Différenciation automatique randomisée et application au pricing rapide

M2 Probabilités et Finance

Auteur: Fares FEKI
Tuteur de stage: Bouazza SAADEDDINE
Responsable académique: Gilles PAGES

17 avril 2024

Résumé

Dans ce rapport, nous commençons par examiner une nouvelle technique appelée "Différenciation Automatique Randomisée", introduite en 2020 dans l'article [1]. Il s'agit d'une variante randomisée de la différentiation automatique standard qui permet d'obtenir un estimateur de gradient non biaisé en utilisant un échantillonnage au niveau du graphe de calcul. Dans un premier temps, nous présentons les bases théoriques de cette méthode dans un cadre général. Ensuite, nous nous penchons sur son impact en termes de performances et d'utilisation de la mémoire lors de l'entraînement de réseaux de neurones.

Dans la deuxième partie de ce rapport, nous étudions une méthode permettant d'approximer une espérance conditionnelle à l'aide de réseaux de neurones. Nous appliquons ensuite cette méthode dans le contexte d'un modèle de crédit avec une intensité de défaut stochastique, en particulier pour calculer les spreads des contrats CDS et calibrer les paramètres du modèle en utilisant des données de marché.

Remerciements

Je tiens à exprimer ma sincère reconnaissance envers toute l'équipe de Recherche Quantitative de CA-CIB pour leur accueil chaleureux, leur bienveillance et les précieuses interactions que j'ai eues avec eux au cours de mon stage.

Je souhaite particulièrement remercier mon tuteur de stage, Bouazza Saadeddine, pour ses conseils, son assistance constante et sa disponibilité tout au long de mon stage. Ses compétences en finance, en mathématiques et en informatique ont été d'une aide inestimable.

J'adresse également mes remerciements à Pierre Fichter, Responsable de l'équipe Transversal Quants, pour m'avoir fait confiance et m'avoir donné l'opportunité de rejoindre l'équipe et de travailler sur des sujets très intéressants.

Je tiens à exprimer ma profonde gratitude envers Christophe Michel, Responsable de la Recherche Quantitative de la Division Global Market, pour m'avoir permis de faire partie de son équipe, pour ses remarques avisées et son sens de l'humour.

Par ailleurs, je tiens à exprimer ma reconnaissance à Nicolas Damay, Responsable de IA GMD, pour avoir facilité l'accès aux serveurs GPU internes, ce qui m'a permis de mener mes expériences numériques dans des conditions exceptionnelles.

Enfin, je tiens à remercier l'ensemble des professeurs du programme M2 Probabilités et Finance qui m'ont transmis les connaissances théoriques essentielles pour comprendre les enjeux actuels des marchés financiers.

Table des matières

Introduction

Dans le domaine de l'apprentissage automatique et du calcul des sensibilités des produits financiers, la Différenciation Automatique s'est imposée comme un outil puissant, offrant à la fois rapidité et précision dans le calcul des gradients. Cependant, un défi courant lors de l'entraînement de réseaux neuronaux est l'utilisation de la mémoire, qui devient souvent un goulot d'étranglement. Cela soulève une question importante : Est-il judicieux d'allouer d'importantes ressources mémoire pour le calcul des gradients exacts, notamment lors de l'utilisation de méthodes d'optimisation stochastique ? Pour répondre à cette préoccupation, la Différenciation Automatique Randomisée entre en jeu, offrant des estimations de gradients non biaisées tout en réduisant les besoins en mémoire, bien que cela se fasse au détriment d'une variance accrue.

1 Différenciation automatique

La différenciation automatique[2], également connue sous le nom d'AD (Automatic Differentiation), regroupe différentes techniques permettant de transformer un programme qui évalue une fonction différentiable $f : \mathbb{R}^n \to \mathbb{R}^m$ en un autre programme capable de calculer les dérivées associées, notamment la jacobienne : $\mathcal{J}[f] = f' : \mathbb{R}^n \to \mathbb{R}^{m \times n}$

Contrairement aux méthodes traditionnelles de différenciation symbolique ou numérique, l'AD permet de calculer les dérivées avec une précision élevée et sans erreur d'arrondi significative.

L'idée fondamentale de l'AD est de décomposer une fonction en une séquence d'opérations élémentaires plus simples, telles que les additions, les multiplications et les fonctions élémentaires. Ces opérations sont ensuite combinées en un graphe de calcul, où chaque nœud représente une opération et chaque arête représente un flot de dépendance.

L'AD fonctionne en évaluant simultanément la fonction et sa dérivée en utilisant les règles de dérivation standard. En utilisant un graphe computationnel linéarisé (LCG), l'AD peut calculer les dérivées exactes de la fonction donnée en effectuant une seule passage de calcul.

Les composantes de f sont représentés par y_j, les entrées par θ_i, et les variables intermédiaires par z_l. L'AD peut être décrite comme le calcul d'une dérivée partielle qui se fait en sommant tous les chemins à travers le graphe computationnel linéarisé (Bauer, 1974) [3] :

$$\frac{\partial y_j}{\partial \theta_i} = \mathcal{J}_\theta[f]_{j,i} = \sum_{[i \to j]} \prod_{(k,l) \in [i \to j]} \frac{\partial z_l}{\partial z_k} \tag{1}$$

où $[i \to j]$ représente les chemins du sommet i au sommet j et $(k, l) \in [i \to j]$ décrit l'ensemble des arêtes dans ce chemin.

La différenciation automatique se déroule en deux étapes : le **forward pass**, qui évalue la fonction d'origine et stocke les valeurs intermédiaires, et le **backward**

pass, qui calcule les dérivées par rapport aux variables d'entrée en utilisant les valeurs intermédiaires stockées et la règle de dérivation en chaîne.

Remarque : En pratique, afin de simplifier le graphe de calcul, z_l peut être un vecteur de dimension d_l ce qui fait que $\partial z_l / \partial z_k \in \mathbb{R}^{d_l \times d_k}$ représente la jacobienne intermédiaire de l'opération $z_k \to z_l$.

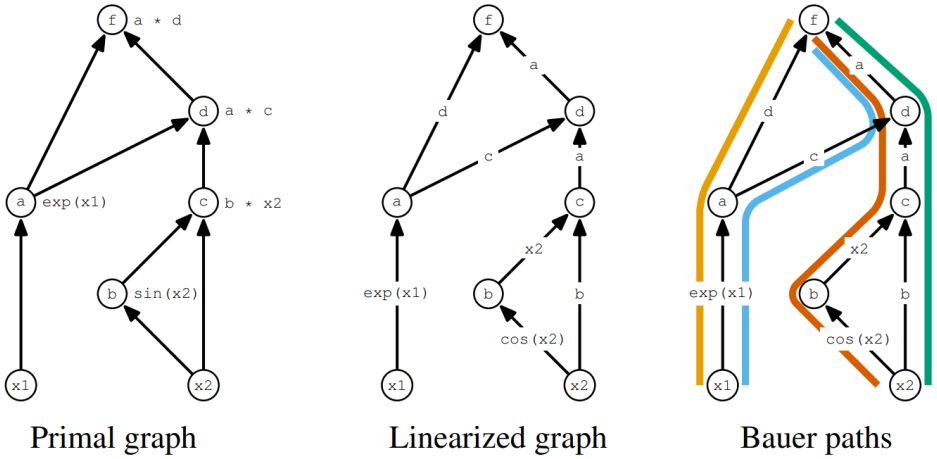


FIGURE 1 – Exemple d'un graphe de calcul

2 Différenciation automatique randomisée

2.1 Présentation générale

La différenciation automatique randomisée [1] consiste à construire un graphe computationnel linéarisé (LCG) **creux** et **aléatoire** lors du forward pass, de sorte que l'application de l'AD à ce nouveau graphe génère une estimation **non biaisée** du vrai gradient. Il est important de noter que le graphe computationnel d'origine est utilisé lors du forward pass, tandis que la randomisation est utilisée pour déterminer un LCG à utiliser lors du backward pass. Cette approche permet de réduire les coûts de mémoire en stockant directement le LCG creux ou en conservant un nombre limité de variables intermédiaires nécessaires au calcul de ce LCG. Une stratégie simple consiste à échantillonner uniformément et aléatoirement des chemins à partir du graphe de calcul, et à former une estimation de Monte Carlo de l'équation (1).
L'échantillonnage à partir d'un LCG composé d'opérations vectorielles peut être réalisé grâce à ce que nous appelons "l'injection de matrice aléatoire". Prenons l'exemple de la figure (2)

$$\frac{\partial y}{\partial \theta} = \frac{\partial y}{\partial C}\frac{\partial C}{\partial B}\frac{\partial B}{\partial A}\frac{\partial A}{\partial \theta}$$

où A, B, C sont des vecteurs dont les composantes sont $a_i, b_i, c_i, \partial C/\partial B, \partial B/\partial A$ sont les matrices Jacobienne pour les opérations intermédiaires de taille 3 ×3, $\partial y/\partial C$ est un vecteur ligne 1×3, et $\partial A/\partial \theta$ est un vecteur colonne 3×1.

On remarque que la contribution du chemin $p = \theta \to a_1 \to b_2 \to c_2 \to y$ au gradient y est donnée par

$$\frac{\partial y}{\partial C} P_2 \frac{\partial C}{\partial B} P_2 \frac{\partial B}{\partial A} P_1 \frac{\partial A}{\partial \theta}$$

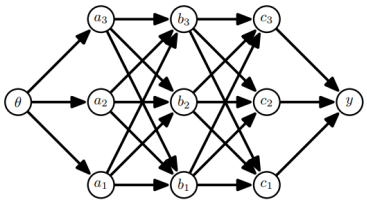
avec $P_i = e_i e_i^T$ (où les e_i représentent la base canonique de $\mathbb{R}^3$). En général, pour échantillonner des chemins d'une étape $B \to C$ dans un LCG vectorisé, où $B \in \mathbb{R}^d, C \in \mathbb{R}^m$, on utilise la matrice aléatoire $P_B = \frac{d}{k} \sum_{s=1}^{k} R_s$. Les R_s sont i.i.d $\sim \mathcal{U}(P_1, P_2, \ldots, P_d)$. On a $\mathbb{E}[R_1] = \sum_{i=1}^{d} \frac{1}{d} P_i = \frac{1}{d} I_d$ et $\mathbb{E}[P_B] = \frac{d}{k} \sum_{s=1}^{k} \mathbb{E}[R_1] = \frac{d}{k}.k.\frac{1}{d} I_d = I_d$.

Nous considèrons trois matrices P_A, P_B et P_C suivant la même loi que $\frac{d}{k} \sum_{s=1}^{k} R_s$ et l'estimateur de gradient suivant :

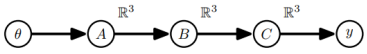
$$\widehat{\frac{\partial y}{\partial \theta}} = \frac{\partial y}{\partial C} P_C \frac{\partial C}{\partial B} P_B \frac{\partial B}{\partial A} P_A \frac{\partial A}{\partial \theta}$$

$\mathbb{E}\left[\widehat{\frac{\partial y}{\partial \theta}}\right] = \frac{\partial y}{\partial C} \mathbb{E}[P_C] \frac{\partial C}{\partial B} \mathbb{E}[P_B] \frac{\partial B}{\partial A} \mathbb{E}[P_A] \frac{\partial A}{\partial \theta} = \frac{\partial y}{\partial \theta}$

Par conséquent, $\widehat{\frac{\partial y}{\partial \theta}}$ est un estimateur **non biaisé** du gradient.



(b) Fully Interleaved Graph



(c) Vector graph for (b).

FIGURE 2 – Graphe de calcul vectorisé

2.2 Pourquoi un estimateur non-biasé

2.2.1 Algorithme de Robbins-Monro

L'algorithme Robbins-Monro[4], introduit en 1951 par Herbert Robbins et Sutton Monro, présente une méthodologie pour résoudre un problème de recherche de racine d'une fonction, où la fonction est représentée sous forme d'espérance : $h(\theta) = \mathbb{E}[H(\theta, Z)]$, avec $Z \sim \mu$. L'algorithme s'écrit alors :

$$\theta_{n+1} = \theta_n - \gamma_{n+1} H(\theta_n, Z_{n+1}), \quad n \geq 0 \tag{2}$$

, avec Z_n des varibales i.i.d $\sim \mu$ et $(\gamma_n)_{n\in\mathbb{N}^*}$ une suite de réels positifs.

Théorème (Algorithme de Robbins-Monro)[5] Supposons que la fonction moyenne h soit continue et que :

1. $\forall\theta \in \mathbb{R}^d, \theta \neq \theta_*, \quad (\theta - \theta_* \mid h(\theta)) > 0.$
2. $\theta_0 \in L^2$ et que $\forall\theta \in \mathbb{R}^d, \quad \|H(\theta, Z)\|_2 \leq C(1 + |\theta|)$
3. La suite des pas $(\gamma_n)_{n\geq 1}$ satisfait la condition Decreasing Step (DS) :

$$\sum_{n\geq 1} \gamma_n = +\infty \quad and \quad \sum_{n\geq 1} \gamma_n^2 < +\infty$$

Alors

$$\{h = 0\} = \{\theta_*\} \qquad \text{et} \qquad \theta_n \xrightarrow{\text{p.s.}} \theta_*.$$

La convergence est également valable dans tout $L^p, p \in (0, 2)$ (et $(|\theta_n - \theta_*|)_{n\geq 0}$ est bornée dans L^2).

2.2.2 Descente de gradient stochastique (SGD)

On se place dans le cas où l'on souhaite trouver une racine du gradient d'une certaine fonction exprimée sous forme d'une espérance : $h(\theta) = \nabla L(\theta) = \mathbb{E}[H(\theta, Z)]$, avec $L : \mathbb{R}^d \to \mathbb{R}_+$ une fonction différentiable . Dans ce cas, $H(\theta, Z)$ est **un estimateur non biasé** de ∇L.

Théorème (Descente de gradient stochastique)[5] Supposons que :

1. $\lim_{|\theta|\to+\infty} L(\theta) = +\infty$, ∇L est Lipschitzienne, $|\nabla L|^2 \leq C(1 + L)$ et $\{\nabla L = 0\} = \{\theta_*\}$.
2. $\|H(\theta, Z)\|_2 \leq C\sqrt{1 + L(\theta)}$ et que $L(\theta_0) \in L^1(\mathbb{P})$.
3. $(\gamma_n)_{n\geq 1}$ satisfasse (DS).

Alors

$$L(\theta_*) = \min_{\mathbb{R}^d} L \qquad \text{et} \qquad \theta_n \xrightarrow{\text{p.s.}} \theta_* \qquad \text{lorsque} \qquad n \to +\infty.$$

De plus, $\nabla L(\theta_n)$ converge vers 0 dans tous les $L^p, p \in (0, 2)$ (et $(L(\theta_n))_{n\geq 0}$ est bornée dans L^1 de sorte que $(\nabla L(\theta_n))_{n\geq 0}$ est bornée dans L^2).

Dans le cadre d'apprentissage automatique supervisé, on dispose d'un jeu de données étiquetées $(z_k)_{k=1:N}$, avec $z_k = (x_k, y_k)$. On cherche à minimiser une fonction de coût empirique en fonction des paramètres du modèle θ. La fonction de coût globale s'écrit comme une moyenne de fonctions de coût locales θ : $L(\theta) = \frac{1}{N}\sum_{k=1}^{N} \ell(f_\theta(x_k), y_k)$. Le gradient associé est

$$\nabla L(\theta) = \frac{1}{N}\sum_{k=1}^{N} \nabla_\theta \ell(f_\theta(x_k), y_k) = \mathbb{E}^{\mu_N}[\nabla_\theta l(f_\theta(X), Y)] \text{ avec } \mu_N = \frac{1}{N}\sum_{k=1}^{N} \delta_{(x_k,y_k)}$$

Ainsi, à chaque itération de l'algorithme de SGD, on tire d'une manière indépendante un indice $i_n \sim \mathcal{U}(\{1, \ldots, N\})$ et on met à jour les paramètres suivant la règle :

$$\theta_{n+1} = \theta_n - \gamma_{n+1} \nabla_\theta \ell \left(f_{\theta_n}(x_{i_{n+1}}), y_{i_{n+1}} \right), \quad n \geq 0 \tag{3}$$

Dans l'équation (5), on suppose qu'on peut calculer de façon assez précise le gradient de la loss locale ∇_θ, ce qui est vrai dans le cas d'un réseau de neurones grâce à la technique d'autodifférentiation. Ainsi, la seule source de bruit est le tirage des échantillons. Cependant, la nouveauté de la méthode de différenciation automatique randomisée est de remplacer ce calcul exact par un estimateur du gradient grâce à un échantillonnage effectué au niveau du graphe de calcul de l'AD et d'une manière indépendante du tirage des échantillons, tout en conservant le caractère non biaisé afin de rester dans le cadre de l'algorithme de Robbins-Monro. Le nouveau algorithme de SGD s'écrit :

$$\theta_{n+1} = \theta_n - \gamma_{n+1} \widehat{\nabla_\theta \ell}^{\text{rad}} \left(f_{\theta_n}(x_{i_{n+1}}), y_{i_{n+1}} \right), \quad n \geq 0 \tag{4}$$

3 Réseau de neurones

3.1 Définition

Soient L, $n^{[0]}$, $n^{[1]}$, ..., $n^{[L]} \in \mathbb{N}$. Soit $\sigma : \mathbb{R} \to \mathbb{R}$ et pour tout $\ell = 1, \ldots, L$, $A_\ell : \mathbb{R}^{n^{[l-1]}} \to \mathbb{R}^{n^{[l]}}$ une fonction affine. Une fonction $F : \mathbb{R}^{n^{[0]}} \to \mathbb{R}^{n^{[L]}}$ définie comme suit :

$$F(x) = A_L \circ F_{L-1} \circ \cdots \circ F_1 \text{ avec } F_\ell = \sigma \circ A_\ell \text{ pour } \ell = 1, \ldots, L-1$$

est appelée un réseau de neurones [6](feed forward). Ici, la fonction d'activation σ est appliquée composante par composante. L désigne le nombre de couches, $n^{[l]}, \ldots, n^{[L-1]}$ représentent les dimensions des couches cachées et $n^{[0]}, n^{[L]}$ représentent respectivement les dimensions des couches d'entrée et de sortie. Pour tout $\ell = 1, \ldots, L$, la fonction affine A_ℓ est donnée par $A_\ell(x) = W^\ell x + b^\ell$ pour certains $W^\ell \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ et $b^\ell \in \mathbb{R}^{n^{[l]}}$. Pour tout $i = 1, \ldots, n^{[l]}$, $j = 1, \ldots, n^{[l-1]}$, le réel W^ℓ_{ij} est interprété comme le poids de l'arête connectant le nœud i de la couche $\ell - 1$ au nœud j de la couche ℓ.

