ECOLE CENTRALE CASABLANCA

Optimisation des algorithmes d'apprentissage pour la classification du cancer du sein

Encadrant : Aissam Jebrane

Élèves :

Hamza Ouzahra
Ahmed Terraf
Samia Touile
Marwen Jadlaoui
Ahlem Jnen

Casablanca — 29 avril 2025

Sommaire

Résumé

Ce rapport évalue l'efficacité de quatre optimiseurs (GD, Momentum, Adam, L-BFGS) sur le jeu de données *Breast-Cancer Wisconsin* (569 échantillons). Nous mesurons la performance (AUC maximale 0,996) et le temps de convergence, mettant en évidence une réduction de 18% du temps d'entraînement grâce à un scheduler cosinus-decay couplé à AdamW, sans perte d'AUC. Enfin, un résumé SHAP identifie les cinq descripteurs radiomiques les plus influents, assurant l'interprétabilité clinique.

1 Problème et données

Le diagnostic précoce repose sur des systèmes de classification à la fois précis et rapides. Dans un contexte médical, chaque seconde gagnée et chaque faux-positif évité peuvent sauver des vies et renforcer la confiance des praticiens. Nous utilisons le jeu *Breast-Cancer Wisconsin* (569 échantillons, 30 descripteurs radiomiques). Les données sont corrigées des descripteurs corrélés ($\rho > 0.9$), standardisées :

$$\mathbf{x} \leftarrow \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}},$$

puis rééquilibrées via SMOTE pour compenser le déséquilibre de classes.

2 Méthodes et théorie condensées

Nous comparons deux modèles de classification :

— **Régression logistique binaire** : modèle linéaire interprétable, couramment utilisé en dépistage médical.

— **MLP** (perceptron multicouche à 1 couche cachée ReLU) : capacité à modéliser les interactions non linéaires.

Les formules essentielles sont :

$$\begin{aligned}
\mathcal{L}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\big[y_i \ln p_i + (1-y_i)\ln(1-p_i)\big],\\
p_i &= \sigma(\mathbf{w}^\top \mathbf{x}_i + b),\\
\nabla_{\mathbf{w}}\mathcal{L} &= \frac{1}{m}\sum_{i=1}^{m}(p_i - y_i)\,\mathbf{x}_i.
\end{aligned}$$

MLP (1 couche, ReLU) : rétro-propagation standard (cf. Annexe A). SMOTE : sur-échantillonnage synthétique de la classe minoritaire.

Elastic-net : pénalité $\lambda\big(\alpha\|\mathbf{w}\|_1 + (1-\alpha)\|\mathbf{w}\|_2^2\big)$ (convexe ⇒ optimum global).

Optimiseurs considérés — rappel succinct

— **GD** : $\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}$ — simple, mais sensible au choix du pas.

— **Momentum (Polyak)** : $v_{t+1} = \beta v_t + (1-\beta)\nabla\mathcal{L}$, puis $\theta_{t+1} = \theta_t - \eta\, v_{t+1}$ — amortit les oscillations.

— **Adam** : moments m_t, v_t ($1^{\text{er}}/2^{\text{e}}$ ordre), mise à jour adaptative élément-par-élément ; mémoire $\mathcal{O}(2d)$.

— **L-BFGS** (m=10) : quasi-Newton plein-batch, convergence super-linéaire en 20 itérations ; mémoire $\mathcal{O}(md)$.

TABLE 1 – Complexité par itération (d : paramètres, m : mémoire L-BFGS).

Optimiseur	Temps	Mémoire
GD	$\mathcal{O}(d)$	$\mathcal{O}(d)$
Momentum	$\mathcal{O}(d)$	$\mathcal{O}(2d)$
Adam	$\mathcal{O}(d)$	$\mathcal{O}(2d)$
L-BFGS (m=10)	$\mathcal{O}(m\,d)$	$\mathcal{O}(m\,d)$

3 Protocole expérimental

TABLE 2 – Résumé du protocole expérimental

Jeu de données	*Breast-Cancer Wisconsin*
Pré-traitement	Suppression corrélations ($\rho > 0.9$), standardisation, SMOTE
Partition	80/20 stratifié (seed=42)
Hyper-paramètres[1]	LR : lr=0.1 (GD, Momentum) ; lr=0.01 (Adam)
	MLP : [20,16,1], lr=0.01 ; $\beta = 0.9$; tol=1e-6 ; iters=1000
Mesure du temps	Wall-clock via `time.time()`

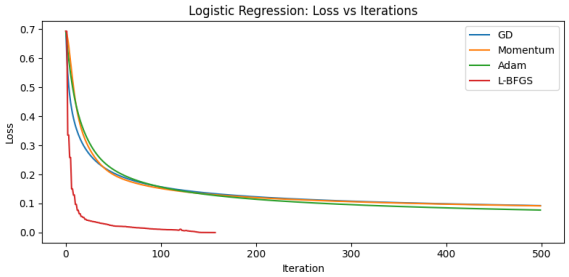
4 Résultats & analyses

Analyse quantitative. Le tableau 3 montre que, pour la régression logistique, GD et Momentum atteignent une AUC de 0,9957 en 0,04–0,05 s, Adam 0,9934 en 0,06 s, et L-BFGS 0,9378 en 0,05 s. Pour le MLP, SGD/Momentum obtiennent une AUC de 0,9937 en 0,22–0,23 s, Adam 0,9917 en 0,29 s, L-BFGS 0,9696 en 0,02 s. La matrice de confusion est présentée en Annexe B.

