Université Paris-Dauphine – PSL
Mémoire de stage, M2 Ingénierie Statistique et Financière

Agrégation Byzantine-résiliente pour l'apprentissage distribué

La librairie Krum

Arthur Danjou

Stage effectué au
CMAP, École Polytechnique

Maître de stage : El-Mahdi El-Mhamdi, CMAP

Soutenu le 2 novembre 2026

Année universitaire 2025–2026

Remerciements

Je remercie El-Mahdi El-Mhamdi, mon maître de stage au CMAP, pour sa confiance, sa disponibilité et l'exigence avec laquelle il a encadré ce travail, de la conception de la librairie jusqu'à la rédaction du papier qui l'accompagne.

Je remercie [Nom Prénom] pour [encadrement quotidien, relectures, discussions].

Je remercie l'ensemble du CMAP pour son accueil, ainsi que l'équipe pédagogique du master ISF de l'Université Paris-Dauphine – PSL pour la formation qui a rendu ce stage possible.

Je remercie les membres du jury d'avoir accepté d'évaluer ce mémoire.

Enfin, je remercie mes proches pour leur soutien pendant ces mois de stage.

Table des matières

Résumé

Mots-clés : apprentissage distribué, tolérance aux fautes byzantines, agrégation robuste, reproductibilité.

Chapitre 1 Introduction

1.1 Contexte

1.2 Problématique

1.3 Contribution : la librairie Krum

1.4 Plan du mémoire

Chapitre 2 Cadre du stage

Ce chapitre décrit le cadre du stage : le laboratoire d'accueil, l'équipe d'encadrement, la mission initiale et le déroulement. La mission et le calendrier éclairent les choix techniques des chapitres suivants.

2.1 Le CMAP et ses axes de recherche

Le stage s'est déroulé au Centre de Mathématiques Appliquées (CMAP) de l'École Polytechnique, unité mixte de recherche (UMR 7641) du CNRS et de l'École, laboratoire de mathématiques appliquées. Le centre couvre l'optimisation, les probabilités, les statistiques et le calcul scientifique, avec une ouverture croissante vers l'apprentissage automatique. C'est dans ce dernier axe que s'inscrit le stage : l'apprentissage distribué robuste, à la frontière entre optimisation et systèmes distribués.

2.2 Équipe d'encadrement

Le maître de stage est El-Mahdi El-Mhamdi, chercheur au CMAP. Il est co-auteur des deux papiers fondateurs du sujet, côté règle d'agrégation comme côté attaque. Il s'agit de l'agrégation byzantine-résiliente de Blanchard et al. [2] et de la vulnérabilité cachée d'El-Mhamdi et al. [13] (chapitre 3, sections 3.3 et 3.4). Le stage M2 s'inscrit dans le master Ingénierie Statistique et Financière de l'Université Paris-Dauphine – PSL, avec le CMAP comme laboratoire d'accueil.

2.3 Objectifs du stage (travail demandé)

La mission initiale, telle que définie en début de stage, tient en une phrase : concevoir et développer une librairie de référence pour l'agrégation robuste de gradients. L'exigence centrale est la reproductibilité des protocoles publiés. La librairie unifie les agrégateurs, les attaques et l'orchestration de milliers d'expériences, adossée à des primitives testées (section 4.1). La reproduction de trois protocoles publiés est venue ensuite. Elle démontre la généricité de la librairie et son intégration. Elle s'adresse aux autres chercheurs (chapitre 5). Ces trois protocoles sont ceux de Blanchard et al. [2], d'El-Mhamdi et al. [13] et de Farhadkhani et al. [7] (section 4.2). L'orchestration n'était pas un outillage annexe mais un objectif du stage. Elle déroule des milliers d'expériences à partir d'une seule déclaration (section 4.3). L'orchestrateur collecte des métriques typées, c'est-à-dire

des relevés dont le type est vérifié à chaque ajout. Il ne relance que les expériences ratées, nouvelles ou modifiées, jamais celles qui ont réussi. Si une dépendance change, il relance tout le balayage, c'est-à-dire l'ensemble des expériences, pour ne jamais mélanger deux versions du code.

2.4 Déroulement du stage

Le stage a duré du 7 avril au 14 octobre 2026, soit un peu plus de six mois.

2.4.1 Une équipe recherche-produit de quatre

Le travail s'est fait dans une équipe recherche-produit de quatre personnes. Peva Blanchard, responsable produit (Product Owner : carnet de tickets, vision du produit et priorisation des fonctionnalités), est co-auteur du papier `Krum` de Blanchard et al. [2]. Sébastien Rouault, responsable technique (Tech Lead : architecture et revues de code), est co-auteur du papier `Bulyan` d'El-Mhamdi et al. [13]. Son code était déjà largement repris. Ses agrégateurs, ses attaques et son emballage sans copie (wrapper, section 4.1) circulaient tels quels dans de nombreux codes de recherche publiés. Mohamad-Ammar Said et moi-même assurions le développement. Le flux de travail suit les conventions du logiciel libre sur GitHub. Le suivi passe par tickets (issues : une fiche par tâche). Les contributions passent par demandes de fusion (pull requests), relues avant intégration. L'intégration continue vérifie chaque envoi : contrôle du style (lint), formatage, typage et tests automatiques. Ces rôles annoncent les acquis du stage : la priorisation côté produit et l'exigence d'architecture et de revue de code. Ils sont repris parmi les bénéfices du stage (chapitre 7).

Le rythme hebdomadaire était fixe. Le mardi et le mercredi matin, un point de synchronisation de quinze minutes avec Peva, Sébastien et Mohamad-Ammar servait à poser les questions sur le code et à lever les blocages. Le jeudi, une réunion de deux heures à cinq, avec El-Mahdi El-Mhamdi, traitait l'organisation générale et la revue des demandes de fusion. Une fois le code stabilisé, elle a traité la rédaction du papier compagnon. Le vendredi matin, une session de préparation du carnet (grooming : tri et chiffrage des tickets) sur le tableau du projet GitHub réunissait l'équipe resserrée.

2.4.2 Jalons

Le dépôt Krum porte la trace du calendrier : près de 600 commits et 26 demandes de fusion déposées au total au 14 septembre 2026, dont 23 fusionnées. Avant tout développement, le stage a commencé par un audit. Il s'agissait d'identifier le code de Sébastien dans les codes

de recherche des papiers publiés. Le constat était net : les chercheurs peinaient à intégrer ce code. Ils reprenaient agrégateurs et attaques intacts et copiaient-collaient l'emballage sans copie. C'est de cet audit qu'est née la liste des idées de la librairie. En parallèle, des reproductions de papiers menées de zéro sur un dépôt dédié [1] ont servi à identifier les tâches les plus coûteuses pour un chercheur qui implémente sa théorie en Python pour mener des expériences. Le socle initial reprend en avril du code préexistant de Sébastien Rouault, sous licence MIT (licence libre permissive), avec sa documentation. En mai-juin viennent les primitives : agrégateurs et attaques (demande de fusion n° 28, section 4.1). La simulation `MoNNA` décentralisée les rejoint à la mi-juin (n° 32), puis l'orchestrateur naît en version initiale fin juin (n° 39). En juillet, les modèles sont standardisés (n° 46) et les simulations et expériences s'étoffent (n° 51, n° 50), avec les tutoriels fin juillet (n° 55). Les partitionneurs (découpe des données entre workers, section 4.1) non IID arrivent en août (n° 59). La fin de l'été consolide l'orchestration. Septembre concentre la finition : la scission de `GeoMed` et `Medoid` (n° 63), les audits de conformité aux papiers (n° 65, en cours à mi-septembre) et les corrections de fidélité (section 5.4). Parallèlement, la rédaction du papier compagnon commence. Ce papier vise la rubrique Machine Learning Open Source Software (logiciels libres) du Journal of Machine Learning Research (JMLR [2]). Cette revue est la référence en apprentissage automatique pour les contributions logicielles. Elle évalue le code autant que les résultats, ce qui correspond à l'objet du stage. La figure 2.1 résume cette chronologie, état à mi-septembre.

Ce cadre fixe le décor et la mission. Le chapitre suivant pose les fondations scientifiques : protocoles distribués, règles d'agrégation et attaques (chapitre 3).

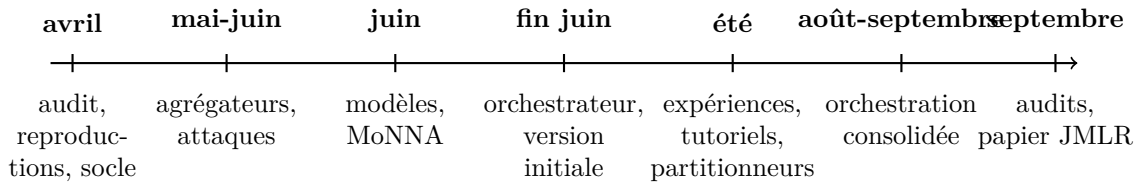

FIGURE 2.1 – Chronologie du stage, d'avril à septembre 2026, état à mi-septembre.

1. https://github.com/ArthurDanjou/Byzantine-Robust-Aggregation-Reproduction.
2. https://www.jmlr.org/

Chapitre 3 État de l'art

Ce chapitre pose les fondations reprises dans tout le mémoire. Il fixe le vocabulaire (section 3.1), décrit les protocoles et la résilience (section 3.2), les règles (section 3.3), les attaques (section 3.4) et le paysage des librairies (section 3.5).

3.1 Terminologie minimale

Ce chapitre fixe le vocabulaire repris dans tout le mémoire. Chaque chapitre reste lisible seul et réutilise ces termes avec un renvoi à cette section, sans les redéfinir.

3.1.1 Acteurs et protocoles

Worker. Un worker est une machine participante qui calcule un gradient sur sa part des données. Le mémoire note n leur nombre total. Il note f le nombre de byzantins.

Apprentissage fédéré. L'apprentissage fédéré entraîne un modèle commun sans centraliser les données. Chaque worker garde sa part locale. Ni les autres workers ni le serveur ne voient jamais ces données, seulement ses mises à jour de gradient ou de modèle. La figure 3.1, section 3.2, l'illustre.

Protocole centralisé. Dans un protocole centralisé, un serveur de paramètres reçoit les gradients et les agrège. Il diffuse ensuite le modèle mis à jour. Le serveur de paramètres est la machine qui centralise cette agrégation.

Protocole décentralisé. Dans un protocole décentralisé, les workers échangent leurs modèles de pair-à-pair sans serveur central. Pair-à-pair signifie que chaque worker dialogue directement avec un sous-ensemble d'autres workers.

Gossip. Le gossip est un mode d'échange où chaque worker tire au hasard les pairs dont il reçoit les modèles à chaque tour. Un tour (round) est une itération du cycle suivant : diffusion des paramètres, calcul local, puis agrégation. Ce tirage rend aléatoire le nombre de modèles byzantins reçus.

Byzantin. Un worker byzantin est un participant au comportement arbitraire, possiblement malveillant. Un worker honnête suit exactement le protocole prescrit.

3.1.2 Données et modèle

Données IID. Des données IID sont indépendantes et identiquement distribuées entre workers. Chaque part locale ressemble alors au jeu global. Le cas contraire est dit non IID.