On note $\mathcal{NN}^\sigma_{d_0,d_1}$ l'ensemble des réseaux neuronaux qui vont de $\mathbb{R}^{d_0}$ à $\mathbb{R}^{d_1}$ avec une fonction d'activation σ.

3.2 Fonctions de perte

Dans les réseaux de neurones, il existe deux types couramment utilisés de fonctions de perte : l'Erreur Quadratique Moyenne (MSE) et la Perte de l'Entropie Croisée. Ces fonctions sont utilisées dans des contextes de régression et de classification, respectivement.

1. L'Erreur Quadratique Moyenne (MSE) est utilisée dans les problèmes de régression. Elle mesure la moyenne des carrés des différences entre les valeurs prédites par le réseau neuronal et les valeurs réelles de la variable cible. Cette fonction de perte est définie par la formule :

$$\text{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

où y représente les vraies valeurs de la variable cible, $\hat{y}$ représente les valeurs prédites par le réseau neuronal, et n est le nombre total d'échantillons.

2. La Perte de l'Entropie Croisée est utilisée dans les problèmes de classification. Elle mesure la dissimilitude entre les probabilités prédites par le réseau et les probabilités réelles des classes. Cette fonction de perte est définie par la formule :

$$\text{CrossEntropy}(y, \hat{y}) = -\sum_{i=1}^{n} y_i^T \log(\hat{y}_i)$$

où y représente les vraies classes (encodées en "one-hot"), $\hat{y}$ représente les probabilités prédites des classes.

Dans le cas de la classification, il est courant d'appliquer une fonction d'activation softmax aux sorties du réseau de neurones avant d'utiliser la perte de l'entropie croisée. Cela permet de convertir les sorties en probabilités normalisées pour chaque classe.

La fonction softmax est définie comme suit :

$$\text{softmax}(z) = \left(\frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}}\right)_{i=1,\ldots,c}$$

où z_i est la valeur de la i-ème sortie du réseau neuronal et c est le nombre total de classes.

En ce qui concerne la régression, il n'est généralement pas nécessaire d'appliquer une activation spécifique aux sorties du réseau.

3.3 Analyse rapide de la complexité spatiale

Dans cette section, nous examinerons le cas des réseaux neuronaux entièrement connectés avec des activations ReLU. L'objectif est d'identifier les étapes gourmandes en mémoire lors de la rétropropagation afin de dériver une bonne stratégie d'échantillonnage de chemins. À titre d'illustration, nous considérons un exemple simple de la figure (3), où nous avons un réseau à deux couches et utilisons la descente de gradient stochastique. Le gradient de la perte associée à l'échantillon x_1 par rapport à W_1 est donné par :

$$\frac{\partial y}{\partial h_{2,1}}\frac{\partial h_{2,1}}{\partial a_{1,1}}\frac{\partial a_{1,1}}{\partial h_{1,1}}\frac{\partial h_{1,1}}{\partial W_1}$$

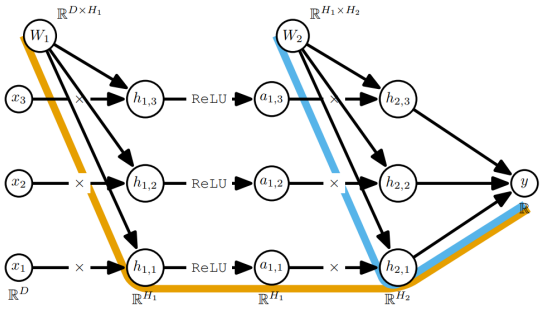


FIGURE 3 – Graphe de calcul d'un réseau de neurones avec deux couches cachées.

Analysons séparément chaque terme dans la formule précédente :

— $\frac{\partial y}{\partial h_{2,1}}$: Dépend de la dimension de sortie du problème de classification ou de régression, qui est généralement de l'ordre de 10, cette variable n'est donc pas un goulot d'étranglement en matière de mémoire.

— $\frac{\partial h_{2,1}}{\partial a_{1,1}}$: est la jacobienne de la deuxième couche cachée par rapport à l'activation de la première couche. Elle peut être construite à partir de W_2, qui est déjà stockée en mémoire.

— $\frac{\partial a_{1,1}}{\partial h_{1,1}}$: représente la jacobienne de la fonction d'activation. Dans notre cas d'une activation ReLU, il s'agit d'une matrice binaire et peut donc être stockée avec un bit par entrée.

— $\frac{\partial h_{1,1}}{\partial W_1}$: La jacobienne de la sortie de la couche cachée par rapport à W_1, qui, dans un perceptron multicouche, est équivalent aux entrées de la couche en question (c'est-à-dire les activations de la couche précédente). Dans le cas de la figure (3), cela correspond à x_1. Dans le cas général, avec m échantillons dans le mini-batch de dimension $n^{[l-1]}$ chacun, la taille de cette matrice est de $m \cdot n^{[l-1]}$. Par conséquent, le stockage de cette variable constitue un goulot d'étranglement en termes de mémoire si on augmente les tailles ou le nombre d'échantillons par mini-batch ou bien le nombre de couches cachées.

Remarque : Dans l'analyse précédente, nous avons uniquement pris en compte la contribution d'un échantillon au calcul des gradients de la fonction coût du réseau. Cela correspond à sélectionner le chemin jaune sur la figure (3). Par conséquent, la descente de gradient stochastique (SGD) ou la Mini-Batch SGD peuvent être considérées comme des cas particuliers de la différenciation automatique randomisée, où nous isolons la contribution d'un ou de plusieurs échantillons à chaque itération.

3.4 L'algorithme de rétropropagation

Pour les détails de calcul des gradients utilisés dans cette section, nous renvoyons le lecteur à [7].

3.4.1 Notations

Nous introduisons ici quelques notations qui seront utilisées dans cette section.

— L est le nombre de couche cachée dans le réseaux, $n^{[l]}$ est le nombre de neurones de la couche l et m la taille du mini-batch.

— $z^{[l]} \in \mathbb{R}^{m \times n^{[l]}}$ répresente la sortie de la couche l avant activation et $a^{[l]} \in \mathbb{R}^{m \times n^{[l]}}$ celle après activation.

— $W^{[l]} \in \mathbb{R}^{n^{[l-1]} \times n^{[l]}}$ et $b^{[l]} \in \mathbb{R}^{n^{[l]}}$. représentent respectivement les poids et les biais de la couche l

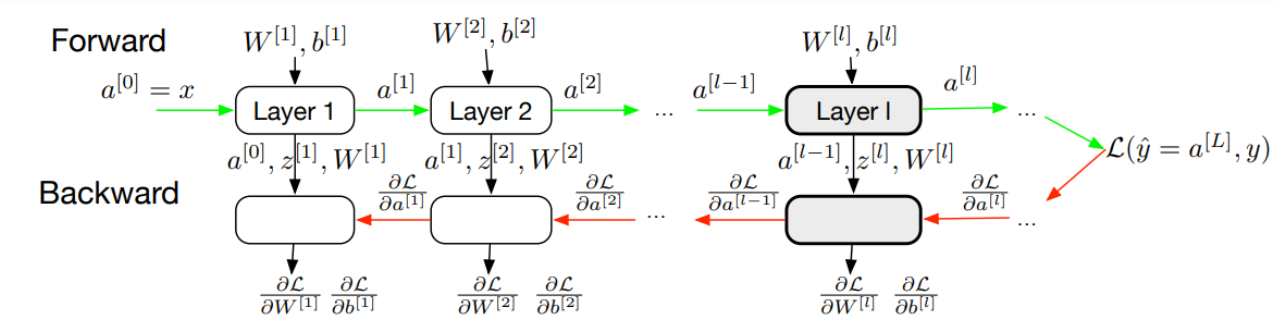


FIGURE 4 – Algorithme de rétropropagation pour un réseau feed forward.

3.4.2 Forward

Durant le pass forward, les activations des neurones sont calculées couche par couche en se propageant de l'entrée vers la sortie du réseau. Cela peut être résumé par les équations suivantes :
Initialisation : $a^{[0]} = x$, où x est la matrice de données de taille $m \times n^{[0]}$.
Entrée : $a^{[l-1]}$

— $z^{[l]} = a^{[l-1]} W^{[l]} + b^{[l]}$

— $a^{[l]} = g^{[l]}(z^{[l]})$

Sortie : $a^{[l]}$
Variables sauvegardées pour le pass backward : $a^{[l-1]}$, $z^{[l]}$ et $W^{[l]}$

3.4.3 Backward

Pendant cette étape, chaque couche prend en entrée le gradient de la fonction perte par rapport à son activation, ainsi que certaines variables sauvegardées lors de la pass forward, calcule le gradient par rapport à ses poids, puis produit le gradient par rapport aux activations de la couche précédente. Les pass backward peut être résumé par la récurrence rétrograde suivante :
Initialisation : $\frac{\partial \mathcal{L}}{\partial a^{[L]}}$.

— régression (MSE) : $\frac{\partial \mathcal{L}}{\partial a^{[L]}} = \frac{2}{m}(a^{[L]} - y)$

— classification (Cross Entropy) : $\frac{\partial \mathcal{L}}{\partial a^{[L]}} = \frac{1}{m}(a^{[L]} - y)$ (y encodé en "one hot").

Entrée : $\frac{\partial \mathcal{L}}{\partial a^{[l]}}$
Entrée provenant du pass forward : $a^{[l-1]}$, $z^{[l]}$

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

— $\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{\partial z^{[l]}}{\partial W^{[l]}}^T \frac{\partial \mathcal{L}}{\partial z^{[l]}} = a^{[l-1]T} \frac{\partial \mathcal{L}}{\partial z^{[l]}}$

— $\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial z^{[l]}}$

— $\frac{\partial \mathcal{L}}{\partial a^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} W^{[l]^T}$

Sortie : $\frac{\partial \mathcal{L}}{\partial a^{[l-1]}}$

Sortie pour la mise à jour des paramètres : $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ et $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$

Remarque : Si l'activation g est la fonction ReLU, lors du pass forward, nous sauvegardons $g^{[l]'}\left(z^{[l]}\right)$ à la place de $z^{[l]}$ afin de profiter de la représentation binaire à faible mémoire.

3.5 Stratégies d'échantillonnage

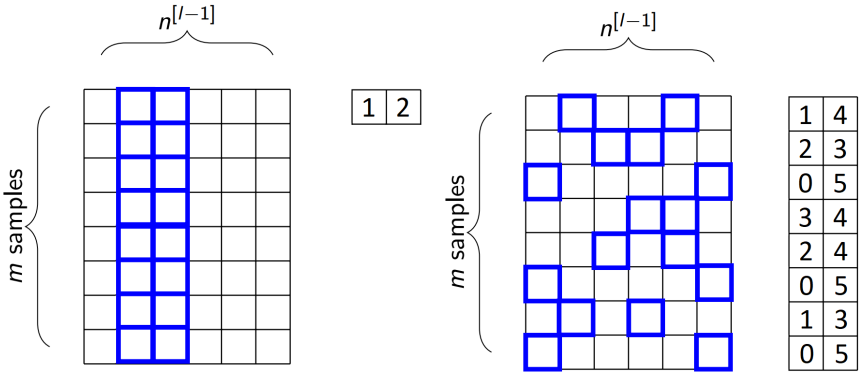
En tenant compte de l'analyse de la complexité spatiale précédente (3.3), nous choisissons d'échantillonner à partir de la jacobienne $\frac{\partial z^{[l]}}{\partial W^{[l]}}$ qui égale à $a^{[l-1]}$. Ainsi, les étapes du pass forward restent exactement les mêmes. Cependant, nous ne conservons qu'une fraction $h \in]0, 1[$ des activations $a^{[0]}, ..., a^{[L-1]}$ pour le pass backward, d'où l'économie en mémoire. Cette stratégie sera appliquée en parallèle avec l'échantillonnage des points de données dans le contexte de la descente de gradient par mini-batch.

Il existe deux schémas d'échantillonnage :

— **"même activation"** : nous tirons avec remise une fraction de h des activations, et les mêmes activations sont choisies pour chaque élément du mini-batch. Dans ce cas, les activations conservées peuvent être directement stockées dans une matrice $\tilde{a}^{[l-1]}$ de taille réduite $m \times \tilde{n}^{[l-1]}$ avec $\tilde{n}^{[l-1]} = \lfloor h \times n^{[l-1]} \rfloor$. Cependant, lors de l'utilisation de ce schéma, seulement une fraction $\tilde{n}^{[l-1]}$ parmi les $n^{[l-1]}$ lignes de la matrice gradients $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ est calculée, ce qui signifie que seule une partie des poids sera mise à jour. Cette approche est alors équivalente à un algorithme d'optimisation connu sous le nom de descente de coordonnées par bloc[8].

— **"activation différente"** : Les activations sont tirées indépendamment pour chaque élément du mini-batch. Comme dans le cas précédent, les activations choisies sont conservées dans une matrice de taille réduite. Cependant, vu qu'on a choisi des activations potentiellement différentes pour chaque échantillon, on doit réorganiser les activations sauvegardées sous la forme d'une matrice de taille $m \times n^{[l-1]}$ (en complétant avec des zéros) avant de l'utiliser dans le pass backward. Contrairement au cas précédent, tous les poids sont mis à jours.

  Remarque : les gradients des biais ne sont pas modifiés pour les deux méthodes d'échantillonnage.



(a) "même activation" (b) "activation différente"

FIGURE 5 – Activations échantillonnées et leurs indices correspondants

3.6 Optimiseurs

3.6.1 Mini-Batch SGD

Le mini-batching est un compromis entre la SGD et la descente de gradient sur tout l'ensemble des données. Au lieu d'utiliser un seul échantillon à la fois ou l'ensemble complet d'apprentissage, nous prenons un petit sous-ensemble et effectuons une estimation Monte Carlo du gradient, puis mettons à jour les paramètres du modèle. La nouvelle règle de mise à jour est la suivante :

$$\theta_{n+1} = \theta_n - \frac{\gamma_{n+1}}{M} \sum_{k=1}^{M} \nabla_\theta l \left(f_{\theta_n}(x_k^{(n+1)}), y_k^{(n+1)} \right), \quad n \geq 0 \tag{5}$$

Le mini-batch SGD peut être préféré à la SGD pour deux raisons principales. La première est qu'il fournit un estimateur de variance plus faible, ce qui peut conduire à des mises à jour moins bruitées et une meilleure vitesse de convergence. La deuxième raison est qu'il peut profiter de la parallélisation. Nous avons constaté dans la section (3.4) qu'il est possible d'effectuer la rétropropagation pour un sous-ensemble d'échantillons en utilisant des opérations matricielles, ce qui peut tirer parti des capacités de calcul des GPU.

En examinant l'algorithme de rétropropagation du réseau de neurones dans la section précédente, nous remarquons qu'une partie des matrices manipulées dépend de la taille du batch (sur la première dimension). Par conséquent, ce paramètre représente un compromis entre la variance de l'estimation du gradient et l'utilisation de la mémoire. Le cadre de différenciation automatique randomisée introduit un hyperparamètre supplémentaire, qui est la fraction d'activation échantillonnée h, permettant également de trouver un compromis entre le gain de mémoire et la variance du gradient. L'algorithme dans ce cas s'écrit :

$$\theta_{n+1} = \theta_n - \frac{\gamma_{n+1}}{M} \sum_{k=1}^{M} \widehat{\nabla_\theta l}^{rad,h} \left(f_{\theta_n}(x_k^{(n+1)}), y_k^{(n+1)} \right), \quad n \geq 0 \tag{6}$$

3.6.2 Weight decay

Le weight decay, également connu sous le nom de régularisation L2 , est une technique couramment utilisée en apprentissage automatique pour prévenir le sur-apprentissage et améliorer la capacité de généralisation d'un modèle. En pénalisant les poids élevés, le weight decay contribue à limiter la complexité du modèle et à éviter qu'il ne devienne trop sensible aux données d'entraînement. La nouvelle fonction de côut empirique et l'algorithme de descente sont donnée par :

$$L(\theta) = \frac{1}{N}\sum_{k=1}^{N} l\left(f_\theta(x_k), y_k\right) + \lambda\|\theta\|^2 \tag{7}$$

$$\theta_{n+1} = \theta_n - \gamma_{n+1}\left(\frac{1}{M}\sum_{k=1}^{M} \widehat{\nabla_\theta l}^{rad,h}\left(f_{\theta_n}(x_k^{(n+1)}), y_k^{(n+1)}\right) + 2\lambda\cdot\theta_n\right), \quad n \geq 0 \tag{8}$$

3.6.3 ADAM

Adam (*Adaptative moment estimation*) [9] est un algorithme d'optimisation du premier ordre. Cette méthode est une combinaison des méthodes Adadelta/RMSprop [10][11] et de la descente de gradient avec momentum [12], ce qui aide à accélérer la descente de gradient dans la direction pertinente et à atténuer les oscillations tout en adaptant le taux d'apprentissage individuellement pour chaque poids. Si l'on désigne par g_n le gradient de la fonction objective par rapport aux paramètres, la règle de mise à jour est la suivante :

$$\begin{cases} M_n = \beta_1 \times M_{n-1} + (1-\beta_1)\times g_n \\ V_n = \beta_2 \times V_{n-1} + (1-\beta_2)\times g_n \odot g_n \\ \theta_n = \theta_{n-1} - \alpha\cdot\frac{M_n}{\sqrt{V_n+\epsilon}} \end{cases}$$

Les hyperparamètres sont les suivants :

- α : taux d'apprentissage (doit être ajusté).
- β_1 : 0.9 (taux de décroissance exponentielle du premier moment).
- β_2 : 0.999 (taux de décroissance exponentielle du deuxième moment).
- ϵ : 10^{-8} (pour éviter la division par zéro).