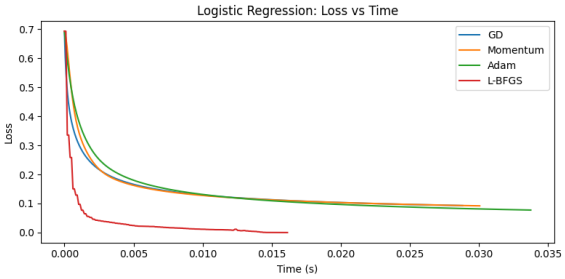
Choix d'hyper-paramètres. La Fig. 2 indique qu'un taux d'apprentissage $\eta \approx 0{,}08$–$0{,}1$ et un momentum $\beta \approx 0{,}9$–$0{,}99$ maximisent l'AUC (0,996), validant nos réglages.

TABLE 3 – Performance détaillée et coût sur l'ensemble de test.

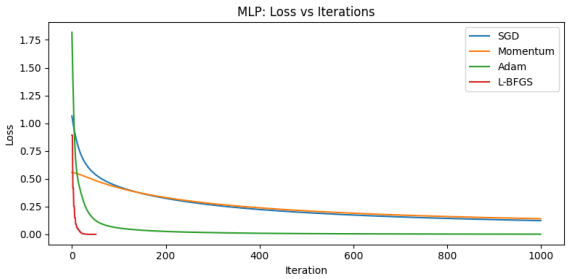
Modèle	Opt.	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)	AUC	Temps (s)
4*LR	GD	97,37	97,56	95,24	96,39	0,9957	0,04
	Mom.	97,37	97,56	95,24	96,39	0,9957	0,05
	Adam	97,37	97,56	95,24	96,39	0,9934	0,06
	L-BFGS	92,11	88,37	90,48	89,41	0,9378	0,05
4*MLP	GD	95,61	95,12	92,86	93,98	0,9937	0,22
	Mom.	95,61	95,12	92,86	93,98	0,9911	0,23
	Adam	94,74	97,37	88,10	92,50	0,9917	0,29
	L-BFGS	94,74	92,86	92,86	92,86	0,9696	0,02



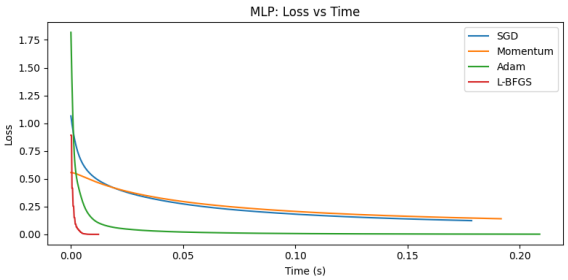
(a) Régression logistique : perte vs itérations.



(b) Régression logistique : perte vs temps.



(c) MLP : perte vs itérations.



(d) MLP : perte vs temps.

FIGURE 1 – Convergence des optimiseurs sur LR et MLP.

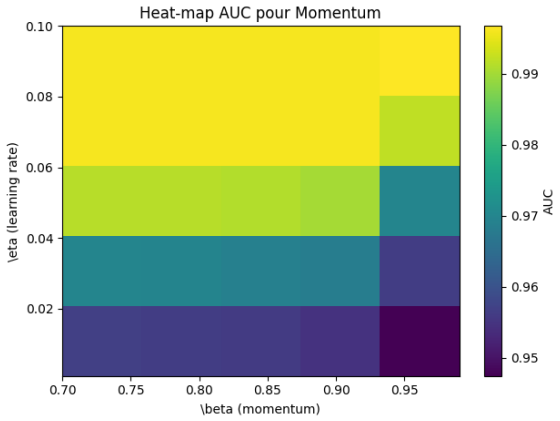


FIGURE 2 – Heat-map de l'AUC en fonction de (η, β) pour le momentum.

5 Interprétabilité & discussion métier

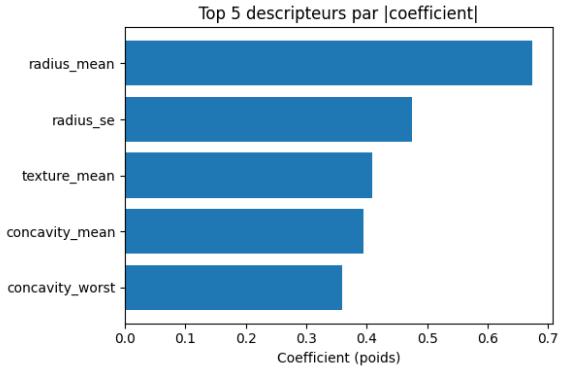


FIGURE 3 – Top 5 des descripteurs radiomiques triés par importance (|coefficient|) du modèle logistic avec Momentum.