Perceptron. Un perceptron multicouche (MLP) empile des couches linéaires suivies d'activations non linéaires.

Régularisation. La régularisation ajoute une pénalité L_2 à la perte pour limiter la taille des poids. Dans le gradient, elle vaut un rappel vers zéro.

Xavier. Xavier calibre la variance initiale selon la taille des couches [8] pour stabiliser le début de l'entraînement.

Dirichlet. Dirichlet tire les proportions de chaque classe par worker selon une loi de Dirichlet [10], et un α petit donne un déséquilibre fort.

3.1.3 Optimisation

Conditions de Robbins-Monro. Un pas décroissant $\eta(t)$ satisfait les conditions de Robbins et Monro [17] lorsque la somme des pas diverge, $\sum_t \eta(t) = \infty$, et que la somme de leurs carrés converge, $\sum_t \eta(t)^2 < \infty$. Ces conditions sont les hypothèses classiques de convergence de la descente stochastique.

Calendrier à décroissance. Le calendrier paramétrique $\eta(t) = \frac{r_\eta \eta_0}{t + r_\eta}$ vérifie ces conditions ; il est retenu pour rejouer les expériences d'El-Mhamdi et al. [13].

Momentum. Le momentum entretient une moyenne mobile des gradients passés pour lisser la trajectoire locale.

3.2 Apprentissage distribué Byzantine-résilient

Deux degrés de robustesse structurent ce chapitre. « Non robuste » (la moyenne) signifie l'absence de toute garantie : un seul gradient byzantin non borné suffit à déplacer l'agrégat arbitrairement loin. « Non résilient » (le médoïde) signifie l'échec au seul critère (α, f) de Blanchard et al. [2], précisé ci-dessous : l'attaquant doit pour cela coordonner ses f vecteurs, car un byzantin isolé aberrant n'est jamais sélectionné. Les garanties des médianes sont d'ordre statistique, la résilience (α, f) est une condition géométrique sur l'agrégat.

Le modèle de menace est simple. Chaque worker calcule un gradient sur sa part locale.

Les workers honnêtes suivent le protocole. Les workers byzantins proposent des vecteurs arbitraires, éventuellement coordonnés entre eux et informés des vecteurs honnêtes. Dans le protocole centralisé, un serveur de paramètres diffuse le modèle, collecte un gradient par worker à chaque tour synchrone, agrège les n vecteurs en une mise à jour unique, puis diffuse le modèle mis à jour. Sans défense, l'agrégation est la moyenne. Un seul vecteur byzantin non borné suffit alors à déplacer la moyenne arbitrairement loin, comme le montre le lemme 1 de Blanchard et al. [2]. L'architecture est résumée figure 3.1.

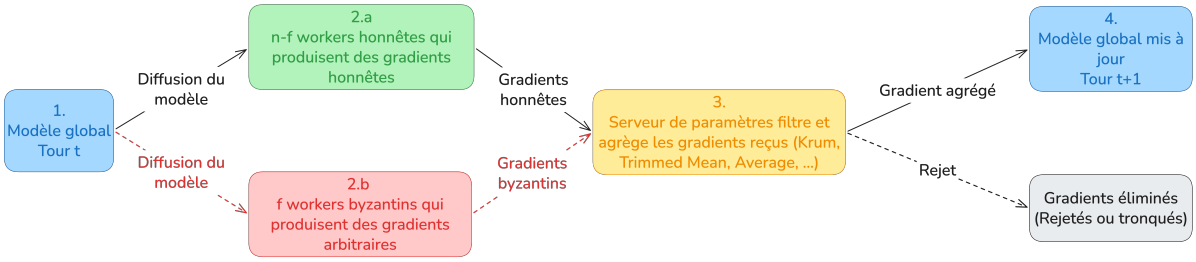

Figure 3.1 – Architecture de l'apprentissage fédéré centralisé en présence byzantine, du tour t au tour $t+1$. Les $n-f$ workers honnêtes calculent des gradients sur leurs données locales et les envoient au serveur de paramètres. Les f workers byzantins voient le modèle et envoient des vecteurs arbitraires, que le serveur écarte avant de diffuser le modèle mis à jour.

Le protocole décentralisé supprime le serveur. Chaque worker entretient son propre modèle, effectue un pas local sur son lot (batch, le sous-ensemble de ses données locales servant à ce pas), puis remplace son modèle par un mélange des modèles reçus de ses pairs. Quand la réception est aléatoire, chaque worker reçoit entre 0 et f modèles byzantins par tour. C'est le régime du gossip (section 3.1), repris par le protocole `MoNNA` de Farhadkhani et al. [7]. La figure 3.2 illustre ce tour décentralisé.

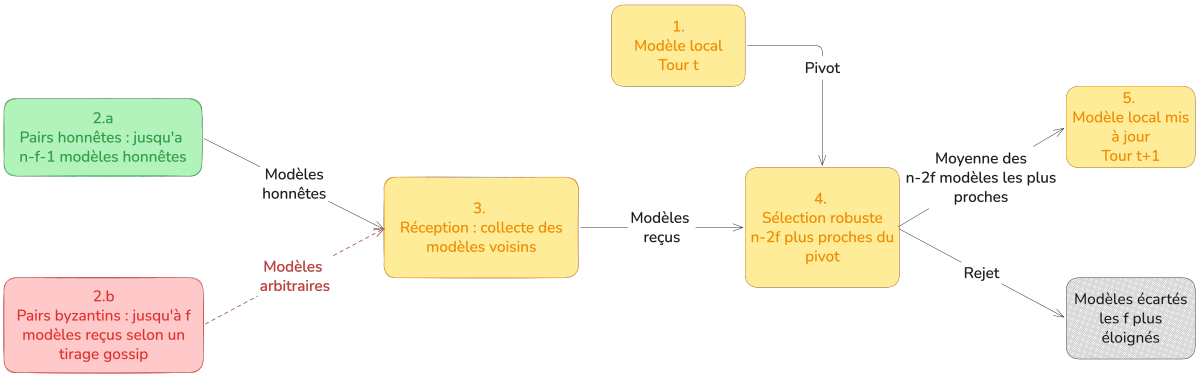

Figure 3.2 – Tour décentralisé, vu d'un worker honnête, du tour t au tour $t+1$. Le worker collecte les modèles de ses pairs, dont 0 à f modèles byzantins tirés au hasard selon le gossip. Il conserve les $n-2f$ modèles les plus proches de son modèle local, pris comme pivot, écarte les f plus éloignés, puis met à jour son modèle.

La résilience (α, f) de Blanchard et al. [2] donne un sens précis à « robuste ». Elle

demande deux choses à la règle d'agrégation. D'abord, son espérance doit pointer dans la même direction que le vrai gradient, à un angle $\alpha < \pi/2$ près. Ensuite, ses moments jusqu'à l'ordre 4 doivent rester contrôlés par ceux de l'estimateur honnête, pour que la dynamique discrète de la descente reste maîtrisée.

Définition 3.2.1 (Résilience (α, f) [2])**.** Soient $0 \leq \alpha < \pi/2$ et $0 \leq f \leq n$ un entier. Soient $\boldsymbol{V}_1, \ldots, \boldsymbol{V}_n$ des vecteurs aléatoires indépendants et identiquement distribués dans $\mathbb{R}^d$, avec $\boldsymbol{V}_i \sim G$ et $\mathbb{E}[G] = \boldsymbol{g}$, où G désigne la loi commune des estimateurs honnêtes et $\boldsymbol{g}$ leur espérance commune, le vrai gradient. Soient $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_f$ des vecteurs aléatoires quelconques dans $\mathbb{R}^d$, éventuellement dépendants des $\boldsymbol{V}_i$. Une règle d'agrégation F est (α, f)-résiliente si, pour tous $1 \leq j_1 < \cdots < j_f \leq n$, le vecteur

$$F = F(\boldsymbol{V}_1, \ldots, \underbrace{\boldsymbol{B}_1}_{j_1}, \ldots, \underbrace{\boldsymbol{B}_f}_{j_f}, \ldots, \boldsymbol{V}_n)$$

vérifie les deux conditions suivantes :

(i) $\langle \mathbb{E}[F], \boldsymbol{g} \rangle \geq (1 - \sin\alpha) \cdot \|\boldsymbol{g}\|^2 > 0$;

(ii) pour $r \in \{2, 3, 4\}$, $\mathbb{E}\|F\|^r$ est majoré par une combinaison linéaire de termes $\mathbb{E}\|G\|^{r_1} \cdots \mathbb{E}\|G\|^{r_{n-f}}$ avec $r_1 + \cdots + r_{n-f} = r$. [1]

Une règle (α, f)-résiliente fait converger la descente malgré f byzantins, pourvu que le bruit honnête reste petit devant le gradient.

Les chapitres suivants utilisent des pas fixes puis décroissants. En régime décroissant, la convergence suppose un pas au sens de Robbins et Monro [17] : la somme des pas diverge et la somme de leurs carrés converge. Sans elles, la garantie géométrique de la règle ne produit aucune convergence. El-Mhamdi et al. [13] utilisent le calendrier paramétrique de la section 3.1. L'initialisation suit Xavier (section 3.1), sans que sa théorie soit développée ici.

3.3 Règles d'agrégation de gradients

Les règles se rangent en quatre familles : sélection à base de distances, filtrage coordonnée par coordonnée, voisinage, et méthodes géométriques ou énumératives. La moyenne simple reste hors de ces familles : sans garantie d'après le lemme 1, elle sert de baseline, la référence témoin, dans tout le mémoire.

1. La définition d'origine [2] écrit $n - 1$ facteurs, alors que la preuve de cette même référence borne $\mathbb{E}\|F\|^r$ par les moments des $n - f$ estimateurs honnêtes. C'est cette version, effectivement démontrée, qui est retenue ici.

3.3.1 Sélection à base de distances

La famille à base de distances sélectionne les gradients selon leur proximité mutuelle.

Définition 3.3.1 (Krum [2])**.** Pour $i \neq j$, on note $i \to j$ le fait que $\boldsymbol{V}_j$ figure parmi les $n-f-2$ plus proches voisins de $\boldsymbol{V}_i$. Le score est $s(i) = \sum_{i \to j} \|\boldsymbol{V}_i - \boldsymbol{V}_j\|^2$. Krum renvoie

$$\mathrm{Kr}(\boldsymbol{V}_1, \ldots, \boldsymbol{V}_n) = \boldsymbol{V}_{i^*},$$

où i^* minimise s (plus petit identifiant en cas d'égalité). La garantie exige $2f + 2 < n$.

Le coût est $O(n^2(d + \log n))$.

Définition 3.3.2 (MultiKrum [2])**.** MultiKrum sélectionne un vecteur par Krum, retire ce vecteur de la liste et répète ce schéma $m-1$ fois, tant que $n-m > 2f+2$. Le départage de la définition 3.3.1 s'applique à chaque itération. Il renvoie

$$\mathrm{Kr}_m = \frac{1}{m} \sum_{s=1}^{m} \boldsymbol{V}_{i_s^*}.$$

Krum est le cas $m = 1$.