Remarque : l'algorithme ci-dessus est une version simplifiée de l'original qui utilise une correction de biais pour les estimations du premier et du deuxième moment.

3.7 Implémentation informatique

Toutes les implémentations et expérimentations ont été menées à bien en utilisant Python, sur un serveur interne équipé d'un puissant GPU Nvidia A100.

L'implémentation est de bas niveau dans le sens où toutes les couches et réseaux neuronaux ont été construits à partir de zéro en utilisant uniquement des opérations matricielles simples et des fonctions mathématiques standard. Nous avons utilisé CuPy.

CuPy est une bibliothèque open-source basée sur les bibliothèques du CUDA Toolkit. Elle permet d'exécuter des calculs sur des tableaux multidimensionnels en utilisant des GPU. Cupy fournit une interface similaire à NumPy et exploite la puissance de calcul parallèle des GPU grâce à CUDA. En utilisant les fonctionnalités CUDA, CuPy permet d'accélérer les calculs et d'obtenir des performances optimisées par rapport à l'exécution sur CPU.

3.8 Choix impactant la complexité spatiale

3.8.1 Choix des types de variables

Étant donné que notre principal objectif est de réduire la consommation de mémoire, il est important de choisir soigneusement le type de variable pour une implémentation optimale en termes de RAM.

Pour les différents indices, nous avons choisi de travailler avec le type `uint16` au lieu du type par défaut `int64`. La plage de `uint16` est [0, 65535], ce qui ne nous restreint pas car le nombre de neurones ne dépasse généralement pas quelques centaines. Cela réduit de 4 fois la mémoire allouée pour enregistrer tous les indices d'activation échantillonnés, ce qui peut représenter une partie non négligeable de la consommation de mémoire persistante lors de la propagation avant avec le mode "activation différente".

Un autre point important concerne le choix de la variable utilisée pour récupérer la jacobienne de l'activation durant le pass backward. Lors de la propagation avant, pour une activation générale, il est équivalent en termes de mémoire de sauvegarder $z^{[l]}$ ou $g^{[l]'}\left(z^{[l]}\right)$. Cependant, dans le cas de l'activation ReLU, nous pouvons profiter du fait que la jacobienne est binaire et ne sauvegarder que $g^{[l]'}\left(z^{[l]}\right)$ en tant qu'entrée pour la rétropropagation. Par conséquent, cette astuce peut théoriquement réduire la consommation de mémoire persistante lié à la cette jacobienne d'un facteur de 64. Cependant, le type `cupy.bool` occupe 1 octet, ce qui représente 8 fois la mémoire requise. Par conséquent, nous devons encoder manuellement la jacobienne en utilisant `cupy.packbits` et la décoder par la suite avec `cupy.unpackbits` pour l'utiliser dans les calculs des gradients comme dans 3.4.3.

3.8.2 Représentation sparse

Dans le cas où nous échantillonnons des activations différentes, il est possible de sauvegarder le résultat après la propagation avant sous forme de matrice creuse. Nous pouvons utiliser l'implémentation de `cupyx.scipy.sparse` pour créer et manipuler des matrices creuses, qui sont des matrices principalement composées de zéros. Cette représentation nous permet d'économiser de l'espace mémoire en ne stockant que les valeurs non nulles de la matrice, ainsi que leurs indices.

L'avantage de cette approche est que nous n'avons pas besoin d'allouer de mémoire temporaire pendant la rétropropagation pour reconstituer la matrice des activations de taille complète $m \times n^{[l]}$, car `cupyx.scipy.sparse` permet de réaliser des multiplications entre une matrice dense et une matrice creuse. Théoriquement, cela peut accélérer les calculs en exploitant la structure de la matrice creuse pour ne considérer

que les éléments non nuls. Cependant, en pratique, le code devient environ 5 fois plus lent.

3.8.3 Sauvegarder ou régénérer les indices ?

Avoir les indices qui ont été utilisés dans l'échantillonnage des activations est nécessaire pendant la phase de rétropropagation et de descente de gradient afin de reconstituer la matrice complète dans le cas où le mode est défini comme "activation différente", et de mettre à jour les bonnes composantes des poids dans le cas où le mode est défini comme "même activation". Ainsi, tous les indices sont accumulés de la même manière que les activations correspondantes pendant la phase de propagation en avant.

Dans le cas "même activation", étant donné que nous utilisons très peu d'indices, cela est négligeable par rapport à l'espace alloué aux activations. Cependant, dans le cas "activation différente", cela représente une allocation de mémoire persistante supplémentaire importante. Par conséquent, il est envisageable de régénérer ces indices au lieu de les sauvegarder. En pratique, lors de la propagation en avant, pour chaque couche, nous pouvons générer un nombre entier servant de graine (seed) pour générer les indices, puis le sauvegarder afin de pouvoir les régénérer ultérieurement.

3.9 Principales classes et des fonctions

— `RadLayer` : une classe générique similaire aux `layers` de Keras ou à `nn.Module` de PyTorch. Comme son nom l'indique, cette classe implémente une couche générique d'un réseau neuronal, mais avec la possibilité d'utiliser la différenciation automatique randomisée. Elle contient des variables d'état telles que les poids et les biais de la couche, ainsi que les moments lorsque l'optimiseur ADAM est utilisé. Elle implémente la méthode `gradient_descent`, qui est la même indépendamment des spécificités de la couche. De plus, elle contient des méthodes `forward` et `backward` vides qui doivent être implémentées par les classes filles.

— `truncate` : est une fonction qui prend en entrée un tableau d'activations et renvoie le tableau tronqué ainsi que les indices des activations sélectionnées. Cette fonction prend en charge deux modes :

  — "même activation" : on tire avec remise $\lfloor h \times n^{[l-1]} \rfloor$ entiers.

  — "activation différente" : on tire avec remise $m \times \lfloor h \times n^{[l-1]} \rfloor$ entiers.

  Dans les deux cas, le tableaux résultant doit être multiplié par $\frac{\lfloor h \times n^{[l-1]} \rfloor}{n^{[l-1]}}$ pour avoir un estimateur non-biasé du gradient.

— `LinearRadLayer` : Il s'agit d'une classe qui hérite de `RadLayer`. Elle représente une couche sans activation qui peut être utilisée à l'intérieur du réseau ou en tant que couche de sortie pour des problèmes de régression.

  — La méthode `forward` effectue simplement une multiplication par les poids et l'ajout du terme de biais, comme dans l'équation de la section 3.4.2,

avec g égale à la fonction identité. La nouveauté réside dans le fait que, lorsque le mode RAD est utilisé, nous sauvegardons une fraction des activations d'entrée pour la rétropropagation, ainsi que leurs indices correspondants. La sauvegarde des indices est nécessaire car elle permet de savoir quelles composantes des poids mettre à jour lorsque le mode d'activation "même activation" est utilisé, et de reconstruire le tableau complet des activations à utiliser lors de l'étape de rétropropagation en mode "activation différente".

— La méthode `backward` implémente l'équation de la section 3.4.3 en utilisant les variables sauvegardées lors de la propagation avant et le gradient de la perte par rapport aux activations de sortie.

  — "même activation" : la méthode utilise directement les activations tronquées pour calculer le gradient par rapport aux poids. Ces gradients ont une taille réduite correspondant uniquement aux composantes des activations d'entrée choisies.

  — "activation différente" : nous devons convertir les activations à leur forme originale en ajoutant des zéros, ce qui donne un tableau sparse. Dans ce cas, nous obtenons un estimateur de gradient de taille complète.

— `ReluRadLayer` : Similaire à `LinearRadLayer`, mais avec une implémentation efficace en termes de mémoire des activations ReLU et de leur dérivée, en utilisant les fonctions d'encodage/décodage.

— `SoftmaxRadLayer`

— `NeuralNetworkClassifier` : Réseaux de neurones pour la classification, permet d'empiler les couches définies précédemment avec une couche `SoftmaxRadLayer` en sortie. les méthodes de cette classe sont : `forward`, `backward`, `train` et `predict`.

— `NeuralNetworkRegressor` : Réseaux de neurones pour la régression, similaire à `NeuralNetworkClassifier` mais avec une couche `LinearRadLayer` en sortie.

3.10 Memory profiling

Pour mieux analyser la méthode de différenciation automatique randomisée, nous avons effectué une étude approfondie de l'utilisation de la mémoire RAM. Pour ce faire, nous avons utilisé la fonctionnalité `LineProfileHook` de la bibliothèque CuPy. Cette fonctionnalité nous a permis de suivre les allocations et les libérations de mémoire qui se produisent pendant l'exécution du programme.

Le `LineProfileHook` attribue des étiquettes uniques à chaque allocation et libération de mémoire, et enregistre ces informations ainsi que les numéros de ligne correspondants dans le code source. Cela nous a donné une vision détaillée de la quantité de mémoire allouée et libérée à chaque ligne de code.

Dans le cadre de notre étude expérimentale, nous avons divisé les allocations de mémoire en deux catégories en fonction de leur durée de vie dans la mémoire : **persistante** ou **temporaire**. Nous considérons qu'une variable est temporaire si

elle n'est présente en mémoire que pendant une seule itération (correspondant à une seule couche) lors de la propagation avant ou de la rétropropagation.

Un exemple d'allocation temporaire est la création de la variable $\frac{\partial \mathcal{L}}{\partial z^{[l]}}$ pendant chaque itération du pass backward qui servira comme intermédiaire pour le calcul $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ et $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$. Pour les variables persistantes, on peut citer les paramètres du réseau ou bien les activations qui seront en mémoire durant tout le pass forward pour être utilisées dans le calcul des gradients durant le pass backward.

L'analyse des allocations de mémoire nous a également permis d'identifier des améliorations potentielles du code afin d'obtenir une implémentation optimale en termes d'utilisation de la RAM. Par exemple, lors de la multiplication d'une matrice d'activations a par un scalaire s pour la normaliser, même si nous affectons le résultat à une variable existante, un espace supplémentaire et temporaire de la même taille que a est alloué pour stocker le résultat de $s \times a$. Cependant, il est possible d'utiliser des opérations "inplace" de CuPy pour éviter cette allocation temporaire.

Pour confirmer les résultats expérimentaux et le profilage de la mémoire, nous avons réalisé une analyse théorique détaillé de la complexité spatiale en tenant compte de chaque opération élémentaire dans le code. Nous avons exprimé toutes les complexités en termes du nombre d'unités de type `float64`. Cette analyse a couvert toutes les étapes de l'entraînement du réseau : la propagation avant, la rétropropagation et la descente de gradient. Elle nous a permis de comprendre la dépendance de l'utilisation de la mémoire à chaque étape en fonction du modèle ou des hyperparamètres d'entraînement tels que la taille du mini-batch, le nombre de neurones et le nombre de couches cachées.

3.11 Premiers tests

3.11.1 Vérification des gradients analytiques

Puisque le cœur des méthodes étudiées est le calcul du gradient, nous avons vérifié les gradients analytiques obtenus avec l'algorithme de rétropropagation en utilisant une approximation par différences finies. La version centrée des différences finies est donnée par :

$$[\nabla \mathcal{L}(W)]_{i,j} \approx \frac{\mathcal{L}(W + \epsilon \cdot e_{i,j}) + \mathcal{L}(W - \epsilon \cdot e_{i,j})}{2 \cdot \epsilon}$$

Ainsi, pour estimer le gradient à l'aide de cette méthode, pour chaque couche et pour chaque composante (i, j) de la matrice de poids correspondante, nous ajoutons une perturbation ϵ . Ensuite, nous effectuons une propagation avant à travers le réseau de neurones et calculons la perte $\mathcal{L}(W + \epsilon \cdot e_{i,j})$. Nous répétons la même opération en soustrayant ϵ pour obtenir la perte $\mathcal{L}(W - \epsilon \cdot e_{i,j})$. Enfin, il est important de rétablir la composante du poids à sa valeur initiale avant de passer à l'approximation de la composante suivante.

3.11.2 Test sur un problème de classification

Nous avons testé un réseau de classification sur l'ensemble de données MNIST, largement utilisé comme référence dans le domaine de l'apprentissage automatique

et de la vision par ordinateur. Les images de MNIST représentent des chiffres écrits à la main allant de 0 à 9. Les images contiennet 28x28 pixels et sont en niveaux de gris à un seul canal, ce qui signifie que chaque valeur représente l'intensité du pixel, variant de 0 (noir) à 255 (blanc). Chaque échantillon est étiqueté avec le chiffre correspondant qu'il représente. L'ensemble de données se compose de 60 000 exemples d'entraînement et de 10 000 exemples de test. De plus, l'ensemble de données est équilibré, avec un nombre égal d'exemples pour chaque classe de chiffres.

3.11.3 Test sur un problème de régression

Pour tester les performances du réseaux sur la tâche de régression, nous avons utilisés le dataset "diabetes". L'ensemble de données "diabetes" de scikit-learn est issu de l'étude sur le diabète de la clinique de recherche du NIDDK. Il comprend 442 exemples avec 10 caractéristiques biomédicales prétraitées, telles que l'âge, l'IMC, la pression artérielle et les mesures sanguines. L'objectif est de prédire la progression de la maladie du diabète après un an de traitement. Ce jeu de données est couramment utilisé pour l'entraînement et l'évaluation de modèles de régression dans le domaine de l'apprentissage automatique.

3.12 Premiers résultats

3.13 MNIST dataset

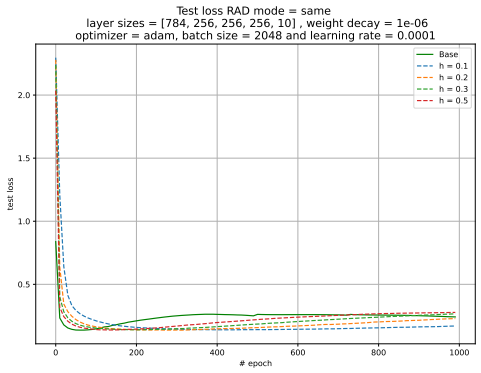


FIGURE 6 – Entropie Croisée

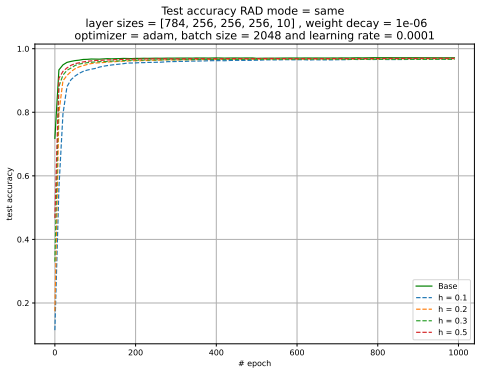


FIGURE 7 – Précision

FIGURE 8 – Évolution de la performance sur le test dataset en fonction des époques avec la stratégie d'échantillonnage "même activation". Tous les réseaux convergent vers une précision similaire. En ce qui concerne la perte, nous remarquons que plus la valeur de la fraction d'activations échantillonnées h est basse, plus la convergence est lente, ce qui est attendu car cela signifie des gradients plus bruités.

