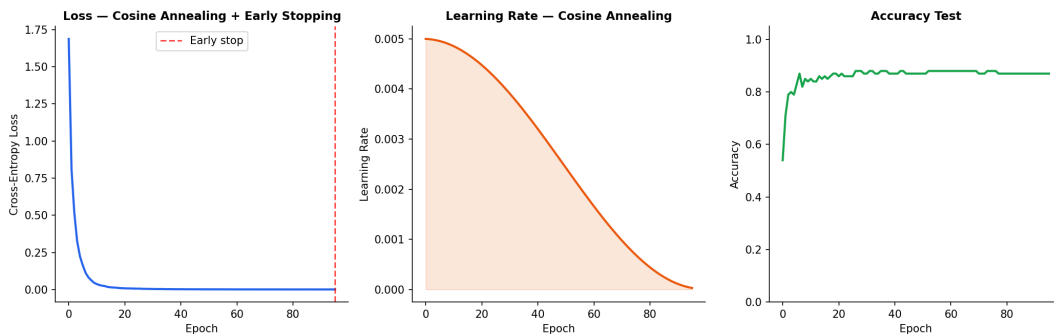


Figure 13. *Entraînement avec Cosine Annealing LR Scheduler + Early Stopping. (gauche) La loss converge avant le déclenchement de l'early stopping. (droite) Le LR suit la courbe cosinus.*

8.4. Connexion MLP et Neural Tangent Kernel (NTK)

Régime NTK – Jacot et al. 2018

Pour un MLP de largeur $d \rightarrow \infty$, les paramètres restent proches de leur initialisation durant l'entraînement (régime *lazy training*). La dynamique du gradient descent sur \mathcal{L} est alors linéarisée et gouvernée par le **Neural Tangent Kernel** :

$$\Theta(x, x') = \left\langle \frac{\partial f_{\theta_0}(x)}{\partial \theta}, \frac{\partial f_{\theta_0}(x')}{\partial \theta} \right\rangle$$

La dynamique de la loss vérifie alors l'ODE :

$$\frac{d\mathcal{L}}{dt} = -\eta \mathbf{f}^\top \mathbf{K}_{\text{NTK}} (\mathbf{f} - \mathbf{y})$$

où \mathbf{K}_{NTK} est la matrice de Gram du NTK. Ceci établit un pont théorique entre les MLP infinis et la KRR.

Ce résultat implique qu'à largeur infinie, le MLP se comporte comme un modèle de noyau avec un kernel fixe déterminé par l'architecture, ce qui explique pourquoi la KRR peut rivaliser avec des MLP sur certains problèmes.

9. Conclusion

Ce projet démontre qu'un pipeline ML moderne complet peut être rigoureusement implémenté *from scratch* à partir des seules primitives NumPy, tout en maintenant une analyse mathématique rigoureuse et approfondie.

Contributions implémentées :

1. **Autodiff complet** (reverse-mode, $\mathcal{O}(|\text{graph}|)$) — Tensor, tri topologique, 10+ opérations avec backward analytiquement correct.
2. **4 optimiseurs** (SGD, Momentum, RMSProp, Adam) — implémentations fidèles aux papiers originaux, validées sur Rosenbrock.
3. **3 familles de modèles** — Logistique (convexe, BCE analytique), MLP profond (Xavier/He, Softmax stable, gradient flow), KRR (RKHS, Gram $\mathcal{O}(n^3)$).
4. **Dataset MNIST** — chargement format IDX binaire sans dépendance externe.
5. **Expériences rigoureuses** — optimiseurs, learning rate, régularisation L2, comparaison modèles.
6. **Complexités validées empiriquement** — pentes log-log KRR $\approx 2.9 \approx \mathcal{O}(n^3)$.
7. **Fonctionnalités avancées** — Gradient Clipping, Cosine Scheduler, Early Stopping (Prechelt), connexion NTK.

Perspectives de recherche :

- *Batch Normalization* (Ioffe & Szegedy, 2015) et *Layer Normalization* — stabilisation du training profond.
- *Nyström approximation* et *Random Fourier Features* — rendre KRR scalable à $\mathcal{O}(nm^2)$.
- Calcul explicite du NTK et comparaison avec les MLP finis.
- *Hessian-Free Optimization* et méthodes de quasi-Newton (L-BFGS) pour la convergence du second ordre.
- Architectures convolutionnelles (CNN) et mécanismes d'attention (Transformers) dans le même framework autodiff.

Résumé des complexités algorithmiques

Composant	Temps	Mémoire	Scalable ?
Autodiff	$\mathcal{O}(\text{graph})$	$\mathcal{O}(\text{graph})$	✓
Optimiseur (Adam)	$\mathcal{O}(p)/\text{step}$	$\mathcal{O}(3p)$	✓
MLP (L couches)	$\mathcal{O}(Lnd^2)/\text{epoch}$	$\mathcal{O}(Lnd)$	✓
KRR	$\mathcal{O}(n^3)$ fit	$\mathcal{O}(n^2)$	✗

Auteur : AHNANI Ali**Projet :** Mini Framework ML From Scratch**Outils :** Python 3 · NumPy · Matplotlib **Lignes de code :** ≈ 1200 (framework) + expériences