Définition 3.3.3 (Bulyan [13])**.** Soit A une règle (α, f)-résiliente, avec $n \geq 4f + 3$ gradients reçus. `Bulyan`(A) sélectionne $\theta = n - 2f$ vecteurs par itérations de A : à chaque itération, le vecteur le plus proche de la sortie de A rejoint l'ensemble S et quitte les candidats, jusqu'à $|S| = \theta$. Puis, pour chaque coordonnée i, soit median$[i]$ la médiane des θ valeurs et $G[i]$ la moyenne des $\beta = \theta - 2f$ valeurs les plus proches de la médiane. `Bulyan`(A) est aussi (α, f)-résiliente. La garantie exige $f \leq (n-3)/4$.

`Bulyan` est hybride : sa sélection est à base de distances, son filtrage opère coordonnée par coordonnée.

3.3.2 Filtrage coordonnée par coordonnée

Cette famille opère indépendamment sur chaque dimension du gradient.

Définition 3.3.4 (Médiane coordonnée par coordonnée [22])**.** Pour $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$ dans $\mathbb{R}^d$, la médiane coordonnée par coordonnée est le vecteur dont la k-ième coordonnée est la médiane des n k-ièmes coordonnées, soit $\mathrm{med}\{x_k^i : 1 \leq i \leq n\}$.

Définition 3.3.5 (Moyenne tronquée coordonnée par coordonnée [22])**.** Pour $\beta \in [0, 1/2)$

et des vecteurs $\boldsymbol{x}^1, \dots, \boldsymbol{x}^n$ dans $\mathbb{R}^d$, la moyenne β-tronquée a pour k-ième coordonnée

$$g_k = \frac{1}{(1-2\beta)n} \sum_{x \in U_k} x,$$

où U_k est obtenu en retirant les fractions β extrêmes des k-ièmes coordonnées. Avec $\beta = f/n$, on retire exactement f valeurs de chaque côté, et $\beta = 0$ redonne la moyenne.

La garantie de Yin et al. [22] couvre les deux règles sous majorité stricte, $f \leq (n-1)/2$. Pour la moyenne tronquée, la fraction élaguée β doit de plus couvrir la fraction byzantine réelle et rester strictement inférieure à $1/2$, ce que réalise le choix $\beta = f/n$. La médiane exige en outre une asymétrie bornée coordonnée par coordonnée.

3.3.3 Voisinage

Cette famille moyenne les voisins d'un pivot, le vecteur de référence depuis lequel se mesurent les distances.

Définition 3.3.6 (NNA [7])**.** Pour $\boldsymbol{z}^{(0)}, \dots, \boldsymbol{z}^{(n-f-1)}$ dans $\mathbb{R}^d$,

$$\texttt{NNA}(\boldsymbol{z}^{(0)}; \{\boldsymbol{z}^{(i)}\}) \coloneqq \frac{1}{n-2f} \sum_{i=0}^{n-2f-1} \boldsymbol{z}^{(\tau(i))},$$

où τ ordonne $\{1, \dots, n-f-1\}$ par distance croissante à $\boldsymbol{z}^{(0)}$ et où l'on pose $\tau(0) = 0$, de sorte que le pivot figure dans la somme. Dans le protocole `MoNNA`, chaque nœud agrège ainsi les $n-f-1$ vecteurs reçus avec le sien en éliminant les f plus éloignés. Le protocole `MoNNA` prend le modèle local comme pivot.

3.3.4 Méthodes géométriques et énumératives

Les règles restantes sont géométriques ou énumératives : elles exploitent directement la géométrie du nuage de gradients soumis.

Définition 3.3.7 (Médiane géométrique [19, 15])**.** La médiane géométrique de vecteurs $\boldsymbol{z}^1, \dots, \boldsymbol{z}^n$ est le minimiseur

$$\texttt{GeoMed}(\boldsymbol{z}^1, \dots, \boldsymbol{z}^n) = \arg\min_{\boldsymbol{x} \in \mathbb{R}^d} \sum_{i=1}^{n} \|\boldsymbol{x} - \boldsymbol{z}^i\|.$$

Elle n'a pas de forme fermée et se calcule par les itérations de Weiszfeld, lissées par un petit paramètre $\nu > 0$ qui évite la division par zéro. Son point de rupture, la fraction maximale de byzantins tolérable, vaut $1/2$, valeur optimale : elle résiste tant que $f < n/2$. Sa garantie est d'ordre statistique.

Définition 3.3.8 (Médoïde [20])**.** Le médoïde restreint la médiane géométrique aux vecteurs soumis : il renvoie celui qui minimise la somme des distances aux autres,

$$\texttt{Medoid}(\boldsymbol{z}^1, \dots, \boldsymbol{z}^n) = \underset{\boldsymbol{z} \in \{\boldsymbol{z}^1, \dots, \boldsymbol{z}^n\}}{\arg\min} \sum_{i=1}^{n} \|\boldsymbol{z} - \boldsymbol{z}^i\|.$$

Sans itération, il n'offre pas la même robustesse : des byzantins coordonnés peuvent le déplacer, de sorte qu'il ne satisfait pas le critère (α, f).

Définition 3.3.9 (AKSEL [3])**.** AKSEL prend pour pivot M la médiane coordonnée par coordonnée des n gradients reçus. Il conserve les $n - f$ gradients les plus proches de M et renvoie leur moyenne. Le coût est linéaire en la dimension, $O(nd)$, contre $O(n^2 d)$ pour les règles qui comparent tous les couples. Sa garantie (α, f) exige $n > 2f$. C'est la variante implémentée dans Krum : le papier sélectionne au seuil médian sans paramètre, voir l'annexe A.

Définition 3.3.10 (Brute [13])**.** Brute énumère tous les sous-ensembles de $n - f$ vecteurs parmi les n soumis et retient celui de diamètre minimal. Le diamètre est la plus grande distance entre deux de ses éléments. Brute renvoie la moyenne du sous-ensemble retenu. C'est la référence exacte de la sélection du sous-ensemble le plus compact, au coût $\Theta\binom{n}{f}$, sous majorité stricte $f \leq (n-1)/2$.

3.4 Stratégies d'attaque byzantines

Une règle ne se juge que face à l'attaque qui la met en défaut : aux familles de règles répondent des stratégies d'attaque, décrites avec la même précision. Toutes supposent l'attaquant informé des honnêtes, voire omniscient au sens d'El-Mhamdi et al. [13] : connaissance parfaite de l'état du système, mais seuls f gradients soumis par tour. Les attaques se rangent en deux groupes : génériques, puis celles de l'étude de la vulnérabilité cachée des règles de sélection.

3.4.1 Attaques génériques

Définition 3.4.1 (SignFlip [21])**.** Dans la classe des manipulations de produit scalaire de Xie et al., l'instanciation locale `SignFlip` renvoie la négation de la moyenne honnête, répétée f fois : tous les byzantins envoient $-\text{scale} \cdot \bar{g}$, où `scale` est un hyperparamètre d'implémentation. L'agrégat pointe alors à l'opposé du vrai gradient et la descente remonte la perte.

Définition 3.4.2 (ALIE [1])**.** Soient μ_j et σ_j la moyenne et l'écart-type honnêtes sur la coordonnée j, et $s = \lfloor n/2 \rfloor + 1 - f$ le nombre de workers à séduire pour une majorité. Avec

Φ la fonction de répartition normale centrée réduite, $z^{\max} = \max\{z : \Phi(z) < (n-s)/n\}$. ALIE fait envoyer à tous les byzantins $(p_{\text{mal}})_j = \mu_j \pm z^{\max}\sigma_j$ sur chaque coordonnée. Les valeurs tombent dans $(\mu - z\sigma, \mu + z\sigma)$ et deviennent la médiane avec grande probabilité, survivant à la moyenne tronquée. La variante oracle connaît exactement μ et σ.

Définition 3.4.3 (Gaussian)**.** Gaussian envoie des gradients tirés selon $\mathcal{N}(\mu, \sigma^2 I_d)$, indépendamment des honnêtes. Attaque agnostique, elle sert de baseline dans ce mémoire.

3.4.2 Attaques de la vulnérabilité cachée

Définition 3.4.4 (FullGradientNegation [13])**.** La négation du gradient complet est l'instanciation locale de l'attaquant omniscient d'El-Mhamdi et al. : connaissant le gradient g_{full} calculé sur tout le jeu, l'attaquant renvoie $-\kappa\, g_{\text{full}}$, répété f fois, où κ est un hyperparamètre d'implémentation.

Définition 3.4.5 (SmallPerturbation [13])**.** Soit Q l'ensemble des $n - f$ gradients honnêtes, de moyenne $\bar{g}$, et E une direction de perturbation. L'attaque place $B(\gamma) = \bar{g} + \gamma E$ et recherche le plus grand γ_m encore sélectionné par l'agrégateur cible. Les f byzantins envoient tous $B(\gamma_m)$. En dimension $d \gg 1$, deux gradients honnêtes diffèrent naturellement de $\Theta(\sqrt{d})$, si bien qu'un décalage de cet ordre sur une seule coordonnée passe la sélection. Par défaut, E est le vecteur de base de la coordonnée de plus forte variance honnête. Dans la variante en norme infinie, $E = (1, \ldots, 1)$ déplace toutes les coordonnées.

La leçon est démontrée en section 5.2 : ressembler aux honnêtes en norme ne suffit pas, une coordonnée empoisonnée passe les règles de type `Krum`. C'est la malédiction de la dimensionnalité.

3.5 Librairies existantes : un paysage fragmenté

De nombreuses librairies fédérées et byzantines existent. Cinq systèmes ont été retenus comme les plus pertinents : ByzFL [9], FedLab [23], Blades [12], FL-Byzantine-Library [6, 14] et ByzPy [4], sans vocabulaire ni protocole d'évaluation communs.

Le partitionnement illustre la divergence des taxonomies. Quatre partitionneurs derrière une interface unique côté Krum, contre deux à neuf schémas ailleurs, tous bâtis des mêmes mécaniques de tri par étiquette, proportions de Dirichlet et découpage.

Les protocoles se rangent en trois régimes. Le fidèle rejoue les publications à conditions d'origine. Le générique et les baselines ne rejouent aucun protocole précis. Seul Krum est fidèle, avec trois protocoles. Seul ByzPy propose en plus le pair-à-pair, en générique.

L'orchestration confond ailleurs balayage et déploiement. KRUM déclare les balayages en Python et collecte des métriques typées, avec une exécution mono-hôte, quand les autres configurent par fichiers ou par gestionnaires, avec des journaux sur disque. Seuls FedLab et ByzPy dépassent l'hôte unique.

La section 6.3 justifie ce choix parmi les nombreuses librairies existantes et détaille la comparaison en trois blocs.

Les règles, les attaques et leurs garanties étant ainsi fixées, le chapitre suivant décrit comment KRUM les traduit en logiciel, des primitives aux simulations puis à l'orchestration (chapitre 4).