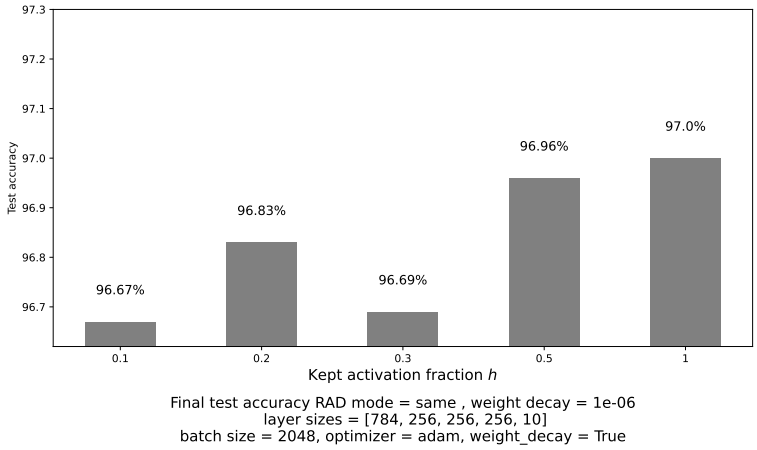


FIGURE 9 – Précisions finales sur l'ensemble de données de test avec la méthode "même activation"

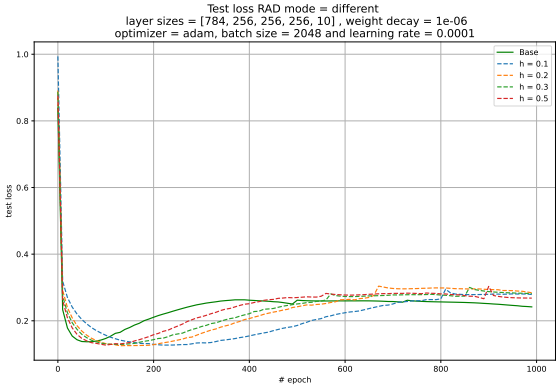


FIGURE 10 – Entropie Croisée

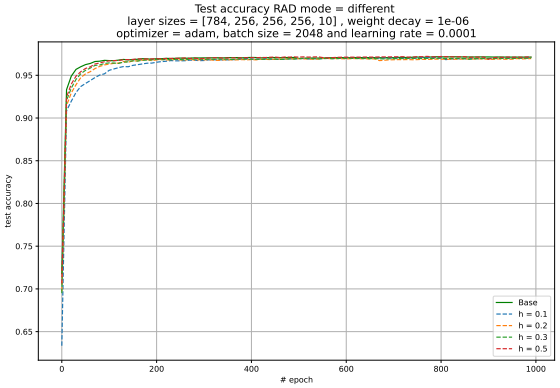


FIGURE 11 – Précision

FIGURE 12 – Évolution de la performance sur l'ensemble de données de test en fonction des époques avec la stratégie d'échantillonnage des "activations différentes". Comme dans le cas précédent, on obtient des performances similaires pour les différentes valeurs de h.

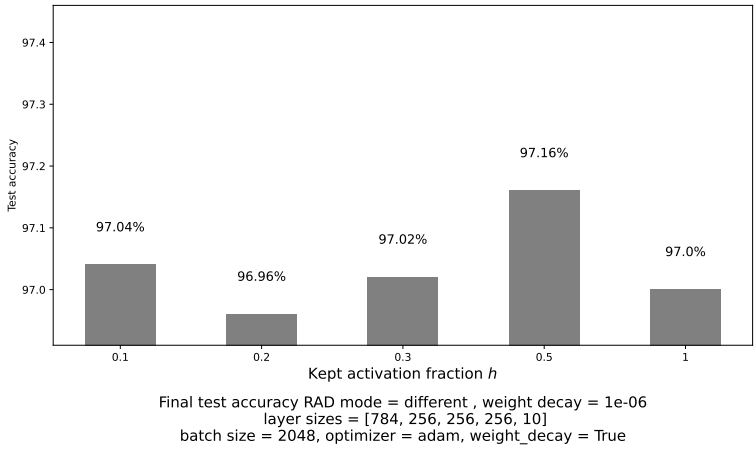


FIGURE 13 – Précisions finales sur l'ensemble de données de test avec la méthode "activation différente". Dans cet exemple, l'utilisation de la différenciation automatique randomisée conduit à de meilleures performances.

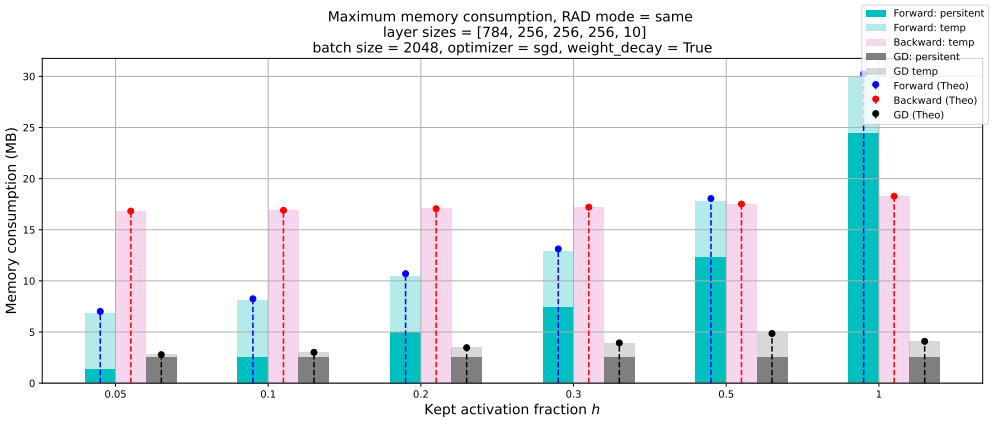


FIGURE 14 – Profilage de la mémoire avec la méthode "même activation" pour différentes valeurs de h. L'analyse théorique de la mémoire correspond parfaitement aux allocations physiques.
Forward (temporaire) : $z^{[l]}$, $g^{[l]'}\left(z^{[l]}\right)$ (avant l'encodage), calcul de softmax, etc.
Forward (persistant) : $\tilde{a}^{[l]}$, $g^{[l]'}\left(z^{[l]}\right)$ encodé et indices échantillonnés.
Backward (temporaire) : décodage de $g^{[l]'}\left(z^{[l]}\right)$, calcul de $\frac{\partial \mathcal{L}}{\partial z^{[l]}}$, $\frac{\partial \mathcal{L}}{\partial a^{[l-1]}}$, $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ et $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$.
Descente de gradient (persistante) : $W^{[l]}$ et $b^{[l]}$.
Descente de gradient (temporaire) : opérations de slicing.

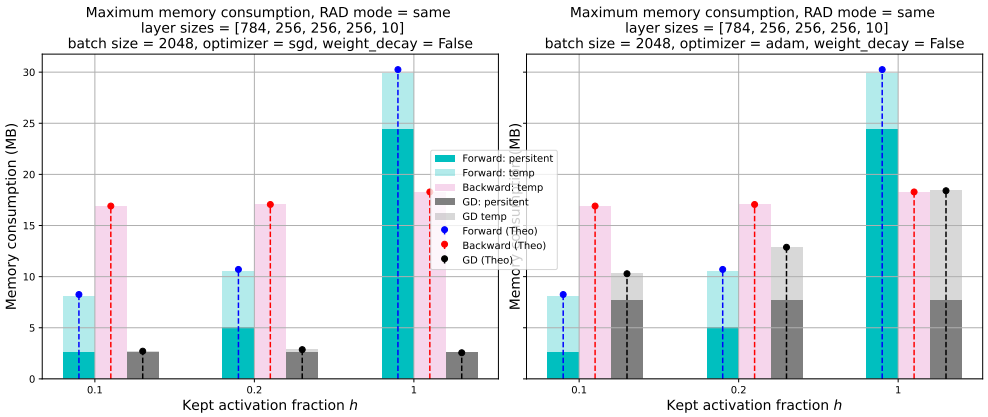


FIGURE 15 – Profilage de la mémoire avec deux optimiseurs différents : SGD standard et ADAM. ADAM consomme trois fois plus de mémoire persistante et entraîne également une allocation de mémoire temporaire supplémentaire pour les calculs intermédiaires et les opérations de slicing (lors de l'utilisation de RAD).

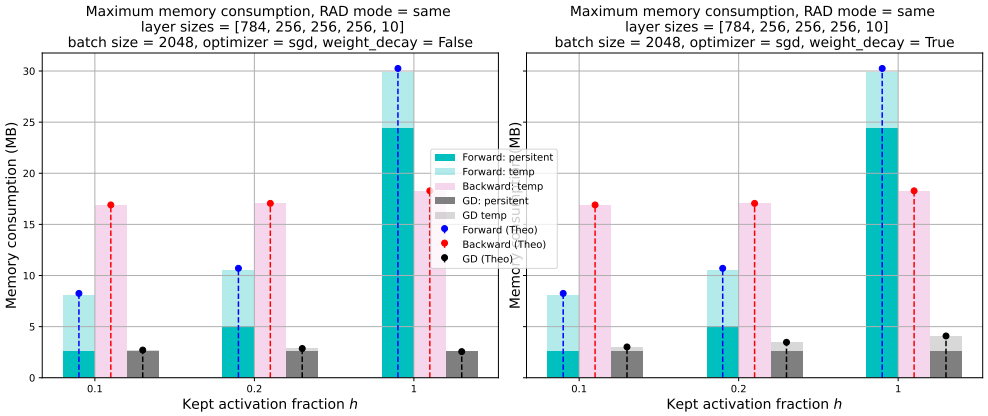


FIGURE 16 – Effet de l'ajout du weight decay sur la consommation de mémoire. Descente de gradient : aucune allocation temporaire pour le cas de base car nous pouvons utiliser des opérations inplace. Lorsque nous multiplions les poids par λ, une allocation temporaire est nécessaire.

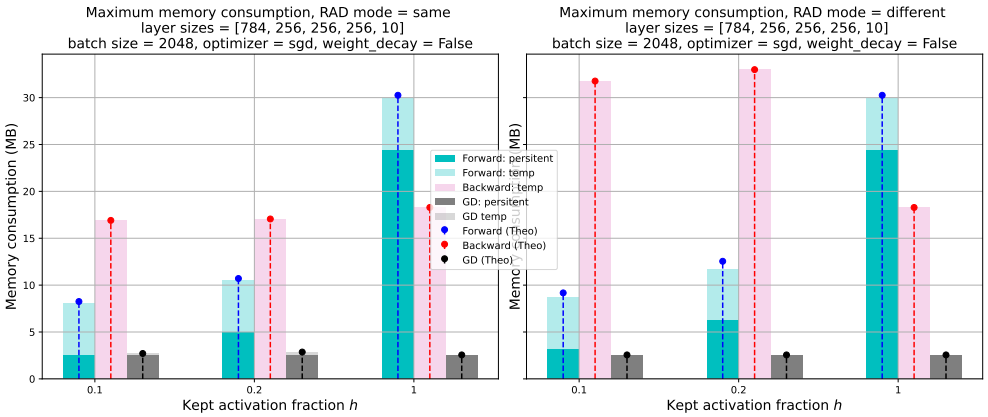


FIGURE 17 – Comparaison des allocations mémoire des deux stratégies d'échantillonnage. Les indices échantillonnés requièrent plus d'espace lorsque l'on utilise le mode est "activation différente", ce qui entraîne une allocation de mémoire persistante supplémentaire durant le pass forward. Aucun slicing n'est nécessaire pendant la descente de gradient lorsque pour cette stratégie d'échantillonnage, ce qui évite toute allocation temporaire. Une consommation temporaire supplémentaire intervient pendant la rétropropagation pour ce mode d'échantillonnage afin reconstruire $\tilde{a}^{[l]}$. Un désaccord entre le profilage et l'analyse théorique (Forward : temp) est dû à la non-prise en compte de la mémoire temporaire utilisée par le générateur aléatoire.

3.14 Diabetes dataset

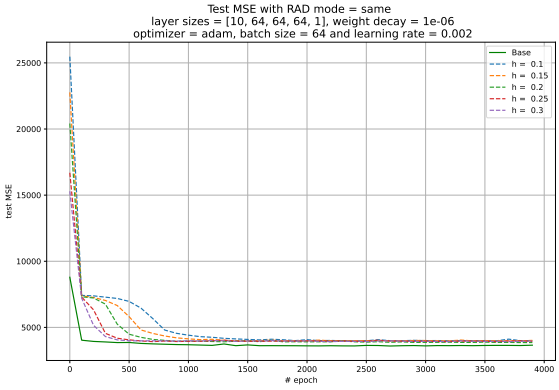


FIGURE 18 – MSE

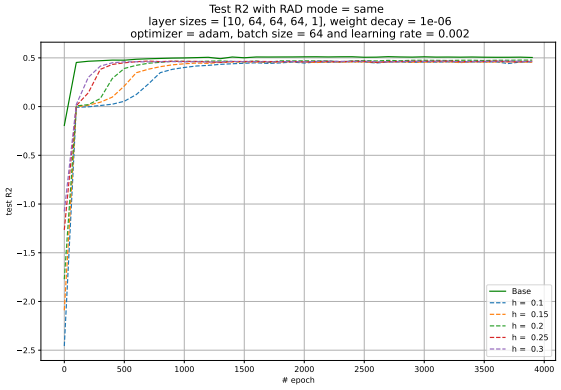


FIGURE 19 – Coefficient de détermination R^2

FIGURE 20 – Évolution de la performance sur test dataset en fonction des époques avec la stratégie d'échantillonnage "même activation".

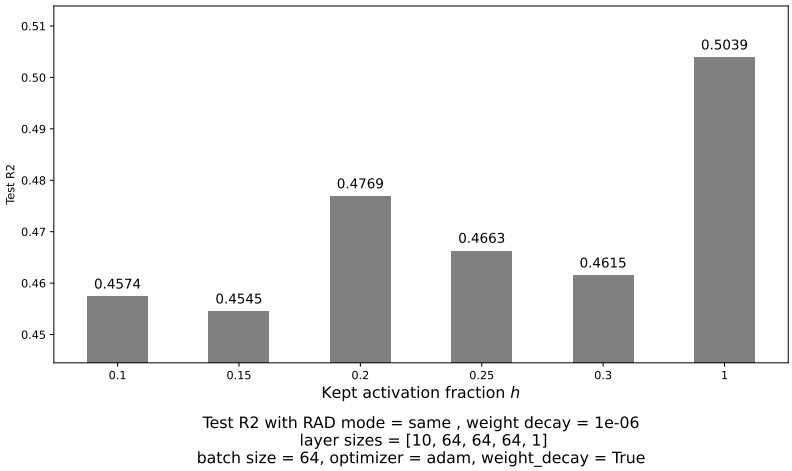


FIGURE 21 – Coefficients de détermination finaux sur l'ensemble de données de test avec la méthode "même activation"

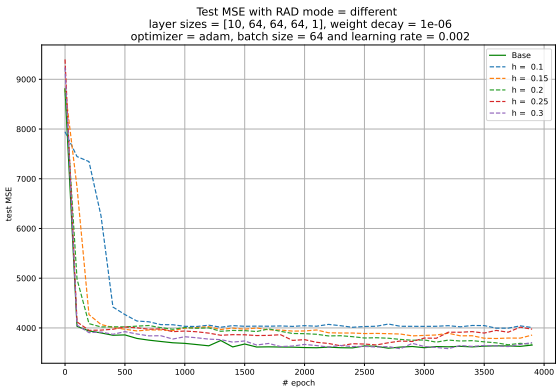


FIGURE 22 – MSE

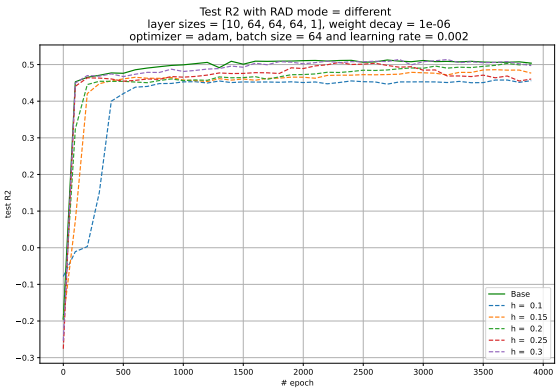


FIGURE 23 – Coefficient de détermination R^2

FIGURE 24 – Évolution de la performance sur test dataset en fonction des époques avec la stratégie d'échantillonnage "activation différente".

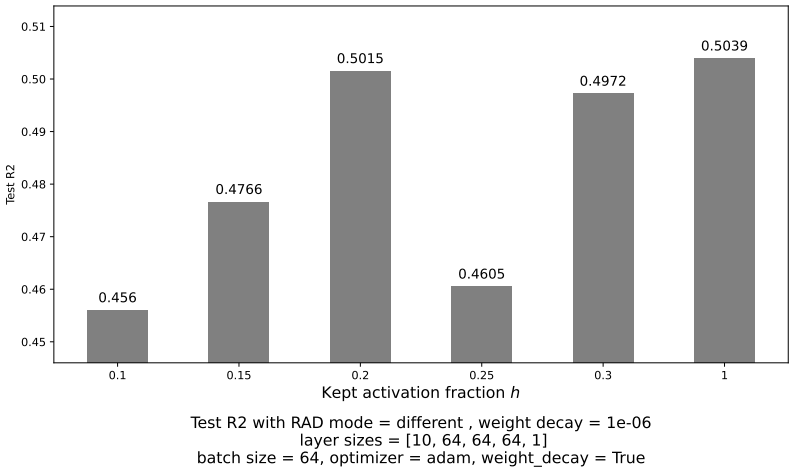


FIGURE 25 – Coefficients de détermination finaux sur l'ensemble de données de test avec la méthode "activation différente"

4 Pricing avec réseaux de neurones

4.1 Projection orthogonale

Soit (X, Y) un couple de variables aléatoires de carré intégrable où $X \in E$ et $Y \in \mathbb{R}$. Nous nous situons dans le contexte où l'on désire approximer une espérance conditionnelle au moyen d'une fonction déterministe h paramétrée par $\theta \in \Theta$.

$$\mathbb{E}[(h(\theta, X) - Y)^2] = \mathbb{E}[h(\theta, X)^2 - 2 \cdot h(\theta, X) \cdot \mathbb{E}[Y|X] + \mathbb{E}[Y^2|X]] \tag{9}$$
$$= \mathbb{E}[(h(\theta, X) - \mathbb{E}[Y|X])^2] + \mathbb{E}[\mathbb{V}ar(Y|X)] \tag{10}$$

Étant donné que le deuxième terme est indépendant de θ :

$$\theta^* = \arg\min_{\theta \in \Theta} \mathbb{E}[(h(\theta, X) - \mathbb{E}[Y|X])^2] = \arg\min_{\theta \in \Theta} \mathbb{E}[(h(\theta, X) - Y)^2]$$

En pratique, on se donne M réalisations du couple (X, Y) et on cherche à minimiser le risque empirique :

$$\hat{\theta}_M = \arg\min_{\theta \in \Theta} \frac{1}{M} \sum_{i=1}^{M} (Y_i - h(\theta, X_i))^2$$

4.2 Lien avec le pricing

Soit $(\Omega, \mathcal{A}, (F_t)_{t \geq 0}, \mathbb{Q})$ un espace probabilisé filtré muni d'une probabilité risque neutre ainsi qu'un mouvement Brownien standard à valeurs dans $\mathbb{R}^d$ $(W_t)_{t \geq 0}$ pour certains $d \geq 1$. Soient $f : \mathbb{R}^+ \times \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ et $g : \mathbb{R}^+ \times \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^m$ deux fonctions différentiables par morceaux telles que nous avons, pour tout $t \geq 0$, $x, y \in \mathbb{R}^d$ et $\xi, \xi_0 \in \mathbb{R}^n$:

$$\|f(t, x, \xi) - f(t, y, \xi)\| + \|g(t, x, \xi) - g(t, y, \xi)\| \leq K\|x - y\|,$$
$$\|f(t, x, \xi) - f(t, x, \xi_0)\| + \|g(t, x, \xi) - g(t, x, \xi_0)\| \leq K(1 + \|x\|)\|\xi - \xi_0\|,$$
$$\|f(t, x, \xi)\| + \|g(t, x, \xi)\| \leq K(1 + \|x\|).$$

Pour un certain $K > 0$, où les normes sont les normes euclidiennes habituelles pour les vecteurs et les normes de Frobenius pour les matrices. Pour chaque $\xi \in \mathbb{R}^n$, nous définissons $(X_t^\xi)_{t \geq 0}$ comme étant une solution forte à l'équation différentielle stochastique multidimensionnelle suivante :

$$dX_t^\xi = f(t, X_t, \xi)\, dt + g(t, X_t, \xi)\, dW_t$$

Nous supposons la même valeur initiale déterministe X_0 pour tous les $\xi \in \mathbb{R}^n$. Le vecteur ξ représente les paramètres du modèle.