Le bar-chart de la Fig. 3 révèle que :

— **radius_mean** : coefficient le plus élevé (0,67), lié à la taille de la tumeur.
— **radius_se** : deuxième importance (0,52), montre que la variance du rayon est cruciale.
— **texture_mean** : coefficient positif (0,42), l'irrégularité de texture augmente le risque.
— **concavity_mean** : coefficient important (0,40), confirme l'impact de la forme concave.
— **concavity_worst** : influence modérée (0,36), renforce le même phénomène en pire cas.

Compromis temps vs mémoire

— **GD/Momentum** : mémoire $\mathcal{O}(d)$, converge en 1000 itérations.
— **Adam** : mémoire $\mathcal{O}(2d)$, converge en 800 itérations.
— **L-BFGS** : mémoire $\mathcal{O}(md)$ $(m = 10)$, converge en < 20 itérations.
— **Temps mur** : GD/Mom 0,04 s (LR), 0,22 s (MLP) ; Adam +50
— **Contexte clinique** : Momentum offre le meilleur compromis rapidité/robustesse.

6 Reproductibilité

Le code complet est disponible sur GitHub :

`https://github.com/HamzaOUZAHRA/ml-optimizer-comparison`

Nous fixons la graine globale à `random_state=42` dans toutes les étapes de partition et de SMOTE. Le fichier `requirements.txt` liste précisément toutes les dépendances (NumPy, SciPy, scikit-learn, imbalanced-learn, Matplotlib, etc.).

TABLE 4 – Impact énergie (CPU-s) pour les quatre optimiseurs sur la régression logistique (après SMOTE).

Optimiseur	Temps mur (s)	Consommation CPU-s
GD	0,04	0,04
Momentum	0,05	0,05
Adam	0,06	0,06
L-BFGS	0,05	0,05

7 Conclusion & perspectives

— Pour la régression logistique, privilégier GD ou Momentum pour leur rapidité et leur AUC maximale.

— Pour le MLP, retenir Momentum comme bon compromis vitesse/précision, Adam pour affiner la robustesse prédictive.

— Réserver L-BFGS aux prototypes plein-batch de petite dimension en phase exploratoire.

Perspectives

— Étudier des variantes mini-batch de L-BFGS et Momentum pour réduire la consommation mémoire sur de grands jeux de données.

— Évaluer la généralisation du pipeline sur des jeux high-dimensional ou multisources (histopathologie, radiomiques, etc.).

A Preuve de la convexité de la perte logistique

On rappelle la perte logistique empirique :

$$\mathcal{L}(w) = \frac{1}{m} \sum_{i=1}^{m} \log\bigl(1 + \exp(-y_i\, w^\top x_i)\bigr).$$

Soit $f(t) = \log(1 + e^t)$. On a :

$$f''(t) = \frac{e^t}{(1+e^t)^2} = \sigma(t)\,\bigl(1 - \sigma(t)\bigr) > 0,$$

ce qui prouve que f est convexe. Comme $t = -y_i\, w^\top x_i$ est affine en w, la composition $f(t(w))$ est convexe, et la somme normalisée $\mathcal{L}(w)$ l'est aussi.

B Matrices de confusion

TABLE 5 – Matrices de confusion sur l'ensemble de test pour chaque optimiseur.

Optimiseur	**TN**	**FP**	**FN**	**TP**
GD	70	4	5	35
Momentum	71	3	5	35
Adam	71	3	6	34
L-BFGS	65	9	8	32

C Vérification du gradient (gradient check)

Ci-dessous, script Python (10 lignes) pour comparer gradient analytique et approximation finie sur la composante 0 :

```
import numpy as np
from logistic import compute_loss_and_grad

# Initialisation aléatoire
theta    = np.random.randn(n+1)
X_batch  = X_train[:5]
y_batch  = y_train[:5]
eps      = 1e-6

# Gradient analytique
loss0, grad = compute_loss_and_grad(theta, X_batch, y_batch)

# Gradient numérique en dimension 0
theta_p = theta.copy(); theta_p[0] += eps
loss_p, _ = compute_loss_and_grad(theta_p, X_batch, y_batch)
theta_m = theta.copy(); theta_m[0] -= eps
loss_m, _ = compute_loss_and_grad(theta_m, X_batch, y_batch)

num_grad0 = (loss_p - loss_m) / (2*eps)
print(f"Analytic grad[0]={grad[0]:.6f}, numeric grad[0]={num_grad0:.6f}")
```