Chapitre 4 Conception de Krum

Krum s'organise en trois couches distinctes : les *primitives*, briques de base sans mémoire interne, appelées comme de simples fonctions (agrégateurs, attaques, partitionneurs) ; les *simulations*, qui reproduisent fidèlement les protocoles publiés ; et l'*orchestration*, qui déroule un balayage de milliers d'expériences à partir d'une seule déclaration, ne relance que ce qui a échoué et restitue des résultats prêts à analyser. Ce chapitre suit cette structure : chaque section correspond à une couche de la librairie, et les choix de conception y sont systématiquement justifiés par l'objectif d'ensemble, rendre les protocoles d'agrégation byzantine-résiliente fidèlement reproductibles. Le code, les tutoriels et la documentation sont disponibles en ligne [1]

4.1 Primitives sans état

Les primitives sont les briques de calcul de Krum : ce sont elles qui transforment des tenseurs, sans mémoire d'un appel à l'autre. Cette section expose d'abord le pattern de conception unique qui les unit et l'extensibilité qu'il procure, puis passe en revue les trois familles de briques, à savoir les agrégateurs, les attaques et les partitionneurs, avant de décrire les modèles, entre wrapper zéro-copie et architectures reprises de la littérature.

4.1.1 Un pattern de conception unique

Toutes les briques de base de Krum suivent un même pattern : chaque classe s'emploie sans être instanciée, avec une unique méthode appelable directement et aucun état conservé entre deux appels. Les agrégateurs implémentent `aggregate`, les attaques `generate`, les partitionneurs `partition`. Pour les agrégateurs et les attaques, le premier argument est toujours le tenseur d'entrée, passé par position et sans nom : les gradients des n workers en vues plates pour une règle d'agrégation, les gradients honnêtes pour une attaque. Les hyperparamètres spécifiques sont nommés explicitement, comme le nombre de byzantins f pour les règles d'agrégation. Dans tout le chapitre, n et f suivent la terminologie minimale (section 3.1).

Ce choix élimine la configuration d'objets, source classique d'erreurs silencieuses : il n'existe pas d'état à initialiser, donc pas d'oubli d'initialisation possible. Une méthode de classe se comporte comme une fonction pure, à l'exception de l'écriture éventuelle dans le buffer `out` fourni par l'appelant ; elle est donc triviale à tester et à composer. Enfin,

1. https://github.com/calicarpa/krum et https://calicarpa.github.io/krum.

agrégateurs et attaques tournent dans la boucle chaude de l'entraînement, à chaque pas et pour des balayages de milliers d'exécutions. Leur méthode accepte donc un buffer `out` pré-alloué en argument optionnel : sans `out`, elle alloue et renvoie un tenseur neuf ; avec `out`, elle écrit le résultat en place et renvoie ce même objet. Un appelant qui enchaîne les pas alloue ainsi une fois pour toutes et réutilise le même espace : plus aucune allocation dans la boucle.

4.1.2 Extensibilité

La conséquence directe de ce pattern est qu'une extension de la librairie se réduit à une méthode. Ajouter une règle d'agrégation revient à implémenter `Aggregator.aggregate` ; ajouter une attaque, `Attack.generate` ; ajouter un partitionneur, `DataPartitioner.partition`. Aucune instance n'est à configurer, aucun état n'est à maintenir, et les hyperparamètres restent de simples arguments nommés.

L'extension s'intègre alors immédiatement aux couches supérieures, sans modification de celles-ci. Une nouvelle règle d'agrégation est utilisable telle quelle dans les trois protocoles de simulation (section 4.2) et dans les balayages d'orchestration (section 4.3) : elle devient comparable aux onze règles existantes dans le même balayage d'expériences, avec le même cache et les mêmes métriques. C'est cette propriété qui fait de Krum une base d'expérimentation plutôt qu'une collection figée d'algorithmes.

4.1.3 Règles d'agrégation

Krum embarque onze règles d'agrégation couvrant les familles les plus citées de la littérature. Le tableau 4.1 les récapitule avec leurs garanties. Dans la suite, les noms de classes sont abrégés : `NNA` pour `NearestNeighborAverage`, et les attaques sans leur suffixe `Attack` (`ALIE` pour `ALIEAttack`, par exemple) ; `Krum` désigne la règle d'agrégation, à distinguer de Krum qui nomme la librairie.

Règle	Principe	Garantie
Sélection à base de distances		
`Krum`	gradient minimisant la somme des distances au carré aux plus proches	$f \leq (n-3)/2$ [2]
`MultiKrum`	moyenne des m gradients sélectionnés	$f \leq (n-3)/2$, $m \leq n - 2f - 3$ [2]
Voisinage		
`NNA`	moyenne des k plus proches voisins d'un pivot ($k =$ `num_closest`)	politique de l'appelant ($n - 2f$ pour `MoNNA`) [7]
Filtrage coordonnée par coordonnée		
`Average`	moyenne simple	baseline non robuste
`Median`	médiane coordonnée par coordonnée	$f \leq (n-1)/2$ et bruit contrôlé [22]
`TrimmedMean`	moyenne tronquée par coordonnée	$f \leq (n-1)/2$ et $\beta \geq f/n$ [22]
`Bulyan`	`MultiKrum` puis `TrimmedMean` coordonnée par coordonnée	$f \leq (n-3)/4$ [13]
Méthodes géométriques et énumératives		
`GeoMed`	médiane géométrique par itérations de Weiszfeld lissées [19, 15]	$f \leq (n-1)/2$ [15]
`Medoid`	médoïde (médiane restreinte aux vecteurs soumis)	baseline non résiliente [2] ; définition [20]
`Aksel`	pivot médian, temps linéaire en $O(nd)$	$f \leq (n-1)/2$ (point de rupture optimal) [3]
`Brute`	énumération des sous-ensembles	exacte ($f \leq (n-1)/2$)

TABLE 4.1 – Règles d'agrégation implémentées dans KRUM et leurs garanties. n désigne le nombre total de workers, f le nombre de byzantins, m le nombre de gradients sélectionnés par `MultiKrum`, d la dimension des gradients. « Baseline » signale une règle incluse comme point de comparaison. Les garanties des médianes sont statistiques (majorité stricte, sous réserve de bruit, définitions 3.3.4 et 3.3.5), et β désigne la fraction élaguée de `TrimmedMean` ; la résilience (α, f) est géométrique.

Quatre familles se dégagent. Les règles *à base de distances* (`Krum`, `MultiKrum`) sélectionnent les gradients selon leur proximité mutuelle ; `Krum` suppose $f \leq (n-3)/2$, et `MultiKrum` généralise ce principe en moyennant les m gradients les mieux classés, avec $m \leq n - 2f - 3$. `Krum` correspond au cas $m = 1$. Les règles *coordonnée par coordonnée* (`Average`, `Median`, `TrimmedMean`) opèrent indépendamment sur chaque dimension du gra-

dient ; c'est cette propriété que `Bulyan` exploite en composant `MultiKrum` (sélection) avec `TrimmedMean` (filtrage), ce qui lui permet de tolérer tout f vérifiant $f \leq (n-3)/4$ [13]. Les règles de *voisinage* (`NNA`) moyennent les plus proches voisins d'un pivot, le vecteur de référence depuis lequel se mesurent les distances ; leur garantie est une politique de l'appelant ($n - 2f$ pour `MoNNA`). Enfin, les règles *géométriques et énumératives* (`GeoMed`, `Medoid`, `Aksel`, `Brute`) exploitent la structure du nuage de gradients : `GeoMed` calcule la médiane géométrique par itérations de Weiszfeld lissées [15] ; `Medoid` en est la variante médoïde restreinte aux vecteurs soumis [20], baseline sans résilience (α, f) [2] ; `Aksel` pivote sur la médiane coordonnée par coordonnée et moyenne les $n - f$ gradients les plus proches en temps linéaire $O(nd)$; et `Brute` examine exhaustivement les sous-ensembles de taille $n - f$ et sert de référence exacte : le sous-ensemble de diamètre minimal, au prix d'un coût en $\binom{n}{f}$.

4.1.4 Attaques

Évaluer une règle d'agrégation suppose de savoir l'attaquer. KRUM fournit cinq stratégies, également sans état, implémentant `Attack.generate` à partir des gradients honnêtes et du nombre de byzantins f : `SignFlip` retourne $-\bar{g}$, la négation de la moyenne honnête des gradients, répétée f fois à un facteur `scale` près [21] ; `ALIE` place les gradients malveillants en $\mu - z\sigma$, à z écarts-types sous la moyenne honnête coordonnée par coordonnée, dans sa variante oracle (l'attaquant connaît exactement moyenne et dispersion des honnêtes) [1] ; `Gaussian` tire des gradients selon $\mathcal{N}(\mu, \sigma^2 I_d)$ ($\mu = 0$, $\sigma = 200$ par défaut), indépendamment des honnêtes (attaque dite agnostique), et sert de baseline d'attaque. Les deux dernières proviennent de l'étude de la vulnérabilité cachée des règles de sélection [13] : `FullGradientNegation` renvoie $-\kappa\, g_{\text{full}}$, le gradient calculé sur tout le jeu, supposé connu de l'attaquant, avec $\kappa = 100$ par défaut, et `SmallPerturbation` place les gradients malveillants $\bar{g} + \gamma E$ sur la frontière d'une boule $\mathcal{B}(\gamma)$ centrée sur la moyenne honnête, en recherchant le plus grand γ que l'agrégateur cible admet encore.

`SmallPerturbation` mérite une mention particulière : c'est la seule de ces attaques à exploiter finement la frontière de sélection des règles de type `Krum`, qui admettent des vecteurs proches du groupe des gradients honnêtes mais arbitrairement mal orientés, une manifestation de la malédiction de la dimensionnalité. Inclure dans la librairie l'attaque qui met en défaut ses propres règles n'est pas anecdotique : c'est la condition d'une évaluation honnête, et le chapitre 5 s'appuie précisément sur cette configuration.

4.1.5 Partitionneurs de données

Le passage à des données non IID est indispensable pour reproduire les conditions réalistes des protocoles fédérés, au sens de la terminologie minimale (section 3.1). KRUM fournit quatre partitionneurs derrière une interface unifiée : chaque partitionneur implémente `partition` et retourne une liste contenant un sous-jeu de données par worker, à partir d'un jeu d'entraînement et d'une graine.

`IidPartitioner` réalise un découpage IID par mélange et segmentation. `PerLabelsPartitioner` trie le jeu par étiquette puis affecte les segments aux workers avec une granularité $\lambda \in [0, 1]$: à $\lambda = 1$, on retrouve un découpage IID ; à $\lambda = 0$, le déséquilibre pathologique où chaque worker ne voit qu'une seule classe. `DirichletPartitioner` distribue les classes selon le paramètre α (section 3.1) [10] : un α petit produit un déséquilibre extrême, allant jusqu'au cas limite d'un worker sans échantillon d'une classe, que la librairie accepte explicitement (jeux partiels ou vides autorisés). Enfin, `MixingPartitioner` mélange deux partitionneurs quelconques selon un coefficient $\gamma \in [0, 1]$: une fraction $1 - \gamma$ du jeu passe par le premier, le reste par le second, et chaque worker reçoit la concaténation des deux parts. On parcourt ainsi un continuum de régimes IID vers non IID avec un seul paramètre de balayage.

4.1.6 Modèles : wrapper zéro-copie et architectures de la littérature

Les règles d'agrégation opèrent sur des gradients aplatis, or les tenseurs de PyTorch sont structurés par couche. Le wrapper `Model` expose tout `torch.nn.Module` en vues plates de ses paramètres et de ses gradients. Les vues partagent un unique buffer contigu, alloué une seule fois : la lecture et l'écriture se font sans copie par accès.