Pour simplifier, nous supposerons des taux d'intérêt nuls. Nous considérons un contrat de maturité T défini par les paramètres β et un payoff stochastique $Y^{\xi,\beta}$. Ici, ξ et β sont des vecteurs aléatoires mututellment indépendants et indépendants de $(B_t)_{t \geq 0}$. Le prix de ce contrat est alors calculé comme suit :

$$h^*(\xi, \beta, T) = \mathbb{E}[Y^{\xi,\beta}|\xi, \beta, T]$$

On cherchera à approximer l'espérance conditionnelle précédente par une fonction paramétrique appartenant à un espace suffisamment riche :

$$\theta^* = \arg\min_{\theta\in\Theta} \mathbb{E}[(h(\theta,\xi,\beta,T) - \mathbb{E}[Y^{\xi,\beta}|\xi,\beta,T])^2] = \arg\min_{\theta\in\Theta} \mathbb{E}[(h(\theta,\xi,\beta,T) - Y^{\xi,\beta})^2].$$

En pratique, pour $i = 1, .., N_{train}$, on tire un échantillon(ξ_i, β_i, T_i). Conditionnellement à ces valeurs, on diffuse le processus (X_t^ξ) par un schéma d'Euler et on calcule une réalisation du payoff $Y_i^{\xi_i,\beta_i}$. Finalement, on résout le problème de régression suivant :

$$\hat{\theta}_{N_{train}} = \arg\min_{\theta\in\Theta} \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} (Y_i^{\xi_i,\beta_i} - h(\theta,\xi_i,\beta_i,T_i))^2$$

Dans ce projet, le choix de la famille de fonctions paramétrées a été orienté vers l'utilisation de réseaux de neurones, motivé par le théorème suivant :

Théorème d'Approximation Universelle [13] : Si σ est une fonction d'activation non-affine continûment différentiable, $q \in \mathbb{N}$, et $K \subseteq \mathbb{R}^n$ est compact, alors pour tout $\epsilon > 0$ et $f \in C(K, \mathbb{R}^q)$, il existe un réseau de neurones $\mathcal{NN}_{n,q}^\sigma : \mathbb{R}^n \to \mathbb{R}^q$ avec p couches cachées, $q + n + 2$ neurones par couche cachée, et σ comme fonction d'activation, tel que

$$\sup_{x\in K} \|\mathcal{NN}_{n,q}^\sigma(x) - f(x)\| < \epsilon.$$

Pour plus de détails sur cette méthode nous renvoyons le lecteur à [14].

4.3 Calcul des sensibilités

Soit $x = (\xi, \beta, T)$ l'entrée du réseau. Un avantage de l'utilisation des réseaux de neurones est d'avoir accès au gradient analytique de la sortie par rapport aux entrées. Pour ce faire, nous modifions légèrement l'algorithme de rétropropagation de la section 4.4.3. Nous considérons le cas simple où la sortie du réseau est un scalaire et nous fixons la fonction de perte égale à la sortie, c'est-à-dire $\mathcal{L}(a^{[L]}, y) = a^{[L]}$. L'algorithme modifié est le suivant :

Algorithm 1 Calcul des gradients par rapport aux entrées

Initialisation $\frac{\partial \mathcal{L}}{\partial a^{[L]}} \leftarrow \mathbf{1}$
for $l = L$ *à* 1 **do**
  $\frac{\partial \mathcal{L}}{\partial z^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g^{[l]'}(z^{[l]})$
  $\frac{\partial \mathcal{L}}{\partial a^{[l-1]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[l]}} W^{[l]^T}$
end
Sortie : $\frac{\partial a^{[L]}}{\partial x}$

Cet algorithme reste inchangé que nous utilisions ou non la différentiation automatique randomisée, car cette dernière n'affecte que les activations sauvegardées qui sont utilisées uniquement pour calculer les gradients par rapport aux poids $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ lors de la rétropropagation.

4.4 Réduction de variance

Nous pouvons remplacer $Y_i^{\xi_i,\beta_i}$ par une moyenne de N_{red} indépendantes du payoff conditionnellement à la même réalisation de ξ_i, β_i, T_i : $Z_i^{\xi_i,\beta_i} = \frac{1}{N_{red}} \sum_{j=1}^{N_{red}} Y_j^{\xi_i,\beta_i}$. En effet :

$$\mathbb{E}[Z_i^{\xi_i,\beta_i}|\xi_i,\beta_i,T_i] = \mathbb{E}[Y_i^{\xi_i,\beta_i}|\xi_i,\beta_i,T_i] \text{ et } \mathbb{V}ar[Z_i^{\xi_i,\beta_i}|\xi_i,\beta_i,T_i] = \frac{\mathbb{V}ar[Y_i^{\xi_i,\beta_i}|\xi_i,\beta_i,T_i]}{N_{red}}$$

4.5 Exemple simplifié

Considérons le cas où nous souhaitons calculer le prix d'un Call en utilisant une régression par des réseaux de neurones. Ici, $\xi = (S, r, \sigma)$ représentent respectivement le prix initial du sous-jacent, la volatilité et le taux d'intérêt. De plus, le payoff de ce contrat est défini comme $Y^{\xi,\beta} = (X_T^{\xi} - K)_+$ avec $\beta = K$ le prix d'exercice du Call. Pour simplifier, nous fixons tous les paramètres sauf S. Étant donné que nous disposons de formules fermées pour le prix, nous pouvons facilement évaluer les estimations fournies par cette méthode. Nous pouvons ainsi calculer le taux d'erreur absolu moyen en pourcentage (MAPE) par rapport à S sur un ensemble de données de test :

$$\text{MAPE} = \sum_{k=1}^{N_{test}} \frac{|Call_{\mathcal{NN}}(S_k)) - Call_{BS}(S_k)|}{S_k}$$

Nous avons essayé trois différentes stratégies d'optimisation : sans différenciation automatique aléatoire, échantillonnage avec "même activation" et échantillonnage avec "activation différente". Les prix obtenus par régression neuronale sont très proches de ceux de la formule fermée. En ce qui concerne le delta, nous constatons que dans les trois cas, nous obtenons des résultats similaires : une fonction constante par morceaux, ce qui confirme que les réseaux ReLU approximent à l'aide de fonctions affines par morceaux.

TABLE 1 – Paramètres de la simulation numérique

Variable	Valeur
T	1
r	0.03
σ	0.15
K	100
Loi(S)	$\mathcal{U}(80, 120)$
N_{red}	16
N_{train}	2^{20}
N_{test}	100

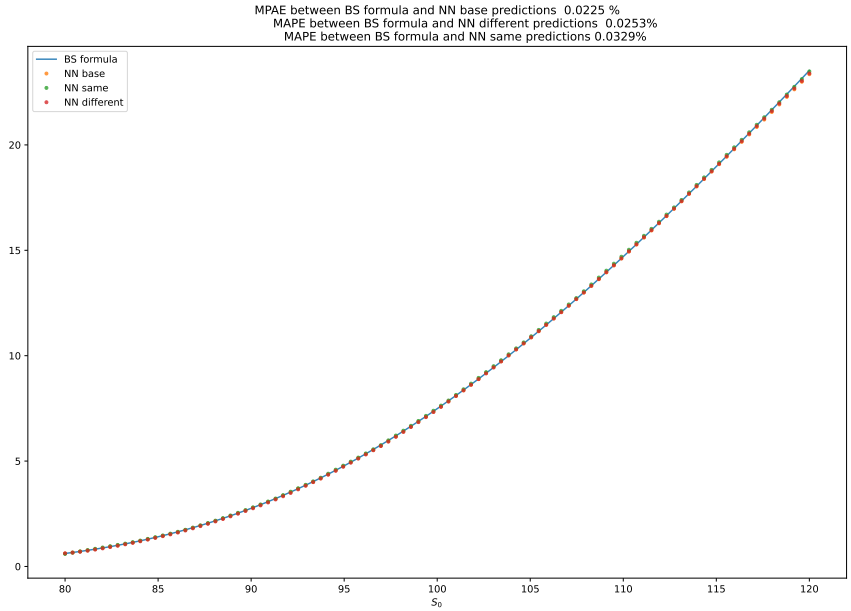


FIGURE 26 – Comparaison entre le prix de la formule fermée et les prix obtenus par régression neuronale en utilisant différentes stratégies d'entraînement.

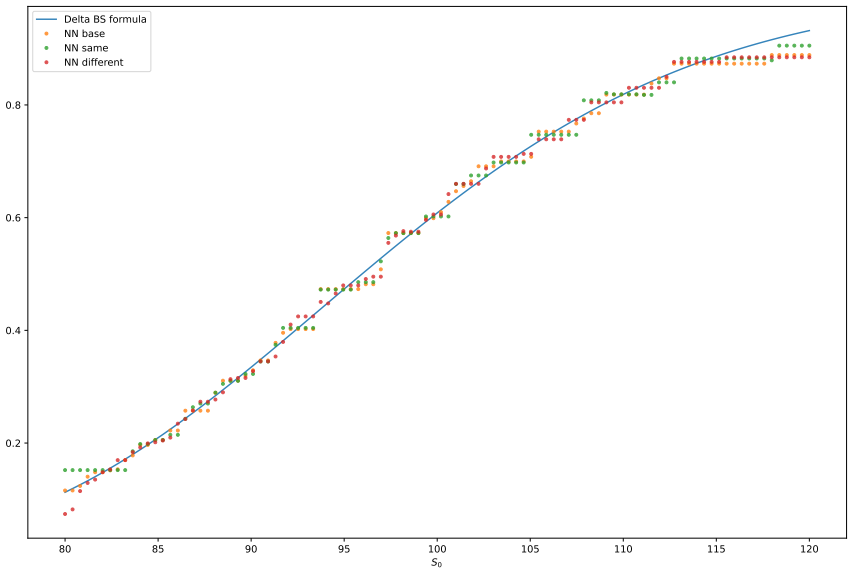


FIGURE 27 – Comparaison entre le delta de la formule fermée et le delta obtenu grâce à la différenciation automatique par rapport à l'entrée après l'entraînement des réseaux.

5 Modèles de crédit

5.1 Approche structurelle

L'approche structurelle de la modélisation du risque de crédit se concentre sur la modélisation de la faillite à partir de la valeur des actifs d'une entreprise. L'événement de défaut de crédit se produit lorsque les actifs d'une entreprise tombent en dessous d'un certain niveau pré-défini. Cette méthode a été initialement introduite par Merton [15].

5.2 Approche intensité de défaut

Les modèles à intensité se concentre sur la modélisation des probabilités de défaut en tant que processus stochastiques[16], contrairement à l'approche structurelle où le défaut est modélisé à partir de la valeur des actifs de l'entreprise. La modélisation du risque de défaut en utilisant des processus de taux de défaut et des variables aléatoires exogènes conduit à l'utilisation d'une filtration élargie [17] qui peut incorporer des informations sur les événements de défaut. On peut décrire l'événement de défaut comme le premier instant de saut τ d'un processus de Poisson avec une intensité déterministe ou stochastique λ qui satisfait l'équation suivante :

$$\lambda(t) := \mathbb{Q}(\tau \leq t + dt | \tau > t)\, dt$$

— Si λ est déterministe : $\mathbb{Q}(\tau > t) = \exp\left(-\int_0^t \lambda_u\, du\right)$

— Si λ est stochastique : $\mathbb{Q}(\tau > t | \mathcal{F}_t) = \exp\left(-\int_0^t \lambda_u\, du\right)$
où $(\lambda_t)_{t\in\mathbb{R}^+}$ est un processus aléatoire adapté à la filtration $(\mathcal{F}_t)_{t\in\mathbb{R}^+}$.

Le temps de défaut τ peut être défini comme :

$$\tau := \inf\left\{t \in \mathbb{R}^+ \,:\, \int_0^t \lambda_u\, du \geq L\right\},$$

où L est une variable aléatoire suivant une loi exponentielle de paramètre 1, indépendante de $(\mathcal{F}_t)_{t\in\mathbb{R}^+}$. En effet :

$$\begin{aligned}\mathbb{Q}\left(\tau > t \mid \mathcal{F}_t\right) &= \mathbb{Q}\left(\int_0^t \lambda_u du < L \mid \mathcal{F}_t\right) \\ &= \exp\left(-\int_0^t \lambda_u du\right) \text{ (par indépendance)}\end{aligned}$$

Nous utiliserons la filtration définie en ajoutant les informations sur le temps de défaut :

$$\mathcal{G}_t = \mathcal{F}_t \vee \sigma(\{\tau < u\} : 0 \leqslant u \leqslant t), t \in \mathbb{R}_+.$$

Lemme[18] : Pour toute variable aléatoire intégrable et $\mathcal{F}_T$-mesurable X :

$$\mathbb{E}\left[X\mathbf{1}_{\{\tau>T\}} \mid \mathcal{G}_t\right] = \mathbf{1}_{\{\tau>t\}}\mathbb{E}\left[Xe^{-\int_t^T \lambda_u du} \mid \mathcal{F}_t\right]$$

On définit un zéro-coupon risqué comme le contrat qui paie une unité de devise à T si le défaut n'a pas eu lieu avant la maturité. Pour calculer son prix, nous prenons $F = \exp\left(-\int_t^T r_u du\right)$ dans le Lemme précédent :

$$\begin{aligned} B_d(t,T) &= \mathbb{E}\left[\mathbf{1}_{\{\tau>T\}} \exp\left(-\int_t^T r_u du\right) \mid \mathcal{G}_t\right] \\ &= \mathbf{1}_{\{\tau>t\}} \mathbb{E}\left[\exp\left(-\int_t^T (r_u + \lambda_u)\, du\right) \mid \mathcal{F}_t\right] \end{aligned}$$

Si le processus λ_t est positif, nous pouvons constater que la présence d'un temps de défaut τ a pour effet d'augmenter le taux court de la quantité λ_u, ce qui conduit à une réduction du prix de l'obligation.

5.3 Credit Default Swaps

Les Credit Default Swaps (CDS) [19] [20] sont des produits dérivés de crédit qui peuvent être considérés comme des contrats d'assurance contre les dettes. Dans ce contrat, des paiements périodiques fixes sont effectués par l'acheteur de protection et échangés contre la promesse d'un paiement de la part du vendeur de protection si un événement de crédit préalablement spécifié se produit. Si l'événement survient avant l'échéance du CDS, le vendeur verse à l'acheteur un montant pour couvrir la perte et le contrat prend fin.

Le CDS peut être décomposé en deux jambes :

— La jambe de la prime : pendant la période (T_i, T_{i+1}), l'acheteur de la protection paie $c \cdot N \cdot (T_{i+1} - T_i)$ tant que le défaut ne s'est pas produit. N et c désignent respectivement le nominal et le taux de coupon.

— La jambe de protection (ou de défaut) : un instrument qui verse $(1 - \mathcal{R})$ si le défaut se produit et zéro sinon. $\mathcal{R}$ est appelé le taux de recouvrement et il est connu à l'avance. En cas de paiement de protection à l'échéance T, on parle de "protection à maturité", tandis que dans le second cas, avec un paiement survenant à τ, on parle de protection au moment du défaut (plus fréquente sur le marché).

Les valeurs stochastiques et actuelles de la La jambe de la prime :

$$SV_t(\mathcal{PL}) = \sum_{T_m \geq t} c \cdot N \cdot \delta T_m \cdot \mathbf{1}_{\{\boldsymbol{\tau} > T_m\}} \cdot e^{-\int_t^{T_m} r_u du}$$

$$
\begin{aligned}
PV_t(\mathcal{PL}) &= \mathbb{E}\left[\sum_{T_m \geq t} c \cdot N \cdot \delta T_m \cdot \mathbf{1}_{\{\boldsymbol{\tau} > T_m\}} \cdot e^{-\int_t^{T_m} r_u du} \mid \mathcal{G}_t\right] \\
&= \sum_{T_m \geq t} c \cdot N \cdot \delta T_m \cdot \mathbb{E}\left[\mathbf{1}_{\{\boldsymbol{\tau} > T_m\}} \cdot e^{-\int_t^{T_m} r_u du} \mid \mathcal{G}_t\right] \\
&= \sum_{T_m \geq t} c \cdot N \cdot \delta T_m \cdot \mathbf{1}_{\{\boldsymbol{\tau} > t\}} \cdot \mathbb{E}\left[e^{-\int_t^{T_m} (r_u + \lambda_u) du} \mid \mathcal{F}_t\right] \text{ (Lemme)} \\
&= c \cdot N \cdot \mathbf{1}_{\{\boldsymbol{\tau} > t\}} \cdot \sum_{T_m \geq t} \delta T_m \cdot B_d(t, T_m)
\end{aligned}
$$

Dans le cas d'une protection à maturité, la valeur de la jambe de protection est :

$$
SV_t(\mathcal{DL}) = (1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{t < \boldsymbol{\tau} \leq T\}} \cdot e^{-\int_t^T r_u du}
$$

Il s'ensuit que :

$$
\begin{aligned}
PV_t(\mathcal{DL}) &= \mathbb{E}\left[(1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{t < \boldsymbol{\tau} \leq T\}} \cdot e^{-\int_t^T r_u du} \mid \mathcal{G}_t\right] \\
&= (1 - \mathcal{R}) \cdot N \cdot \mathbb{E}\left[(\mathbf{1}_{\{\boldsymbol{\tau} > t\}} - \mathbf{1}_{\{\boldsymbol{\tau} > T\}}) \cdot e^{-\int_t^T r_u du} \mid \mathcal{G}_t\right] \\
&= (1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{\boldsymbol{\tau} > t\}} \cdot \mathbb{E}\left[e^{-\int_t^T r_u du} \cdot (1 - e^{-\int_t^T \lambda_u du}) \mid \mathcal{F}_t\right] \\
&\text{(en utilisant le Lemme et le fait que } \tau \text{ est un } \mathcal{G}_t\text{-temps d'arrêt)}
\end{aligned}
$$

Par la suite, on supposera toujours que la jambe de protection est exactement payée au moment du défaut τ :

$$
\begin{aligned}
PV_t(\mathcal{DL}) &= \mathbb{E}\left[(1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{t < \boldsymbol{\tau} \leq T\}} \cdot e^{-\int_t^\tau r_u du} \mid \mathcal{G}_t\right] \\
&= (1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{\boldsymbol{\tau} > t\}} \cdot \mathbb{E}\left[\int_t^T \lambda_s e^{-\int_t^s (r_u + \lambda_u) du} ds \mid \mathcal{F}_t\right] \\
&= (1 - \mathcal{R}) \cdot N \cdot \mathbf{1}_{\{\boldsymbol{\tau} > t\}} \cdot \int_t^T \mathbb{E}\left[\lambda_s e^{-\int_t^s (r_u + \lambda_u) du} \mid \mathcal{F}_t\right] ds
\end{aligned}
$$

Le spread du CDS *spd* est le taux de coupon équitable c de telle manière que la valeur initiale du contrat soit nulle, c'est-à-dire $PV_t(\mathcal{PL}) = PV_t(\mathcal{DL})$. Si $\tau > t$:

$$
spd = (1 - \mathcal{R}) \frac{\int_t^T \mathbb{E}\left[\lambda_s e^{-\int_t^s (r_u + \lambda_u) du} \mid \mathcal{F}_t\right] ds}{\sum_{T_m \geq t} \delta T_m \cdot \mathbb{E}\left[e^{-\int_t^{T_m} (r_u + \lambda_u) du} \mid \mathcal{F}_t\right]}
$$

On fixe $t = 0$. Soit ξ le vecteur de paramètres du modèle de diffusion du processus (r_t, λ_t). Pour estimer le spread , il suffit d'approximer, pour tout $s > 0$ les deux fonctions suivantes :

$$
F(\xi, s) = \mathbb{E}\left[e^{-\int_0^s (r_u + \lambda_u) du} \mid \xi, s\right] \text{ et } G(\xi, s) = \mathbb{E}\left[\lambda_s e^{-\int_0^s (r_u + \lambda_u) du} \mid \xi, s\right]
$$

5.4 Hypothèses de modélisation

5.4.1 Taux court

On utilise un modèle HJM à un seul facteur. Dans ce cadre, la diffusion de l'obligation zéro-coupon est la suivante :

$$\frac{dB(t,T)}{B(t,T)} = r_t dt - a(t)\left[\int_t^T b(s)ds\right] dW_t^1$$

Dans l'équation précédente, a et b sont des fonctions déterministes. On pose :

$$\begin{aligned}
\beta(t) &= \int_0^t b(s)ds \\
\alpha(t) &= \int_0^t a^2(s)ds \\
\gamma(t) &= \int_0^t a^2(s)\beta(s)ds \\
X_t &= \int_0^t a(s)dW_s^1
\end{aligned}$$

Le taux spot est alors donné par :

$$r_t = f(0,t) + b(t)(\beta(t)\alpha(t) - \gamma(t)) + b(t)X_t$$

Dans l'équation précédente, f désigne le taux forward instantané donné par :

$$B(t,T) = e^{-\int_t^T f(t,s)ds}$$

5.4.2 Intensité de défaut

L'intensité par défaut est définie par le modèle lognormal de Black-Karasinski[21]. Nous considérons le processus auxiliaire d'Ornstein-Uhlenbeck suivant :

$$\begin{cases} dY_t = -\alpha_\lambda Y_t dt + \sigma_\lambda(t)dW_t^2 \\ Y_0 = 0 \end{cases}$$

α_λ est un paramètre constant représentant la vitesse de retour à la moyenne, et la volatilité σ_λ est une fonction déterministe. L'intensité de défaut λ est liée à Y_t de la manière suivante :

$$\lambda_t = (\bar{\lambda}(t) + \lambda^*(t))\mathcal{E}(Y_t)$$

$\bar{\lambda}(t)$ et $\lambda^*(t)$ sont deux fonctions déterministes à calibrer et $\mathcal{E}(Y_t)$ est la semimartingale exponentielle : $\mathcal{E}(Y_t) = \exp(Y_t - \frac{1}{2} < Y >_t)$. L'avantage d'un modèle lognormal est de garantir une intensité de défaut positive. Nous modélisons la corrélation entre l'intensité de défaut et le taux court en utilisant une corrélation constante entre les mouvements browniens W_t^1 et W_t^2 :

$$< W_t^1, W_t^2 >= \rho t$$

5.5 Simulateur de trajectoires

5.5.1 Implémentation

Le schéma classique d'Euler a été utilisé pour discrétiser les EDS. On suppose que toutes les fonctions déterministes dans la dynamique de l'intensité et des taux courts sont constantes par morceaux. Les expressions de β, α et γ ont été obtenues à l'aide de SymPy, une bibliothèque Python open-source spécialisée dans les calculs symboliques, permettant de manipuler des expressions mathématiques symboliquement, effectuer des dérivations, des intégrations, des simplifications et résoudre des équations symboliques.

Ici, le vecteur de paramètres ξ représente les valeurs définissant les fonctions constantes par morceaux a, b, λ^*, $\bar{\lambda}$ et σ_λ, ainsi que α_λ et ρ. Le défi était de pouvoir générer en parallèle plusieurs trajectoires pour de multiples réalisations des paramètres ξ et T. Cela a été accompli en vectorisant les équations des schémas d'Euler permettant de diffuser les processus. On a également défini des noyaux personnalisés pour manipuler toutes les fonctions en morceaux. Pour ce faire, on a converti le code SymPy en code CUDA et utilisé cupy.RawKernel. Plutôt que d'utiliser une boucle for de manière naïve, en adoptant une approche parallèle pour gérer plusieurs réalisations des paramètres ξ et T, nous avons réussi à réduire le temps de calculs pour environ 67 millions de trajectoires de quelques heures à seulement quelques secondes.

Algorithm 2 Piecewise Constant Kernel

Input : array, output, threshs, values, num_segments, input_width
Output : output
col_idx ← blockIdx.x × blockDim.x + threadIdx.x
row_idx ← blockIdx.y
idx ← row_idx × input_width + col_idx ▷ index within the array
x ← array[idx] ▷ input value at index idx
for *i ← 0 **to** num_segments - 1* **do**
  if *x < threshs[i]* **then**
    output[idx] ← values[row_idx × num_segments + i]
    break
  end
end

Supposons que nous ayons N_{batch} réalisations i.i.d de (ξ, T). Dans le cas simple où nous souhaitons évaluer simultanément chaque réalisation a_i (a_i est un tableau de taille num_segments) sur N_{euler} points dans $[0, T_i]$, nous utilisons **Algorithm 1** avec les variables suivantes :

TABLE 2 – Variables utilisées dans le noyau à morceaux constants

Variable	Description	Taille
array	Instants à évaluer	$N_{batch} \times N_{euler}$
input_width	#Points de discrétisation d'Euler	N_{euler}
output	Stocke le résultat	$N_{batch} \times N_{euler}$
values	Stocke les valeurs a_i	$N_{batch} \times num\_segments$

Nous avons utilisé une grille GPU en 2D, où les variables col_idx et row_idx déterminent la position unique de chaque thread. Chaque bloc le long de l'axe y de la grille est associé à une réalisation des paramètres.

L'algorithme précédent a été utilisé pour évaluer a, b, λ^*, $\bar{\lambda}$ et σ_λ. Pour β, α et γ, nous avons exploité SymPy pour générer automatiquement du code C exprimant ces fonctions sur chaque segment. Ensuite, nous avons légèrement ajusté ce code en utilisant une indexation similaire à celle de **Algorithm 1** pour obtenir un code CUDA.

TABLE 3 – Principales fonctions utilisées pour simuler les trajectoires du taux court et de l'intensité de défaut.

Fonction	Description	Taille
brownian_2d	Génère des incréments browniens corrélés pour diffuser r_t et λ_t.	$2 \times N_{batch} \times N_{red} \times N_{euler}$
generate_x_t_paths	Évalue a et génère les trajectoires du facteur HJM X_t.	$N_{batch} \times N_{red} \times (N_{euler} + 1)$
generate_r_t_paths	Évalue b, β, α et γ et génère les trajectoires de r_t.	$N_{batch} \times N_{red} \times (N_{euler} + 1)$
generate_y_t_paths	Évalue σ_λ et génère les trajectoires de Y_t.	$N_{batch} \times N_{red} \times (N_{euler} + 1)$
generate_lambda_t_paths	Évalue $< Y >$, λ et λ^* et génère les trajectoires de λ_t.	$N_{batch} \times N_{red} \times (N_{euler} + 1)$

TABLE 4 – Distributions des paramètres et de la maturité pour les simulations. Pour les fonctions, la loi concerne les constantes qui les définissent sur les différents segments.

variable	Loi
T	$\mathcal{U}(0.1, 6)$
a	$\mathcal{U}(-0.1, 0.1)$
b	$\mathcal{U}(-0.1, 0.1)$
f	$\mathcal{U}(0, 0.1)$
λ	$\mathcal{U}(0, 0.1)$
λ^*	$\mathcal{U}(0, 0.1)$
σ_λ	$\mathcal{U}(0.01, 0.4)$
α_λ	$\mathcal{U}(-0.3, 0.3)$
ρ	$\mathcal{U}(-0.9, 0.9)$

5.5.2 Premiers tests

Pour tester le simulateur de trajectoires, nous avons calculé l'espérance et la variance théoriques des processus simulés et les avons comparées à celles empiriques.

X_t est une martingale locale avec $\mathbb{E}[< X >_t] = \int_0^t a^2(s)ds < \infty$ pour tout t donc c'est une vrai martingale. En particulier $\mathbb{E}[X_t] = 0$ pour tout t. De plus, l'isométrie d'Itô nous donne :$\mathbb{V}ar(X_t) = \int_0^t a^2(s)ds$.

On applique la formule d'itô à la fonction $f(t, Y_t) = e^{\alpha_\lambda t} Y_t$:

$$\begin{aligned} d(e^{\alpha_\lambda t} Y_t) &= \alpha_\lambda e^{\alpha_\lambda t} Y_t dt + e^{\alpha_\lambda t} dY_t \\ &= e^{\alpha_\lambda t} dW_t^2 \\ Y_t = Y_0 \cdot e^{-\alpha_\lambda t} &+ \int_0^t e^{-\alpha_\lambda (t-s)} \sigma_\lambda(s) dW_s^2 \end{aligned}$$

$$= \int_0^t e^{-\alpha_\lambda (t-s)} \sigma_\lambda(s) dW_s^2$$

Ainsi, Y_t est un processus gaussien avec : $\begin{cases} \mathbb{E}[Y_t] = 0 \\ \mathbb{V}ar(Y_t) = \int_0^t \sigma_\lambda^2(s) \cdot e^{-2\alpha_\lambda (t-s)} ds \end{cases}$

En utilisant la fonction génératrice des moments pour la loi normale, on obtient :
$\mathbb{E}[\mathcal{E}(Y_t)] = e^{-\frac{1}{2}\int_0^t \sigma_\lambda^2(s)ds} \mathbb{E}[e^{Y_t}] = e^{-\frac{1}{2}\int_0^t \sigma_\lambda^2(s)ds + \mathbb{E}[Y_t] + \frac{\mathbb{V}ar(Y_t)}{2}}$
$\mathbb{E}[\mathcal{E}(Y_t)^2] = e^{-\int_0^t \sigma_\lambda^2(s)ds} \mathbb{E}[e^{2Y_t}] = e^{-\int_0^t \sigma_\lambda^2(s)ds + 2\mathbb{E}[Y_t] + 2\mathbb{V}ar(Y_t)}$

5.6 Régression neuronale

Nous rappelons que notre objectif est d'approximer les deux espérances conditionnelles suivantes à l'aide de la méthode introduite dans la section 4.2 :

$$F(\xi, s) = \mathbb{E}\left[e^{-\int_0^s (r_u + \lambda_u) du} \mid \xi, s\right] \quad \text{et} \quad G(\xi, s) = \mathbb{E}\left[\lambda_s e^{-\int_0^s (r_u + \lambda_u) du} \mid \xi, s\right]$$

Nous avons choisi d'utiliser le même réseau pour les deux fonctions. Par conséquent, la sortie de notre modèle est bidimensionnelle. L'entrée du réseau consiste en N_{train} réalisations i.i.d des paramètres du modèle de pricing et de la maturité : (ξ_i, s_i), et les sorties cibles sont les réalisations correspondantes de $e^{\frac{s_i}{N_{eu}} \sum_{k=1}^{N_{eul}} (r_{i,k} + \lambda_{i,k})}$ et $\lambda_{i,N_{eul}} e^{\frac{s_i}{N_{eu}} \sum_{k=1}^{N_{eul}} (r_{i,k} + \lambda_{i,k})}$, où N_{eul} est le nombre de pas de discrétisation dans le schéma d'Euler. Après l'entraînement, l'approximation du spread du CDS est donnée par :

$$\hat{spd}(\xi, T, \{T_m\}) = (1 - \mathcal{R}) \frac{\frac{T}{N_{int}} \sum_{k=1}^{N_{int}} \hat{F}(\xi, \frac{kT}{N_{int}})}{\sum_{T_m \geq 0} \delta T_m \hat{G}(\xi, T_m)}$$

, où nous avons utilisé $\frac{T}{N_{int}} \sum_{k=1}^{N_{int}} \hat{F}(\xi, \frac{kT}{N_{int}})$ pour estimer $\int_0^T \hat{F}(\xi, s)ds$

TABLE 5 – Paramètres utilisés pour les simulations

Paramètre	valeur
N_{eul}	100
N_{int}	100
N_{train}	2^{21}
N_{red}	32
fraction de validation	0.1

5.6.1 Choix des hyperparamètres

Les hyperparamètres du réseau ont été sélectionnés à l'aide d'un Grid Search, où nous avons fait varier le nombre de neurones dans les couches cachées et la taille du mini-batch, tout en mesurant les performances sur l'ensemble de validation. Nous avons uniquement considéré des réseaux composés de deux couches cachées et avons choisi ADAM[9] comme optimiseur avec un taux d'apprentissage de 0.0001.

Ci-dessous, les résultats pour trois configurations différentes. Dans la première configuration, nous avons supposé que toutes les fonctions présentes dans les équations de diffusion des processus étaient constantes, et nous avons entraîné les modèles pendant 10 époques. Pour les deux autres configurations, nous avons plutôt envisagé des fonctions constantes par morceaux et entraîné les modèles pendant 100 époques. Les coefficients de détermination r_2^1 et r_2^2 correspondent respectivement aux approximations de F et G. Enfin, nous avons retenu les réseaux ayant les valeurs du loss les plus faibles sur l'ensemble de validation.

Hidden Size	**Batch Size**	**Loss**	r_2^1	r_2^2
512	256	0.0344	0.993	0.938
512	512	0.0348	0.9931	0.9371
256	256	0.0353	0.9928	0.9364
256	512	0.0367	0.9922	0.9343
128	256	0.0371	0.9923	0.9332
128	512	0.0454	0.9891	0.9198

TABLE 6 – Performance avec des fonctions constantes et des réseaux neuronaux sans différenciation automatique randomisée.