L'enjeu est autant la performance que l'ergonomie. Sans ce wrapper, chaque accès à un gradient complet suppose un `flatten`, un aplatissement en un vecteur, et une copie, payés à chaque pas d'entraînement et pour chaque worker. Avec la vue zéro-copie, l'agrégateur lit et écrit directement dans la mémoire du modèle, et le coût de la couche d'interface devient négligeable devant le calcul. Parmi les librairies comparées au chapitre 6, KRUM est la seule à offrir cette garantie.

Le paquet `models` ne se limite pas au wrapper : KRUM embarque aussi six architectures prêtes à l'emploi, reprises des publications reproduites, `Krum2017MLPMnist`, `Krum2017MLPSpambase` et `Krum2017CNN` pour les protocoles centralisés. Pour le protocole décentralisé, `Monna2023SmallMnist`, `Monna2023CNNMnist` et `Monna2023CNNCifar10` reprennent l'appendice de la publication `MoNNA`. Elles sont détaillées en annexe C.

4.2 Simulations fidèles aux protocoles publiés

La couche *simulations* est celle où se joue la promesse de reproductibilité. KRUM embarque trois protocoles d'entraînement implémentés d'après les publications : le serveur de paramètres centralisé de BLANCHARD et al. [2] (NeurIPS 2017), le protocole d'EL-MHAMDI et al. [13] (ICML 2018), et le protocole décentralisé pair-à-pair `MoNNA` de FARHADKHANI et al. [7] (ICML 2023). « Fidèle » a ici le sens que précise la section 5.4 : le cadre du protocole publié est repris, sans choix par défaut génériques, et les expériences du chapitre 5 en retrouvent le phénomène.

Les simulations partagent un contrat de données unique : *le code appelant partitionne, la simulation découpe en lots*. L'utilisateur choisit un partitionneur (section 4.1) et le gestionnaire de simulation se charge des lots, de leur tirage et de la boucle d'entraînement. Les deux topologies de la littérature sont couvertes : centralisée, avec un serveur de paramètres, et décentralisée pair-à-pair pour `MoNNA`. Dans ce dernier cas, avec f byzantins, le paramètre `byzantine_reach` distingue deux régimes de diffusion : `all`, le pire cas où chaque modèle byzantin atteint chaque worker, et `sampled`, le gossip (section 3.1) où chaque worker reçoit entre 0 et f modèles byzantins. Un paramètre `stop_attack_at` permet par ailleurs d'arrêter les attaques en cours d'entraînement, pour étudier la récupération d'un modèle après empoisonnement.

À code, graines et matériel fixés, le déterminisme est total : la graine globale est passée au constructeur et exploitée par `setup`, et chaque worker dispose de son propre générateur aléatoire, ce qui rend deux exécutions strictement identiques. La section 5.4 précise ce que cette garantie recouvre. Le listing 4.1 illustre l'enchaînement canonique d'utilisation.

```
# train_set and test_set are the full training and test sets.
worker_datasets = IidPartitioner.partition(train_set, n=25, seed=42)
sim = KrumSimulation(
    model_cls=Krum2017MLPMnist, train_datasets=worker_datasets,
    test_set=test_set, aggregator=Krum, seed=42,
    n=25, f=5, rounds=100, batch_size=32, lr=0.1,
)
sim.setup()
for step in range(100):
    sim.step()
metrics = sim.evaluate()
```

Listing 4.1 – Enchaînement canonique d'utilisation d'une simulation.

Le chapitre 5 revient en détail sur ce que cette fidélité garantit et ce qu'elle ne garantit

pas.

4.3 Orchestration d'expériences

La troisième couche répond au besoin qui motive l'ensemble : une règle d'agrégation ne s'évalue pas sur une expérience, mais sur des milliers. Une simulation rejoue un protocole à paramètres fixés. Une expérience, aussi appelée exécution, est un appel à `Orchestrator.run`, soit un point de paramètres avec sa collecte de métriques. Un balayage est l'ensemble des expériences croisées par des boucles Python. Le point d'entrée est unique : l'utilisateur déclare en boucles Python le croisement des paramètres à explorer, et l'orchestrateur en déroule toutes les combinaisons, une expérience par appel à `Orchestrator.run`. Les paramètres sont appariés à la signature de la fonction d'expérience (noms et valeurs par défaut de ses arguments) et convertis en clés hachables, des identifiants uniques calculés par hachage, qui servent à la fois d'identité d'exécution et de clé de cache ; il en découle le cycle de travail visé : lancer le balayage, récupérer des résultats prêts à analyser, relancer uniquement ce qui a échoué.

Dans une exécution (*run*), l'utilisateur déclare des métriques typées avec `Metric(name, dtype)` (`dtype` impose le type des valeurs, vérifié à chaque ajout) puis les alimente par `push(step, value)` ; chaque point est automatiquement étiquetté avec les paramètres de l'exécution. La récupération se fait par `get(name).to_pandas()`, qui retourne une table prête à analyser, avec pour colonnes les paramètres du balayage, puis `step` et `value`, directement exploitables avec `filter`, `query`, `groupby` et `merge` de pandas, la librairie des tables de données en Python. L'exemple canonique, un balayage croisant n, f, règles d'agrégation et attaques avec la baseline honnête, est détaillé en annexe B.

L'exécution est conçue pour la robustesse de ces balayages massifs. Elle est à échec rapide (*fail fast*) : toute expérience qui échoue interrompt le balayage en nommant précisément les paramètres fautifs, plutôt que de laisser une courbe vide passer inaperçue. La relance ne rejoue que les expériences nouvelles, modifiées ou ratées, jamais celles qui ont réussi. L'invalidation du cache est à la fois fine et globale : modifier un paramètre ne relance que les expériences concernées ; modifier une dépendance de la simulation est détecté par l'orchestrateur, qui relance alors l'intégralité du balayage, de sorte qu'aucune table de résultats ne mélange jamais deux versions du code. Enfin, l'exécution peut être séquentielle ou parallélisée à raison d'un processus par expérience, sans changer le contrat `push`/`get` ; le déploiement visé tient sur une seule machine (mono-hôte), en cohérence avec la vocation de la librairie : le balayage d'expériences simulées, non l'infrastructure distribuée réelle.

Enfin, la promesse de réutilisabilité se concrétise dans le logiciel publié. Krum est testée

(suite de tests automatiques `pytest` couvrant les garanties des règles), documentée [2] et publiée sur PyPI, le dépôt public de paquets Python, sous licence MIT, avec une compatibilité assurée pour Python 3.10 à 3.14. C'est la preuve que la chaîne complète (primitives, protocoles fidèles, balayage reproductible) est livrée sous forme utilisable, et non simplement décrite. Le chapitre suivant met cette chaîne à l'épreuve en rejouant trois protocoles publiés avec ses primitives, ses simulations et son orchestrateur (chapitre 5).

2. https://calicarpa.github.io/krum.

Chapitre 5 Validation expérimentale

Dans tout le chapitre, n, f, IID et les noms d'outils (perceptron, régularisation, Xavier, Robbins-Monro, momentum, Dirichlet) suivent la terminologie minimale (section 3.1). Les règles et attaques suivent les sections 3.3 et 3.4. La fonction de perte choisie dans les trois expériences est la cross-entropie. Les trois premières sections s'inspirent de protocoles publiés et visent à en retrouver le phénomène ; la section 5.4 en précise le sens.

5.1 Reproduction Blanchard et al. (NeurIPS 2017) : MultiKrum contre Average

L'expérience s'inspire du protocole Spambase de Blanchard et al. [2], en régime centralisé, et vise le phénomène rapporté, la divergence de `Average` sous attaque contre la convergence de `MultiKrum`, plutôt que la recopie de ses réglages. Le jeu de données est Spambase, avec 57 variables pour une classification binaire. Le découpage vaut 80 % d'entraînement et 20 % de test, à graine 42, avec des variables standardisées. Le modèle est un perceptron $57 \to 20 \to 20 \to 2$, avec ReLU après chaque couche cachée. Les tailles cachées 20 et 20 ne sont pas précisées dans le papier et reprennent les références du laboratoire. L'expérience compte $n = 20$ workers dont $f \in \{0, 6\}$ byzantins. L'attaque est `SignFlip` d'amplitude 10. L'entraînement dure 300 tours avec un pas $\eta = 0{,}01$ fixe. La taille de lot vaut 3 par worker. L'évaluation a lieu tous les 15 tours. La régularisation vaut 10^{-4}. L'initialisation est Xavier. La graine vaut 42 et le découpage est IID. Écart assumé : `MultiKrum` tourne avec $m = 12$ sous attaque ($m = 18$ sans attaque), soit $m = n - f - 2$ dans les deux cas, la taille du voisinage de score de Krum. Ce réglage sort de la plage de garantie, qui exige $m \leq n - 2f - 3 = 5$. C'est un réglage large, proche de `Average`, qui isole le comportement byzantin de l'effet d'une sélection stricte.

La figure 5.1 croise les deux règles avec et sans attaque. Sans byzantin, `Average` et `MultiKrum` convergent ensemble : la perte test décroît de 0,80 à 0,44 et l'exactitude atteint 0,82. Sous attaque ($f = 6$), leurs destins divergent : la perte de `Average` décolle vers le tour 25 et sort du cadre avant le tour 50 (poids devenus indéfinis, *NaN*, ensuite), tandis que `MultiKrum` suit sa trajectoire sans attaque, autour de 0,80 d'exactitude. Les panneaux de droite, qui excluent la courbe divergente, montrent cette superposition : à conditions égales, la robustesse coûte ici deux à trois points d'exactitude.

La courbe d'exactitude de `Average` attaqué mérite un mot : après un début autour de 0,40, elle saute à un palier de 0,60. C'est un artefact, pas un apprentissage : une fois les poids

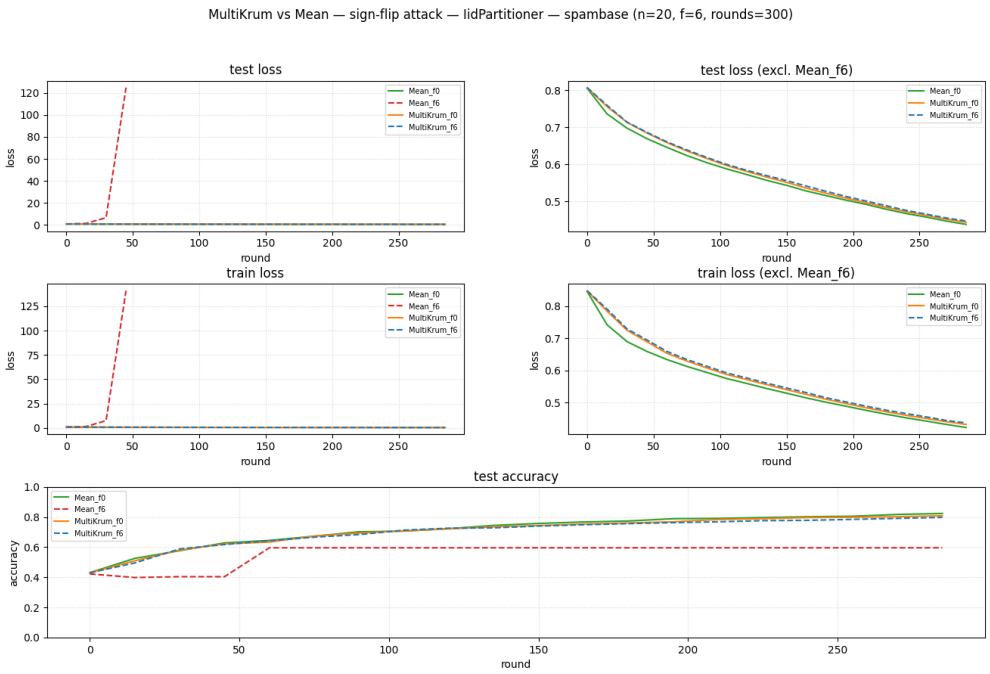

FIGURE 5.1 – Reproduction du protocole Spambase sous `SignFlip` d'amplitude 10, avec $n = 20$, $f = 6$, 300 tours et graine 42. Les panneaux montrent les pertes test et d'entraînement, avec puis sans la courbe divergente de `Average` attaqué. L'exactitude test de `MultiKrum` attaqué rejoint les courbes saines autour de 0,80.

indéfinis, la prédiction bascule sur la classe majoritaire du jeu ($\approx 60\,\%$ des exemples). Le phénomène du papier est reproduit : `MultiKrum` encaisse six byzantins sur vingt, soit 30 %, sans décrocher, là où `Average` s'effondre.