Hidden Size	**Batch Size**	**Loss**	r_2^1	r_2^2
512	512	0.0469	0.979	0.9272
512	256	0.048	0.979	0.9251
256	512	0.0492	0.9783	0.9234
256	256	0.0494	0.9783	0.9231
128	512	0.0542	0.9768	0.9148
128	256	0.0564	0.9765	0.9108

TABLE 7 – Performance avec des fonctions constantes par morceaux et des réseaux neuronaux sans différenciation automatique randomisée.

Hidden Size	**Batch Size**	**Loss**	r_2^1	r_2^2
512	256	0.0505	0.9789	0.9203
512	512	0.0506	0.9787	0.9201
256	256	0.0513	0.9785	0.9189
128	256	0.0542	0.978	0.9138
256	512	0.0554	0.9781	0.9112
128	512	0.0576	0.9775	0.9075

TABLE 8 – Performance en utilisant des fonctions constantes par morceaux et une différenciation automatique randomisée avec la méthode « même activation ».

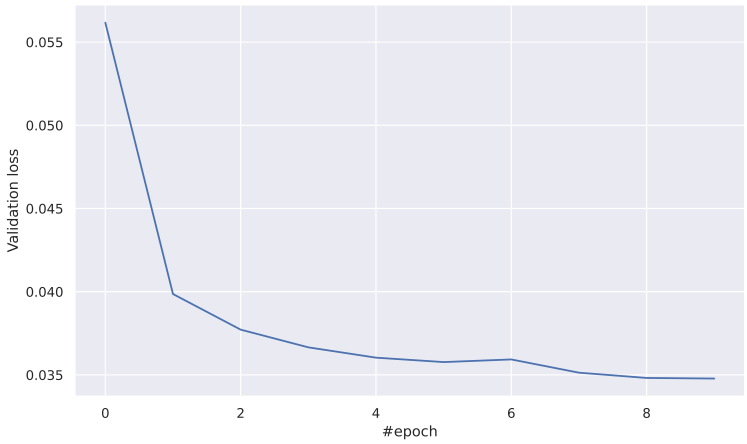


FIGURE 28 – Courbe d'entraînement du meilleur modèle avec des fonctions constantes. Temps d'entraînement : 35s

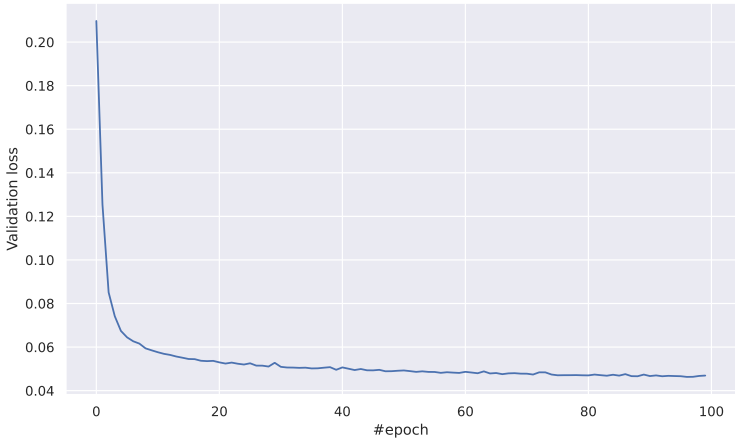


FIGURE 29 – Courbe d'entraînement du meilleur modèle avec des fonctions constantes par morceaux. Temps d'entraînement : 5min33s

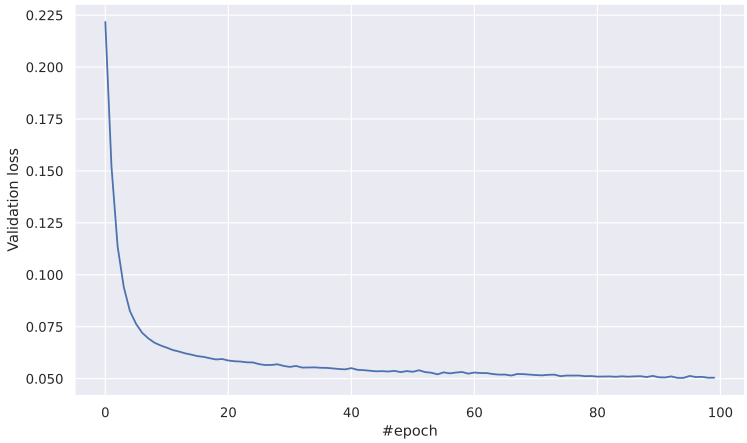


FIGURE 30 – Courbe d'entraînement du meilleur modèle avec des fonctions constantes par morceaux et de la différenciation automatique randomisée. Temps d'entraînement : 6min8s

5.6.2 Monte Carlo imbriqué

Comme nous n'avons pas de formule explicite pour les espérances conditionnelles comme dans le cas simple de Black-Scholes, nous pouvons évaluer les estimateurs de régression à travers une méthode de Monte Carlo imbriqué. Pour ce faire, nous générons N_{test} échantillons de (ξ, T) et pour chaque réalisation (ξ_i, s_i), nous simulons N_{MC} trajectoires et calculons les estimateurs de Monte Carlo correspondants :

$$\begin{cases} \hat{F}_{\text{MC},i} = \frac{1}{N_{\text{MC}}} \sum_{j=1}^{N_{\text{MC}}} \left(e^{\frac{s_i^j}{N_{\text{eu}}} \sum_{k=1}^{N_{\text{eul}}} (r_{i,k}^j + \lambda_{i,k}^j)} \right) \\ \hat{G}_{\text{MC},i} = \frac{1}{N_{\text{MC}}} \sum_{j=1}^{N_{\text{MC}}} \lambda_{i,N_{\text{eul}}}^j \left(e^{\frac{s_i^j}{N_{\text{eu}}} \sum_{k=1}^{N_{\text{eul}}} (r_{i,k}^j + \lambda_{i,k}^j)} \right) \end{cases}$$

En raison de limitations de mémoire, nous ne pouvons pas effectuer une parallélisation efficace. Par conséquent, nous avons opté pour un traitement séquentiel des échantillons de l'ensemble de données, ce qui peut entraîner une lenteur significative. Enfin, nous pouvons comparer les résultats des deux méthodes en utilisant les scores suivants :

$$\begin{cases} \text{MAPE1} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \frac{|\hat{F}_{\text{MC},i} - \hat{F}(s_i, \xi_i)|}{\hat{F}_{\text{MC},i}} \\ \text{MAPE2} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \frac{|\hat{G}_{\text{MC},i} - \hat{G}(s_i, \xi_i)|}{\hat{G}_{\text{MC},i}} \end{cases}$$

	MAPE1	**MAPE2**
Fonctions constantes	0.3943%	1.7034%
Fonctions constantes par morceaux	2.3059	6.0531%
Fonctions constantes par morceaux + RAD	2.3041%	7.9674%

TABLE 9 – Comparaison entre les estimations de Monte-Carlo et les prédictions du modèle avec $N_{MC} = 2^{19}$ et $N_{test} = 2^{14}$.

5.7 Calibration

Nous supposons que le paiement de la prime est effectué trimestriellement, c'est-à-dire $\{T_m\} = \{\frac{kT}{4}\}_{k=1,\dots,4}$ et que le taux de récupération est $\mathcal{R} = 0.4$.

Étant donné un ensemble de données de spreads de marché CDS : $\{\text{spd}^{\text{mkt}}(\tau_k)\}_{k=1,\dots,N_{\text{calib}}}$, nous calibrons le modèle en résolvant le problème d'optimisation suivant :

$$\arg\min_{\xi} \mathcal{L}_{calib}(\xi) = \arg\min_{\xi} \sum_{k=1}^{N_{calib}} (\hat{spd}(\xi, \tau_k) - spd^{mkt}(\tau_k))^2$$

Nous appliquons deux fois l'algorithme de rétropropagations de la section 4.3 pour obtenir $\frac{\partial \hat{F}}{\partial \xi}$ et $\frac{\partial \hat{G}}{\partial \xi}$. Ensuite, nous déduisons le gradient du spread par rapport à ξ :

$$\begin{aligned}\frac{1}{(1-\mathcal{R})} \cdot \frac{\partial \hat{spd}}{\partial \xi} = & \frac{(\sum_{T_m \geq 0} \delta T_m \hat{G}(\xi, T_m)) \cdot (\frac{T}{N_{int}} \sum_{k=1}^{N_{int}} \frac{\partial \hat{F}}{\partial \xi}(\xi, \frac{kT}{N_{int}}))}{(\sum_{T_m \geq 0} \delta T_m \hat{G}(\xi, T_m))^2} \\ & - \frac{(\frac{T}{N_{int}} \sum_{k=1}^{N_{int}} \hat{F}(\xi, \frac{kT}{N_{int}})) \cdot (\sum_{T_m \geq 0} \delta T_m \frac{\partial \hat{G}}{\partial \xi}(\xi, T_m))}{(\sum_{T_m \geq 0} \delta T_m \hat{G}(\xi, T_m))^2}\end{aligned}$$

Finalement, nous pouvons effectuer une descente de gradient sur l'ensemble des données pour trouver un minimum local de $\mathcal{L}_{\text{calib}}$ en utilisant le fait que :

$$\frac{\partial \mathcal{L}_{calib}}{\partial \xi} = \sum_{k=1}^{N_{calib}} \frac{\partial \hat{spd}}{\partial \xi}(\xi, \tau_k) \cdot (\hat{spd}(\xi, \tau_k) - spd^{mkt}(\tau_k))$$

Pour les trois configurations différentes, nous avons utilisé ADAM pour minimiser $\mathcal{L}_{calib}$ avec un pas d'apprentissage égal à $5 \cdot 10^{-3}$.

TABLE 10 – Jeu de données utilisé pour la calibration

Maturité	Spread
0.5	0.003256
1	0.004998
2	0.008716
3	0.011902
4	0.014006
5	0.015816

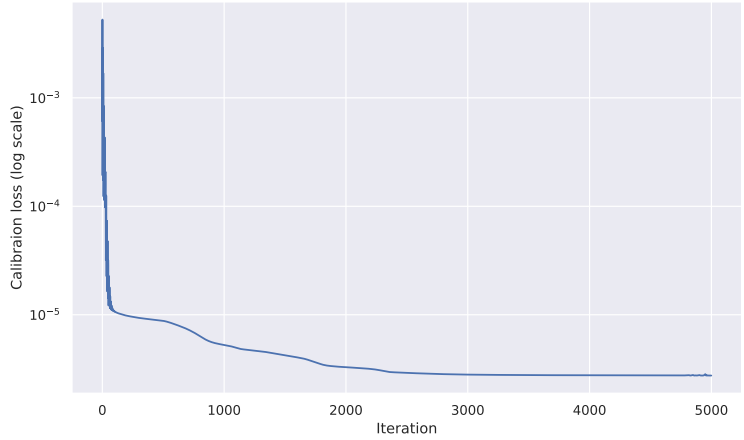


FIGURE 31 – Évolution de la perte $\mathcal{L}_{calib}$ au long des itérations dans le cas des fonctions constantes. La perte stagne aux alentours de 10^{-6}. Paramètres calibrés : $\rho = 0.1174$, $\alpha_\lambda = 0.2336$, $a(t) = -0.7244$, $b(t) = -0.7639$, $\sigma(t) = 0.6563$, $\bar{\lambda}(t) = 0.0900$ et $\lambda^*(t) = 0.0899$.

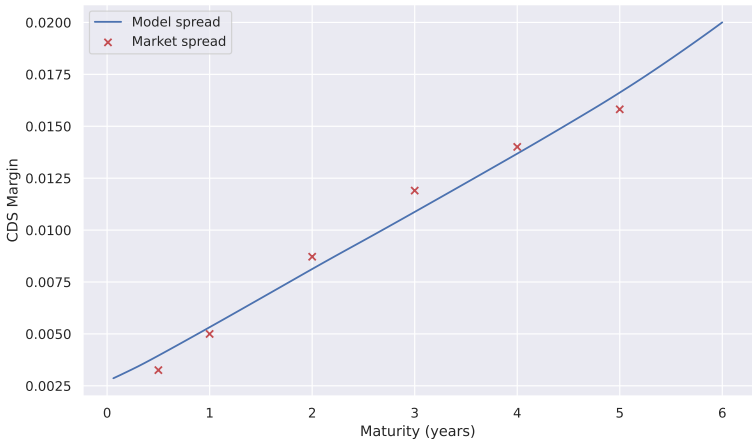


FIGURE 32 – Spreads estimés dans le cas des fonctions constantes. Le réseau de neurones calibré semble être linéaire par rapport à la maturité.

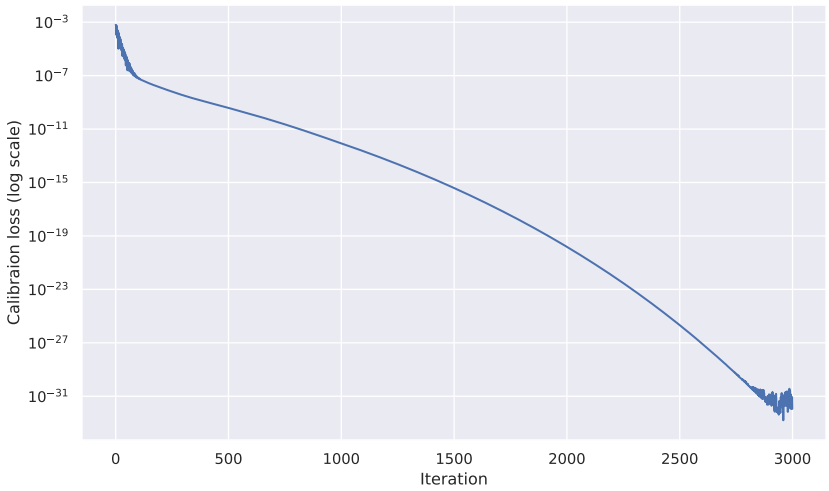


FIGURE 33 – Évolution de la perte $\mathcal{L}_{calib}$ au long des itérations dans le cas des fonctions constantes par morceaux. La perte converge vers une valeur très faible.

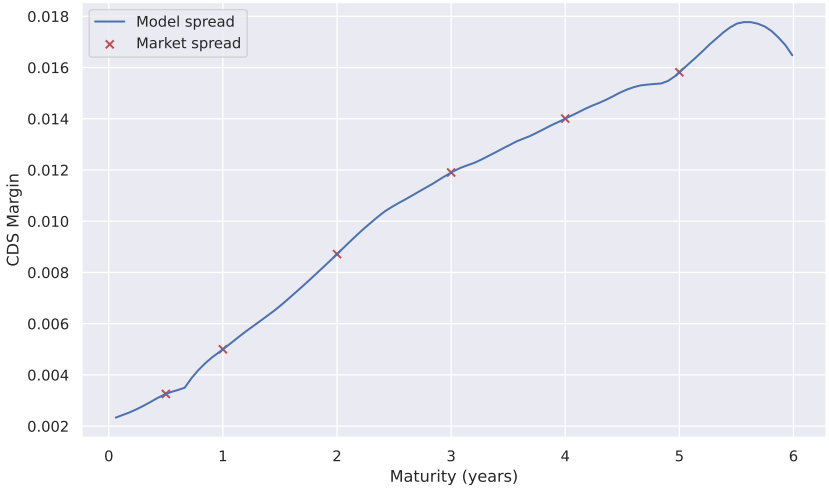


FIGURE 34 – Spreads estimés dans le cas des fonctions constantes par morceaux. Le réseau de neurones calibré interpole parfaitement les spreads du marché.

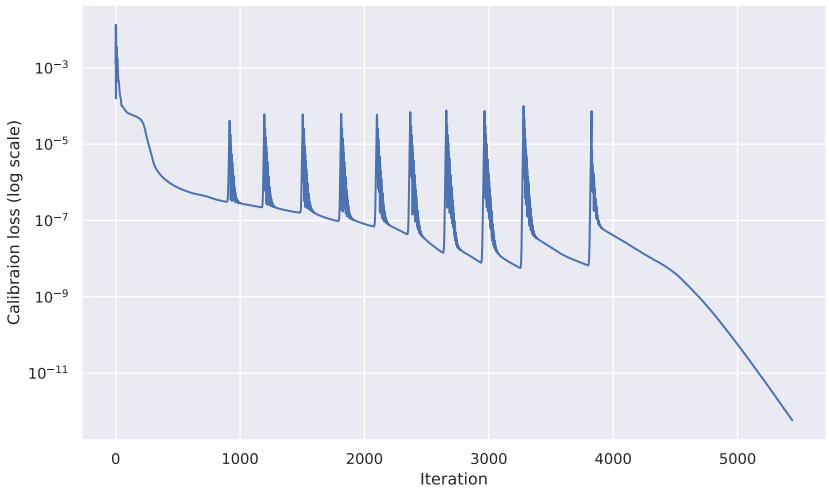


FIGURE 35 – Évolution de la perte $\mathcal{L}_{calib}$ au long des itérations dans le cas des fonctions constantes par morceaux, en utilisant la différenciation automatique randomisée lors de l'entraînement du réseau. La perte oscille avant de converger vers une valeur de l'ordre de 10^{-12}.