5.2 Reproduction El-Mhamdi et al. (ICML 2018) : MultiKrum contre Bulyan

L'expérience s'inspire du protocole d'EL-MHAMDI et al. [13], en régime centralisé, et vise le phénomène rapporté, la vulnérabilité cachée de `MultiKrum` et le filtrage de `Bulyan`, plutôt que la recopie de ses réglages. Le jeu de données est MNIST et le modèle est un perceptron $784 \to 100 \to 10$, avec ReLU après la couche cachée. Le jeu est utilisé en entier, soit 60000 exemples d'entraînement et 10000 de test, normalisés. L'expérience compte $n = 39$ workers dont $f \in \{0, 9\}$ byzantins. Ce compte satisfait $n = 4f + 3$, la borne de `Bulyan`. `MultiKrum` tourne en variante $m = 1$, soit `Krum`. L'attaque est `SmallPerturbation` à $\gamma = 280$ fixé. L'entraînement dure 100 tours. La taille de lot vaut 32 par worker. L'évaluation a lieu tous les 10 tours. Le pas initial vaut $\eta_0 = 0{,}1$ avec une décroissance de Robbins-Monro. Le calendrier utilise $r_\eta = 10000$. La graine vaut 42 et le découpage est IID. La régularisation vaut 10^{-4} et Xavier est actif par défaut. La même attaque, calibrée contre `MultiKrum`,

frappe `Bulyan`. `Bulyan` tourne avec $m = 1$ et un budget déclaré de 9 byzantins. L'attaque vise la norme 2 sur la coordonnée de plus forte variance honnête. Elle est maintenue pendant les 100 tours, sans arrêt en cours d'entraînement. `Average` sans attaque sert de référence saine. Allègement principal : γ est fixé au lieu d'une recherche de frontière à chaque tour.

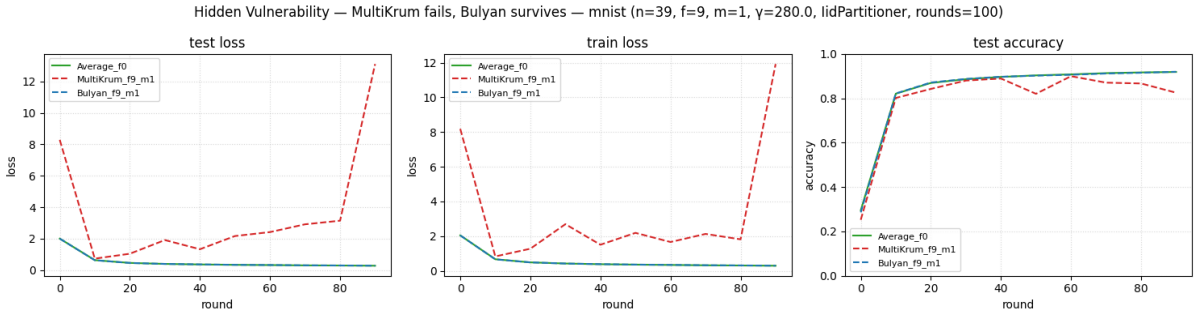

FIGURE 5.2 – Vulnérabilité cachée sous `SmallPerturbation` à $\gamma = 280$ fixé, avec MNIST, $n = 39$, $f = 9$, 100 tours et graine 42. Les panneaux montrent les pertes test et d'entraînement ainsi que l'exactitude test. `MultiKrum` avec $m = 1$ décroche et diverge en fin de course, tandis que `Bulyan` reste proche de la référence sans attaque.

La figure 5.2 montre trois trajectoires, aux tours 0 à 90. L'évaluation a lieu tous les 10 tours pour 100 tours. Sans attaque, `Average` converge proprement : perte test $\approx 0{,}28$, exactitude $\approx 0{,}92$. Sous attaque, `MultiKrum` décroche : sa perte part de ≈ 8, oscille entre 1 et 3 avant de diverger en fin de course (≈ 13), et son exactitude, erratique, finit neuf points derrière ($\approx 0{,}83$ contre $0{,}92$). `Bulyan`, frappé par la même attaque, colle à la référence sur les trois panneaux (perte $\approx 0{,}30$, exactitude $\approx 0{,}915$).

Le mécanisme du papier est reproduit : à $m = 1$, la sortie de `MultiKrum` est le vecteur byzantin lui-même, et chaque pas est empoisonné sur une coordonnée ; la moyenne tronquée coordonnée par coordonnée de `Bulyan` détecte la coordonnée décalée et la filtre. D'où la leçon : ressembler aux honnêtes ne suffit pas à être robuste.

5.3 Étude de cas à échelle réduite de Farhadkhani et al. (ICML 2023) : MoNNA décentralisé (NNA contre moyenne)

Les conditions s'inspirent du protocole de FARHADKHANI et al. [7] : mêmes briques algorithmiques (mélange `NNA`, momentum de Polyak, hétérogénéité Dirichlet), mais un modèle réduit, moins de nœuds et moins de tours pour tenir sur CPU. Le régime est décentralisé. Le jeu de données est MNIST, avec 4096 exemples d'entraînement et 1024 de test. Les images sont seulement converties en tenseurs, sans normalisation. Le modèle est un petit perceptron $784 \to 128 \to 10$ avec ReLU après la couche cachée, au lieu du CNN du papier. L'expérience compte $n = 16$ workers dont $f \in \{0, 5\}$ byzantins. Le mélange est `NNA`,

contre `Average`, avec $n - 2f = 6$ modèles conservés parmi les $n - f = 11$ reçus. L'attaque est `SignFlip` d'amplitude 1. La diffusion est le pire cas, où chaque modèle byzantin atteint chaque worker. Cela correspond à `byzantine_reach` qui vaut `all` par défaut dans le code. Le pas local utilise un momentum avec $\beta = 0{,}99$ et $\eta = 0{,}5$. L'entraînement dure 200 tours. L'évaluation a lieu au tour 1 puis tous les 10 tours. La taille de lot vaut 32 par worker à l'entraînement et 256 au test. La régularisation est nulle. Le biais d'étiquettes est modéré, avec Dirichlet $\alpha = 1$. La graine vaut 0. Il n'existe pas de modèle global en décentralisé. Chaque point est la moyenne sur les modèles locaux des workers honnêtes.

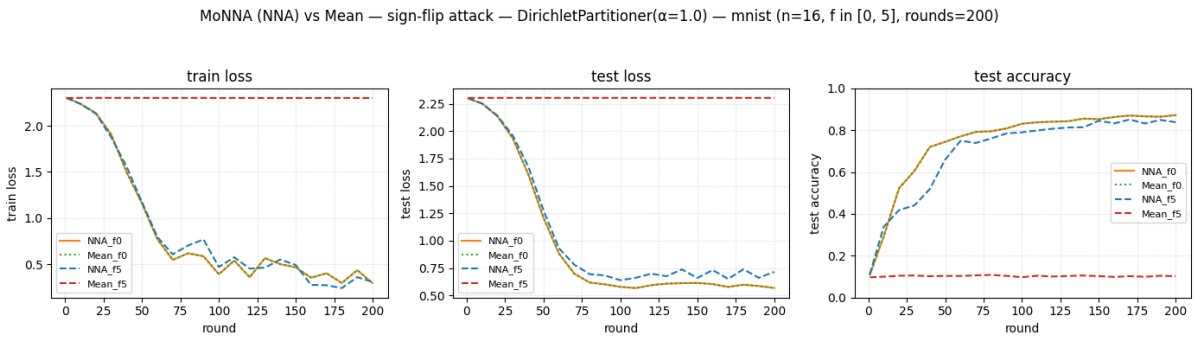

FIGURE 5.3 – Protocole MoNNA décentralisé sous `SignFlip`, avec MNIST, $n = 16$, $f = 5$, 200 tours, graine 0 et Dirichlet $\alpha = 1$. Les panneaux montrent les pertes d'entraînement et test ainsi que l'exactitude test. Sans attaque, `NNA` et `Average` convergent ensemble. Sous attaque, `NNA` rejoint environ 0,84 tandis que `Average` reste au niveau du hasard, autour de 0,10.

La figure 5.3 croise les deux règles avec et sans attaque. Sans byzantin, `NNA` et `Average` convergent ensemble vers $\approx 0{,}87$ d'exactitude. Sous attaque ($f = 5$, près d'un tiers de byzantins), `Average` ne décolle jamais : exactitude bloquée à $\approx 0{,}10$, le niveau du hasard sur dix classes, pertes figées à leur valeur initiale. `NNA`, elle, apprend, avec un retard initial marqué. L'écart est maximal vers les tours 30 à 45, où il approche vingt points, puis se résorbe partiellement, et elle finit autour de $\approx 0{,}84$, à trois points des courbes saines.

Le phénomène MoNNA est reproduit à échelle réduite : tant que les modèles byzantins restent minoritaires parmi les modèles reçus, le mélange aux plus proches voisins les isole et l'apprentissage suit sa trajectoire saine. La règle `Average` naïve, elle, est ramenée au hasard. C'est la version décentralisée de la leçon de la section 5.1 : sans sélection, pas de robustesse.

5.4 Que signifie reproduire fidèlement ?

« Fidèle » a dans ce mémoire un sens précis et borné. Précis signifie que chaque expérience reprend le cadre du protocole publié (données, modèle, règle d'agrégation, topologie) et

documente ses conditions d'entraînement. Le but n'est pas de recopier chaque réglage du papier, mais de retrouver le phénomène rapporté ; plusieurs réglages numériques diffèrent ainsi des publications (attaque, découpage, pas, taille de lot, nombre de tours). Les graines sont fixées, avec 42 pour les deux expériences centralisées et 0 pour la démo décentralisée. Chaque worker dispose de son propre générateur aléatoire. À code, graines et matériel fixés, deux exécutions sont bit à bit identiques. Borné signifie que la fidélité porte sur le cadre, les conditions documentées et le déterminisme. Elle ne porte pas sur l'identité numérique avec les courbes publiées.

Les écarts assumés sont de quatre ordres. D'abord le matériel, différent de celui des papiers, qui interdit à lui seul l'identité bit à bit des poids. Ensuite les allègements : `MultiKrum` à $m = 12$ au lieu de la borne 5, $\gamma = 280$ fixé au lieu de la recherche de frontière, petit MLP, moins de nœuds et 200 tours pour MoNNA. L'ensemble tient de l'ordre de quelques minutes de calcul sur machine ordinaire, le device étant auto détecté avec repli CPU. Puis trois variantes algorithmiques : `Bulyan` sélectionne $\theta = n - 2f - 2$ vecteurs au lieu de $n - 2f$, `Aksel` moyenne les $n - f$ gradients les plus proches du pivot, variante paramétrée par f, au lieu du seuil médian sans paramètre du papier, et `MultiKrum` moyenne les m plus petits scores de `Krum` en une seule passe, sans le retrait itératif du papier. Enfin, ce qui doit coïncider n'est pas la courbe au pixel près mais le phénomène : divergence contre convergence (5.1), admission contre filtrage (5.2), effondrement au hasard contre trajectoire saine (5.3). C'est à cette aune que les trois reproductions réussissent.

La promesse du chapitre 1 trouve ici sa forme opérationnelle : face à la crise de reproductibilité posée en introduction, Krum rejoue les protocoles et retrouve leurs phénomènes, à graines fixées, en quelques minutes et sur machine ordinaire. Chaque réglage est documenté. La reproductibilité n'est plus un effort héroïque a posteriori mais une propriété construite dans le logiciel, couche par couche depuis le chapitre 4.

Le chapitre suivant prend du recul : forces et compromis de ces choix, limites actuelles et comparaison aux autres librairies (chapitre 6).

Chapitre 6 Discussion et limites

6.1 Forces et compromis

6.2 Limites actuelles

6.3 Krum face aux autres librairies

	Krum papier compagnon	ByzFL arXiv 2025 [9]	FedLab **JMLR 2023** [23]	Blades IoTDI 2024 [12]	FL-Byz-Lib TIFS 2024 [14]	ByzPy (sans papier)
Primitives						
Règles d'agrégation	11	12	2	9	**36**	12
Attaques	5	9	—	9	**22**	7
Pré-agrégateurs	—	4	—	—	—	4
Wrapper de modèle	**zéro-copie**	flatten	—	—	flatten	flatten
Partitionnement	4	4	**9**	2	4	—
Simulations						
Topologie	**PS+P2P**	PS	PS	PS	PS	**PS+P2P**
Protocoles	**3 fidèles**	2 génériques	10 baselines	1 générique (FedAvg + DP)	1 générique	2 génériques
Orchestration						
Spécification	**API Python**	configs JSON	ServerHandler	YAML + Ray	CLI dataclass	DAG sched.
Collecte des métriques	**`Metric` typé**	fichiers	fichiers	fichiers	fichiers	fichiers
Relance après échec ou modification	✓	partiel	—	—	—	—
Parallélisme (balayage)	un processus par expérience	Pool (par config)	—	Ray Tune	—	ActorPool (DAG)
Déploiement	mono-hôte	mono-hôte	3 modes	Ray	—	5 backends

PS = serveur de paramètres ; P2P = pair-à-pair ; flatten = tampon aplati avec copies. Les comptes suivent la taxonomie propre à chaque librairie. Le gras signale le meilleur de sa catégorie. Le papier compagnon, qui décrit Krum, vise la rubrique Machine Learning Open Source Software du Journal of Machine Learning Research

Table 6.1 – Comparaison de Krum avec les librairies comparables (comptes d'août 2026).

Chapitre 7 Conclusion et perspectives

7.1 Synthèse des contributions

7.2 Bénéfices du stage

7.3 Travaux futurs

Bibliographie

[1] Gilad Baruch, Moran Baruch et Yoav Goldberg. « A Little Is Enough : Circumventing Defenses For Distributed Learning ». In : *Advances in Neural Information Processing Systems 32 (NeurIPS)*. T. 32. 2019. URL : https://arxiv.org/abs/1902.06156.

[2] Peva Blanchard, El-Mahdi El-Mhamdi, Rachid Guerraoui et Julien Stainer. « Machine Learning with Adversaries : Byzantine Tolerant Gradient Descent ». In : *Advances in Neural Information Processing Systems 30 (NIPS)*. T. 30. 2017. URL : https://arxiv.org/abs/1703.02757.

[3] Amine Boussetta, El-Mahdi El-Mhamdi, Rachid Guerraoui, Alexandre Maurer et Sébastien Rouault. « AKSEL : Fast Byzantine SGD ». In : *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. T. 184. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 8 :1-8 :16. DOI : 10.4230/LIPIcs.OPODIS.2020.8. URL : https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2020.8.

[4] ByzPy. *ByzPy : Byzantine-Robust Learning Runtime*. https://github.com/Byzpy/byzpy. 2025.

[5] Yudong Chen, Lili Su et Jiaming Xu. « Distributed statistical machine learning in adversarial settings : Byzantine gradient descent ». In : *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*. 2018, p. 96-96.

[6] CRYPTO-KU. *FL-Byzantine-Library*. https://github.com/CRYPTO-KU/FL-Byzantine-Library. Code for [14]. 2024.

[7] Sadegh Farhadkhani, Rachid Guerraoui, Nirupam Gupta, Lê Nguyên Hoang, Rafael Pinot et John Stephan. « Robust Collaborative Learning with Linear Gradient Overhead ». In : *Proceedings of the 40th International Conference on Machine Learning (ICML)*. T. 202. PMLR, 2023. URL : https://arxiv.org/abs/2209.10931.

[8] Xavier Glorot et Yoshua Bengio. « Understanding the Difficulty of Training Deep Feedforward Neural Networks ». In : *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*. T. 9. PMLR, 2010, p. 249-256. URL : https://proceedings.mlr.press/v9/glorot10a.html.

[9] Marc González, Rachid Guerraoui, Rafael Pinot, Geovani Rizk, John Stephan et François Taïani. *ByzFL : Research Framework for Robust Federated Learning*. arXiv :2505.24802. 2025. URL : https://arxiv.org/abs/2505.24802.

[10] Tzu-Ming Harry Hsu, Hang Qi et Matthew Brown. « Measuring the effects of non-identical data distribution for federated visual classification ». In : *arXiv preprint arXiv :1909.06335* (2019).

[11] Sai Praneeth Karimireddy, Lie He et Martin Jaggi. « Byzantine-robust learning on heterogeneous datasets via bucketing ». In : *arXiv preprint arXiv :2006.09365* (2020).

[12] Shenghui Li, Edith Ngai, Fanghua Ye, Li Ju, Tianru Zhang et Thiemo Voigt. « Blades : A Unified Benchmark Suite for Byzantine Attacks and Defenses in Federated Learning ». In : *Proceedings of the ACM/IEEE International Conference on Internet of Things Design and Implementation (IoTDI)*. 2024, p. 158-169. URL : https://arxiv.org/abs/2206.05359.

[13] El-Mahdi El-Mhamdi, Rachid Guerraoui et Sébastien Rouault. « The Hidden Vulnerability of Distributed Learning in Byzantium ». In : *Proceedings of the 35th International Conference on Machine Learning (ICML)*. T. 80. PMLR, 2018, p. 3521-3530. URL : https://arxiv.org/abs/1802.07927.

[14] Kerem Ozfatura, Emre Ozfatura, Alptekin Kupcu et Deniz Gündüz. « Byzantines Can Also Learn From History : Fall of Centered Clipping in Federated Learning ». In : *IEEE Transactions on Information Forensics and Security* 19 (2024), p. 2010-2022. DOI : 10.1109/TIFS.2023.3345171. URL : https://ieeexplore.ieee.org/document/10484776.

[15] Krishna Pillutla, Sham M. Kakade et Zaid Harchaoui. « Robust Aggregation for Federated Learning ». In : *IEEE Transactions on Signal Processing* 70 (2022), p. 1142-1154. DOI : 10.1109/TSP.2022.3153135. URL : https://arxiv.org/abs/1912.13445.

[16] Amit Portnoy, Yoav Tirosh et Danny Hendler. « Towards federated learning with byzantine-robust client weighting ». In : *Applied Sciences* 12.17 (2022), p. 8847.

[17] Herbert Robbins et Sutton Monro. « A Stochastic Approximation Method ». In : *The Annals of Mathematical Statistics* 22.3 (1951), p. 400-407. DOI : 10.1214/aoms/1177729586. URL : https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-3/A-Stochastic-Approximation-Method/10.1214/aoms/1177729586.full.

[18] Tianxiang Wang, Zhonglong Zheng et Feilong Lin. « Federated learning framework based on trimmed mean aggregation rules ». In : *Expert Systems with Applications* 270 (2025), p. 126354.

[19] Endre Weiszfeld. « Sur le point pour lequel la somme des distances de n points donnés est minimum ». In : *Tohoku Mathematical Journal, First Series* 43 (1937), p. 355-386.

[20] Cong Xie, Oluwasanmi Koyejo et Indranil Gupta. « Generalized Byzantine-tolerant SGD ». In : *arXiv preprint arXiv :1802.10116* (2018).

[21] Cong Xie, Sanmi Koyejo et Indranil Gupta. « Fall of Empires : Breaking Byzantine-tolerant SGD by Inner Product Manipulation ». In : *Proceedings of the 35th Uncertainty in Artificial Intelligence Conference (UAI)*. T. 115. PMLR, 2020, p. 261-270. URL : https://arxiv.org/abs/1903.03936.

[22] Dong Yin, Yudong Chen, Kannan Ramchandran et Peter Bartlett. « Byzantine-Robust Distributed Learning : Towards Optimal Statistical Rates ». In : *Proceedings of the 35th International Conference on Machine Learning (ICML)*. T. 80. PMLR, 2018, p. 5650-5659. URL : https://arxiv.org/abs/1803.01498.

[23] Dun Zeng, Siqi Liang, Xiangjing Hu, Hui Wang et Zenglin Xu. « FedLab : A Flexible Federated Learning Framework ». In : *Journal of Machine Learning Research* 24.100 (2023), p. 1-7. URL : https://www.jmlr.org/papers/v24/22-0440.html.

Chapitre A Catalogue des algorithmes implémentés

Ce catalogue détaille les briques de la couche *primitives* présentée à la section 4.1 : principe, principaux hyperparamètres tels que les signatures `aggregate`, `generate` et `partition` les exposent, garanties et publication d'origine. Les noms suivent la convention abrégée du chapitre 4.