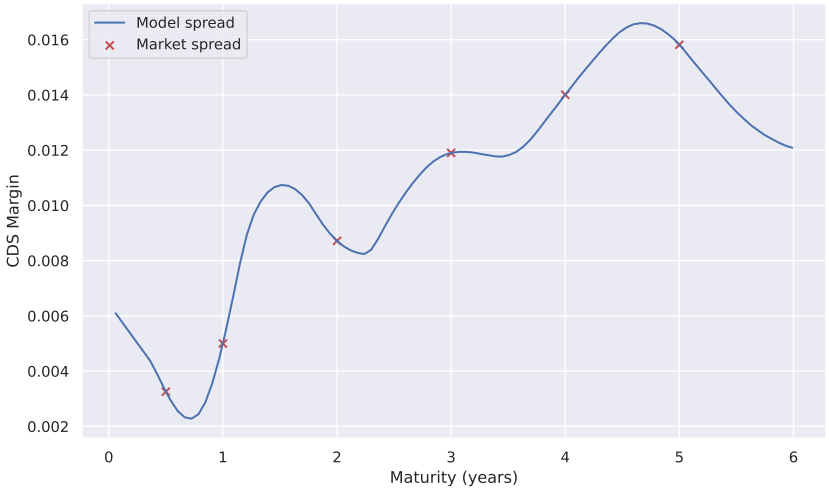
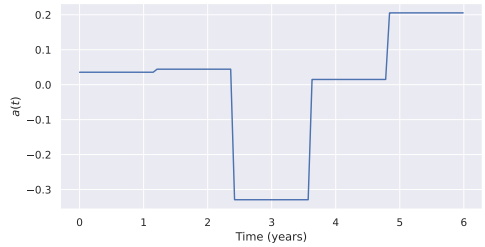
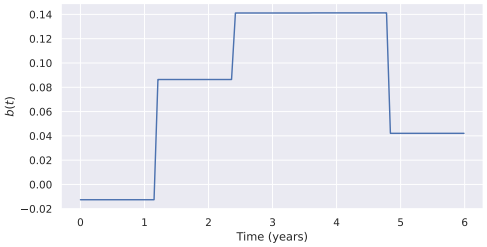


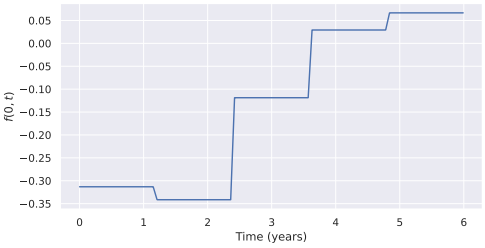
FIGURE 36 – Spreads estimés dans le cas des fonctions constantes par morceaux et avec la différenciation automatique randomisée. Le réseau calibré parvient à reproduire parfaitement les spreads du marché mais présente une variance plus élevée que dans le cas sans RAD.



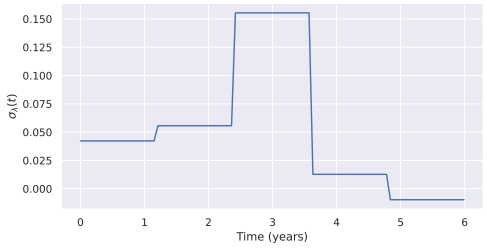
(a) $a(t)$



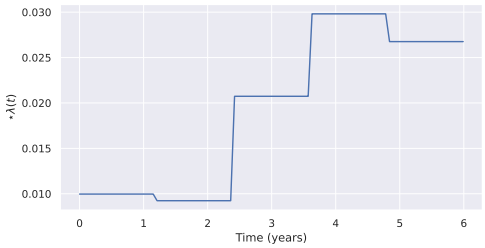
(b) $b(t)$



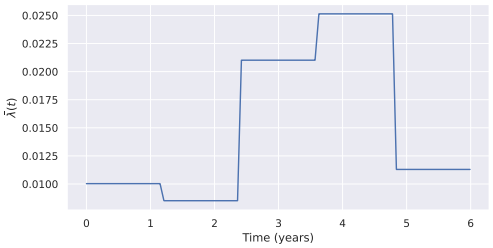
(c) $f(0,t)$



(d) $\sigma_\lambda(t)$

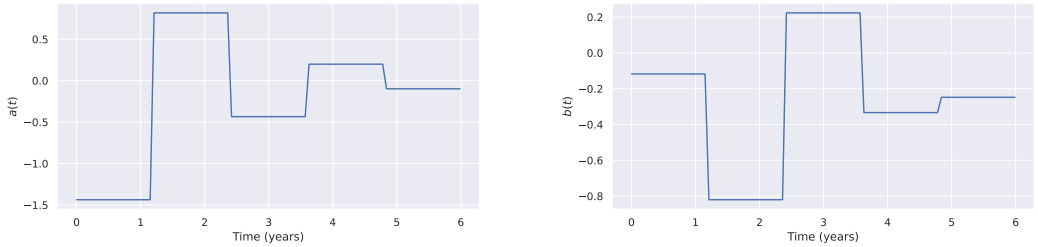


(e) $\lambda^*(t)$



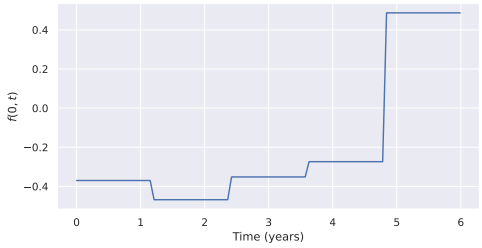
(f) $\bar{\lambda}$

FIGURE 37 – Fonctions calibrées dans le cas d'un entraînement sans RAD. Nous avons considérés cinq constantes sur l'intervalle [0, 6 ans]. Les deux autres valeurs scalaires calibrées sont $rho = 0.03323$ et $\alpha_\lambda = 0.2524$.

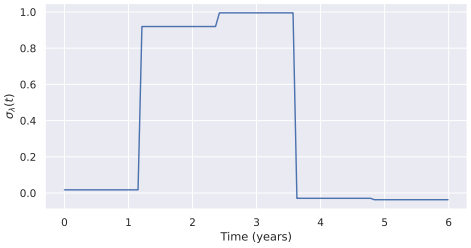


(a) $a(t)$

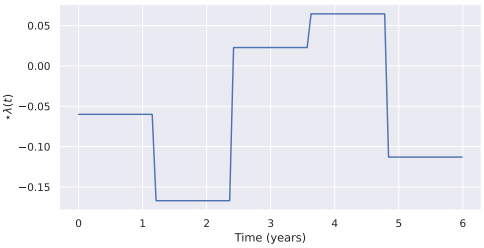
(b) $b(t)$



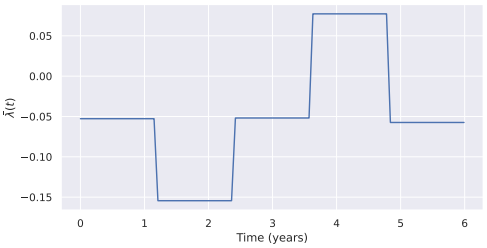
(c) $f(0,t)$



(d) $\sigma_\lambda(t)$



(e) $\lambda^*(t)$



(f) $\bar{\lambda}$

FIGURE 38 – Fonctions calibrées dans le cas d'un entraînement avec RAD. Les deux autres valeurs scalaires calibrées sont $rho = 0.05471$ et $\alpha_\lambda = -0.09102$.

	Temps de calibration	**#itération**	**MAPE**
Fonctions constantes	7min27s	5000	8.46%
Fonctions constantes par morceaux	4min37s	3000	$1.06 \cdot 10^{-12}$ %
Fonctions constantes par morceaux + RAD	8min47s	5500	$1.02 \cdot 10^{-4}$ %

TABLE 11 – Temps de calibration, nombre d'itérations et erreur relative moyenne par rapport aux spreads de marché.

5.7.1 Changement de numéraire

On pose $\sigma_B(t,T) = -a(t)\int_t^T b(s)ds$. Sous $\mathbb{Q}$:

$$\begin{cases} \frac{dB(t,T)}{B(t,T)} = r_t dt + \sigma_B(t,T)dB_t^1 \\ dY_t = -\alpha_\lambda Y_t dt + \sigma_\lambda(t)(\rho dB_t^1 + \sqrt{1-\rho^2}dB_t^2) \end{cases}$$

On considère le changement de probabilités :

$$\frac{d\mathbb{Q}^T}{d\mathbb{Q}}|\mathcal{F}_t = e^{-\int_0^t r_s ds}\frac{B(t,T)}{B(0,t)}$$

$$\begin{cases} dB_t^{T,1} = dB_t^1 - \sigma_B(t,T)dt \\ dB_t^{T,2} = dB_t^2 \end{cases}$$

Sous $\mathbb{Q}^T$:

$$\begin{aligned} dY_t &= \left[-\alpha_\lambda Y_t + \rho\cdot\sigma_\lambda(t)\cdot\sigma_B(t,T)\right]dt + \sigma_\lambda(t)\left[\rho dB_t^{T,1} + \sqrt{1-\rho^2}dB_t^{T,2}\right] \\ &= \left[-\alpha_\lambda Y_t + \rho\cdot\sigma_\lambda(t)\cdot\sigma_B(t,T)\right]dt + \sigma_\lambda(t)d\tilde{W}_t^T \\ &= d\tilde{Y}_t + \rho\cdot\sigma_\lambda(t)\cdot\sigma_B(t,T)dt \end{aligned}$$

L'intensité de défaut est :

$$\begin{aligned} \lambda(t) =& (\bar{\lambda}(t) + \lambda^*(t))e^{Y_t - \frac{1}{2}<Y>_t} \\ &= \underbrace{e^{\rho\int_0^t \sigma_\lambda(u)\cdot\sigma_B(u,T)du}}_{\eta(t)}\cdot\left[(\bar{\lambda}(t)+\lambda^*(t))\cdot e^{\tilde{Y}_t - \frac{1}{2}<\tilde{Y}_t>}\right] \end{aligned}$$

On fixe t=0. Le spread est :

$$\begin{aligned} spd &= (1-\mathcal{R})\frac{\int_0^T \mathbb{E}^{\mathbb{Q}}\left[\lambda_s e^{-\int_0^s (r_u+\lambda_u)du}\right]ds}{\sum_{T_m\geq t}\delta T_m\cdot\mathbb{E}^{\mathbb{Q}}\left[e^{-\int_0^{T_m}(r_u+\lambda_u)du}\right]} \\ &= (1-\mathcal{R})\frac{\int_0^T \mathbb{E}^{\mathbb{Q}^s}\left[\lambda_s e^{-\int_0^s \lambda_u du}\right]B(0,s)ds}{\sum_{T_m\geq t}\delta T_m\cdot\mathbb{E}^{\mathbb{Q}^{T_m}}\left[e^{-\int_0^{T_m}\lambda_u du}\right]B(0,T_m)} \end{aligned}$$

Pour approximer le CDS après ce changement de numéraire, il suffit d'avoir accès aux prix zéro-coupons $B(0,s)$ pour tout $s\in[0,T]$ et d'approximer par régression :

$$\begin{cases} H(\xi,s) = \mathbb{E}^{\mathbb{Q}^s}\left[e^{-\int_0^s\lambda_u du}\mid \xi, s\right] \\ J(\xi,s) = \mathbb{E}^{\mathbb{Q}^s}\left[\lambda_s e^{-\int_0^s\lambda_u du}\mid \xi, s\right] \end{cases}$$

Les zéros-coupons sont donnés par :

$$B(0,s) = e^{-\int_0^s f(0,u)du}$$

Du point de vue de la simulation, on doit diffuser qu'un seul processus au lieu de deux. Du point de vue de l'implémentation, il suffit de définir un noyau `CuPy` pour évaluer η.

La taille du vecteur de paramètres ξ diminue, car la dynamique de λ_t ne dépend pas du taux forward initial $f(0,t)$. On pose :

$$\begin{cases} H_B(\xi,s) = H(\xi,s)\cdot B(0,s) \\ J_B(\xi,s) = J(\xi,s)\cdot B(0,s) \end{cases}$$

Après l'entraînement du réseau de neurones, le calcul du gradient du spread par rapport aux paramètres ξ est similaire au cas précédent :

$$\frac{1}{(1-\mathcal{R})}\cdot\frac{\partial\widehat{spd}}{\partial\xi}=\frac{\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))\cdot(\frac{T}{N_{int}}\sum_{k=1}^{N_{int}}\frac{\partial\widehat{H_B}}{\partial\xi}(\xi,\frac{kT}{N_{int}}))}{(\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))^2}$$
$$-\frac{(\frac{T}{N_{int}}\sum_{k=1}^{N_{int}}\widehat{H_B}(\xi,\frac{kT}{N_{int}}))\cdot(\sum_{T_m\geq 0}\delta T_m\frac{\partial\widehat{J_B}}{\partial\xi}(\xi,T_m))}{(\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))^2}$$

Les constantes de la courbe forward f_i interviennent dans le calcul du spread via les zéros-coupons. Par conséquent, pour la calibration, on doit aussi calculer :

$$\frac{\partial B(0,s)}{\partial f_i}=-B(0,s)\cdot\begin{cases}\frac{T_{\max}}{N_{\text{nodes}}-1}\text{ si }s<s_{i-1}\\ s-s_i\text{ si }s_{i-1}<s<s_i\\ 0\text{ sinon}\end{cases}$$

$$\begin{cases}\frac{\partial\widehat{H_B}}{\partial f_i}(\xi,s)=H(\xi,s)\cdot\frac{\partial B(0,s)}{\partial f_i}\\ \frac{\partial\widehat{J_B}}{\partial f_i}(\xi,s)=J(\xi,s)\cdot\frac{\partial B(0,s)}{\partial f_i}\end{cases}$$

$$\frac{1}{(1-\mathcal{R})}\cdot\frac{\partial\widehat{spd}}{\partial f_i}=\frac{\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))\cdot(\frac{T}{N_{int}}\sum_{k=1}^{N_{int}}\frac{\partial\widehat{H_B}}{\partial f_i}(\xi,\frac{kT}{N_{int}}))}{(\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))^2}$$
$$-\frac{(\frac{T}{N_{int}}\sum_{k=1}^{N_{int}}\widehat{H_B}(\xi,\frac{kT}{N_{int}}))\cdot(\sum_{T_m\geq 0}\delta T_m\frac{\partial\widehat{J_B}}{\partial f_i}(\xi,T_m))}{(\sum_{T_m\geq 0}\delta T_m\widehat{J_B}(\xi,T_m))^2}$$

6 Conclusion

Dans les premières parties du rapport, nous avons examiné en détail la méthode de "Différenciation Automatique Randomisée" dans le contexte de l'algorithme de rétropropagation utilisé pour les réseaux de neurones. Nous avons constaté que, sous réserve d'une implémentation optimale en termes de gestion de la mémoire, nous pouvons réduire de manière significative la quantité de RAM nécessaire pour stocker les activations lors du pass forward. Bien que cette méthode introduise une variance supplémentaire dans les estimations des gradients et qu'elle ralentisse légèrement la vitesse de convergence lors de l'entraînement, son impact sur les performances finales demeure minime.

Par la suite, nous avons étudié l'approximation des espérances conditionnelles à l'aide de réseaux de neurones. Cette approche présente plusieurs avantages. Tout d'abord, elle permet d'obtenir une "formule fermée" après l'entraînement, car l'inférence dans le cas d'un réseau de neurones standard est pratiquement instantanée. Deuxièmement, la différentiation automatique nous permet également d'obtenir le gradient de la fonction approximée par rapport aux entrées. Dans le cas où la fonction à approximer représente le prix d'un contrat financier, ces gradients représentent les sensibilités. Nous avons observé qu'avec une implémentation efficace du simulateur de trajectoires, il est possible d'approximer les spreads des CDS et de calibrer le modèle sur les données de marché en un temps raisonnable.

Références

[1] Joshua Aduol Alex Beatson Ryan P. Adams Deniz Oktay, Nick McGreivy. Randomized automatic differentiation. 2021.

[2] Alexey Andreyevich Radul Jeffrey Mark Siskind Atilim Gunes Baydin, Barak A. Pearlmutter. Automatic differentiation in machine learning : a survey. 2018.

[3] F. L. Bauer. Computational graphs and rounding error. 1974.

[4] Sutton Monro Herbert Robbins. A stochastic approximation method. 1951.

[5] Gilles Pagès. Numerical probability : an introduction with applications to finance. 2018.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Olivier Grisel Charles Ollion. Neural networks and backpropagation. 2022. URL https://m2dsupsdlclass.github.io/lectures-labs/slides/02_backprop/index.html#1.

[8] Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. 2010.

[9] Jimmy Ba Diederik P. Kingma. Adam : A method for stochastic optimization. 2014.

[10] Matthew D. Zeiler. Adadelta : An adaptive learning rate method. 2012.

[11] Geoffrey Hinton. Neural networks for machine learning. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[12] Ning Qian. On the momentum term in gradient descent learning algorithms. 1999.

[13] Terry Lyons Patrick Kidger. Universal approximation with deep narrow networks. 2020.

[14] SAADEDDINE Bouazza. Fast calibration using complex-step sobolev training. 2022.

[15] Robert C. Merton. On the pricing of corporate debt : The risk structure of interest rates. 1974.

[16] Kenneth J. Singleton Darrell Duffie. Modeling term structures of defaultable bonds. 1999.

[17] Nicolas Privault. Financial risk and analytics. pages 213–225, 2023.

[18] Christian Menn Xin Guo, Robert A.Jarrow. A note on lando's formula and conditional independence. 2007.

[19] Marek Rutkowski Tomasz R. Bielecki. Credit risk : Modeling, valuation and hedging. 2002.

[20] Aurélien Alfonsi Damiano Brigo. Credit default swaps calibration and option pricing with the ssrd stochastic intensity and interest-rate model. 2005.

[21] Piotr Karasinski Fischer Black. Bond and option pricing when short rates are lognormal. 1991.