A.1 Règles d'agrégation

Règle	Principe	Hyperparamètres	Garantie	Référence
`Average`	moyenne arithmétique des gradients	aucun	aucune (baseline non robuste)	—
`Median`	médiane coordonnée par coordonnée	aucun	$f \leq (n-1)/2$, bruit contrôlé	[22]
`TrimmedMean`	retire les f plus petites et f plus grandes valeurs par coordonnée, moyenne le reste	f	$f \leq (n-1)/2$, $\beta \geq f/n$	[22]
`Krum`	gradient minimisant la somme des distances au carré aux $n-f-2$ plus proches pairs (cas $m=1$)	n, f	$f \leq (n-3)/2$	[2]
`MultiKrum`	moyenne des m gradients aux plus petits scores de `Krum`	n, f, m ($=n-2f-3$ par défaut)	$f \leq (n-3)/2$; $m \leq n-2f-3$	[2]
`Bulyan`	sélection itérative de $\theta = n-2f-2$ vecteurs via `MultiKrum` ($\theta = n-2f$ dans le papier, écart conservé pour préserver les comparaisons antérieures), puis moyenne tronquée coordonnée par coordonnée	n, f, m (optionnel)	$f \leq (n-3)/4$	[13]
`Brute`	énumère les $\binom{n}{n-f}$ sous-ensembles, moyenne du plus compact	n, f	exacte ($f \leq (n-1)/2$)	[13]
`GeoMed`	médiane géométrique (minimiseur non contraint dans $\mathbb{R}^d$) par itérations de Weiszfeld lissées	n, f (ignoré), `nu` (0,1), `tol` (10^{-6}), `max_iter` (100)	$f \leq (n-1)/2$	[19, 15]
`Medoid`	vecteur soumis minimisant la somme des distances aux autres	n, f (ignoré)	aucune (baseline non résiliente)	[20, 2]
`Aksel`	pivot = médiane coordonnée par coordonnée, moyenne des $n-f$ plus proches, en $O(nd)$	f	$f \leq (n-1)/2$ (point de rupture optimal)	[3]
`NNA`	moyenne des k plus proches voisins d'un pivot ($k =$ `num_closest`)	`num_closest`, `pivot`	politique de l'appelant ($n-2f$ pour `MoNNA`)	[7]

TABLE A.1 – Les onze règles d'agrégation implémentées : principe, hyperparamètres de la signature, garantie et publication d'origine. n désigne le nombre total de workers, f le nombre de byzantins, m le nombre de gradients sélectionnés par `MultiKrum`, β la fraction élaguée de `TrimmedMean`. « Baseline » signale une règle incluse comme point de comparaison.

Trois lignes de ce tableau décrivent une variante de l'implémentation, distincte de leur papier. `Bulyan` sélectionne $\theta = n - 2f - 2$ vecteurs au lieu de $n - 2f$ et `Aksel` moyenne les $n - f$ gradients les plus proches du pivot, en consommant f, au lieu du seuil médian sans paramètre du papier. `MultiKrum` moyenne les m plus petits scores de `Krum` en une seule passe, sans le retrait itératif du papier : `Krum` est identique dans les deux cas ($m = 1$), et la borne $m \leq n - 2f - 3$ reste celle du théorème du papier, établi pour sa construction itérative. Ces écarts sont repris à la section 5.4.

A.2 Attaques

Attaque	Principe	Hyperparamètres	Référence
`SignFlip`	négation de la moyenne honnête coordonnée par coordonnée, même vecteur répété f fois	f, `scale` (1 par défaut)	[21]
`ALIE`	moyenne honnête $\pm z\sigma$ coordonnée par coordonnée (variante oracle : distribution honnête connue, écart-type population)	f, `z` (`"max"` par défaut), `direction` (`NEGATIVE` par défaut)	[1]
`Gaussian`	bruit gaussien isotrope, indépendant des gradients honnêtes	f, `mu` (0), `std` (200)	—
`FullGradient Negation`	$-\kappa\, g_{\text{full}}$, suppose la connaissance du gradient sur tout le jeu	f, `full_gradient`, `kappa` (100)	[13]
`SmallPertur bation`	$B(\gamma)$ = moyenne honnête $+\gamma E$, recherche du plus grand γ encore sélectionné par l'agrégateur cible	f, `aggregator`, n, `p`, `coordinate`, recherche (`p` 2, `coordinate None`, soit max-variance, $\gamma_{\max} = 10^6$, tol 10^{-3})	[13]

TABLE A.2 – Les cinq attaques byzantines implémentées : principe, hyperparamètres de la signature et publication d'origine.

A.3 Partitionneurs

Partitionneur	Principe	Hyperparamètres	Régime
`IidPartitioner`	mélange puis découpe en n parts égales (reste abandonné)	n, `seed` (42)	IID
`PerLabels` `Partitioner`	tri par étiquette, découpe en $n(N/n)^{\lambda}$ segments, arrondi à l'entier, distribués cycliquement	n, `lambda_`, `seed` (42)	non IID ($\lambda = 1$: IID)
`Dirichlet` `Partitioner`	proportions par classe $\sim$ Dirichlet(α) ; un worker peut ne recevoir aucun exemple	n, `alpha`, `seed` (42)	non IID
`Mixing` `Partitioner`	fraction $1-\gamma$ via `p1` et γ via `p2`, concaténées par worker	n, `p1`, `p2`, `gamma`, `p1_kwargs`, `p2_kwargs`, `seed` (42)	IID $\leftrightarrow$ non IID

Table A.3 – Les quatre partitionneurs implémentés : principe, hyperparamètres de la signature et régime de distribution. N désigne la taille du jeu de données.

Chapitre B Exemple de code étendu (orchestration)

Le listing ci-dessous déroule l'exemple canonique évoqué à la section 4.3 : un balayage croisant le nombre de workers, le nombre de byzantins, les règles d'agrégation et les attaques, avec la baseline honnête (`attack=None`). Les couples (n, f) sont choisis pour satisfaire les conditions de toutes les règles balayées, y compris $n \geq 4f + 3$ pour `Bulyan`. Chaque exécution déclare deux métriques typées (`loss` et `accuracy`), alimentées tous les dix pas à partir de `sim.evaluate()`, puis la récupération produit deux tables prêtes à analyser, fusionnées sur les paramètres du balayage et le pas.

```
from torchvision import datasets, transforms
from krum.orchestration import Metric, Orchestrator
from krum.primitives.aggregators.average import Average
from krum.primitives.aggregators.bulyan import Bulyan
from krum.primitives.aggregators.krum import Krum
from krum.primitives.attacks.alie import ALIEAttack
from krum.primitives.attacks.sign_flip import SignFlipAttack
from krum.primitives.data_partitioners.iid import IidPartitioner
from krum.primitives.models.mlp import Krum2017MLPMnist
from krum.simulations.centralised.krum_nips_2017 import KrumSimulation


def experiment(n, f, aggregator, attack, seed):
    """Une exécution : partition, simulation et collecte des métriques."""
    train_set = datasets.MNIST("data", train=True, download=True,
                               transform=transforms.ToTensor())
    test_set = datasets.MNIST("data", train=False, download=True,
                              transform=transforms.ToTensor())
    worker_datasets = IidPartitioner.partition(train_set, n=n, seed=seed)
    sim = KrumSimulation(
        model_cls=Krum2017MLPMnist, train_datasets=worker_datasets,
        test_set=test_set, aggregator=aggregator, attack=attack,
        n=n, f=f, rounds=100, batch_size=32, lr=0.1, seed=seed,
    )
    sim.setup()
    loss = Metric("loss", dtype=float)
    accuracy = Metric("accuracy", dtype=float)
    for step in range(100):
        sim.step()
        if step % 10 == 0:
```

```
            test_loss, test_accuracy = sim.evaluate()
            loss.push(step, test_loss, skip_if_exists=True)
            accuracy.push(step, test_accuracy, skip_if_exists=True)


orch = Orchestrator("krum_byzantine_study")
for n, f in [(13, 2), (25, 5)]:
    for aggregator in [Average, Krum, Bulyan]:
        # attack=None : baseline honnête, sans gradients byzantins.
        for attack in [ALIEAttack, SignFlipAttack, None]:
            orch.run(experiment, n=n, f=f, aggregator=aggregator,
                    attack=attack, seed=42)

loss_df = orch.get("loss").to_pandas()
acc_df = orch.get("accuracy").to_pandas()

# Filtrage : Krum sous attaque ALIE.
krum_alie = loss_df[(loss_df["aggregator"] == Krum)
                  & (loss_df["attack"] == ALIEAttack)]

# Perte moyenne par (règle, attaque, pas).
mean_loss = (loss_df.groupby(["aggregator", "attack", "step"])
            ["value"].mean().reset_index())

# Fusion des deux tables sur les paramètres et le pas.
merged = (loss_df.rename(columns={"value": "loss"}).merge(
    acc_df.rename(columns={"value": "accuracy"}),
    on=["step", "n", "f", "aggregator", "attack", "seed"]))
```

Listing B.1 – Balayage canonique : croisement (n, f), règles d'agrégation et attaques, collecte et fusion des métriques.

Chaque table retournée par `to_pandas()` porte les paramètres du balayage en colonnes, puis `step` et `value` : le filtrage, l'agrégation (`groupby`) et la fusion (`merge`) ci-dessus sont donc de simples opérations pandas, sans suivi manuel des paramètres. Relancer le script après une panne ou une modification ne rejoue que les expériences nouvelles, modifiées ou ratées.

Chapitre C Architectures de modèles prédéfinis

Les six architectures embarquées, dont cinq reprises des publications reproduites, utilisables telles quelles dans les simulations ou comme baselines d'expériences propres. Toutes s'instancient sans argument et s'enrobent dans le wrapper `Model` pour les vues plates (section 4.1).

Modèle	Architecture	Origine
`Krum2017MLPMnist`	perceptron multicouche $784 \to 100 \to 10$ (ReLU), $\approx 8 \times 10^4$ paramètres	[2] (MNIST)
`Krum2017MLP Spambase`	perceptron multicouche $57 \to 20 \to 20 \to 2$ (ReLU)	[2] (Spambase)
`Krum2017CNN`	2 convolutions (16 et 64 filtres) + 3 couches denses (384, 192, 10), ReLU et max-pooling	[13] (CIFAR-10)
`Monna2023Small Mnist`	perceptron à une couche cachée $784 \to 128 \to 10$, variante légère pour CPU, hors papier	—
`Monna2023CNNMnist`	`C(20)-R-M-C(20)-R-M-L(500)-R-L(10)`, convolutions à noyaux 5×5	[7] (MNIST, app. D.2)
`Monna2023CNN Cifar10`	4 convolutions (64, 64, 128, 128, noyaux 5×5) + normalisation par lots + 2 max-pooling + dropout 0,25 + 2 couches denses (128, 10)	[7] (CIFAR-10, app. D.2)

TABLE C.1 – Les six architectures standards embarquées. R = ReLU, M = max-pooling, L = couche dense, C(k) = convolution à k filtres, app. = appendice. La normalisation par lots recentre et réduit chaque lot ; le dropout éteint aléatoirement une fraction des activations pendant l'entraînement.

Trois écarts assumés avec les publications méritent mention. Les tailles cachées du MLP Spambase (20, 20) ne sont pas précisées dans le papier de 2017 et reprennent les implémentations de référence du laboratoire. Les deux CNN du protocole décentralisé renvoient des scores bruts, dits logits, sans le log-softmax final du papier. Le log-softmax convertit ces scores en log-probabilités normalisées. Les renvoyer bruts permet un appariement direct avec `CrossEntropyLoss`, qui applique elle-même cette normalisation. `Monna2023SmallMnist` est une variante légère hors papier, prévue pour les expériences rapides sur CPU.