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 tiens à remercier El-Mahdi El-Mhamdi, mon maître de stage au CMAP, qui a dirigé ce travail scientifiquement et suivi mon stage au quotidien. Tout au long de ces mois, il m'a fait confiance, de la conception de la librairie jusqu'à la rédaction du papier qui l'accompagne. Je le remercie aussi de prolonger cette confiance en me proposant une thèse sous sa direction.

Je remercie également Peva Blanchard et Sébastien Rouault pour leurs conseils, aussi bien d'organisation que de recherche. Ils se sont impliqués dans ce projet, hors de leurs obligations, et leurs relectures et discussions ont beaucoup apporté à la librairie.

Je remercie Katia Meziani, qui m'a mis en relation avec El-Mahdi El-Mhamdi dans le cadre de mon master. Sans elle, ce stage n'aurait pas pu avoir lieu. Elle m'a ensuite soutenu dans la poursuite de mon parcours en thèse.

Je remercie le groupe Robust Learning pour son accueil au CMAP. J'y ai rencontré des collègues passionnés avec qui j'ai aimé travailler au quotidien, et leur présence quotidienne a rendu ces mois particulièrement agréables.

Je remercie l'équipe pédagogique du master ISF de l'Université Paris-Dauphine – PSL, dont la formation m'a préparé à ce stage. Je remercie enfin les membres du jury d'avoir accepté d'évaluer ce mémoire.

Je remercie ma famille et ma compagne pour leur soutien pendant ces mois.

Table des matières

1 Introduction **1**

1.1 Contexte 1

1.2 Problématique 2

1.3 Contribution : la librairie Krum 2

2 Cadre du stage **4**

2.1 Accueil et mission 4

2.2 Déroulement du stage 5

3 État de l'art **8**

3.1 Terminologie minimale 8

3.1.1 Acteurs et protocoles 8

3.1.2 Données et modèle 9

3.1.3 Optimisation 9

3.2 Apprentissage distribué byzantin-résilient 10

3.3 Règles d'agrégation de gradients 12

3.3.1 Sélection à base de distances 12

3.3.2 Filtrage coordonnée par coordonnée 13

3.3.3 Voisinage 14

3.3.4 Méthodes géométriques et énumératives 14

3.4 Stratégies d'attaque byzantines 15

3.4.1 Attaques génériques 15

3.4.2 Attaques de la vulnérabilité cachée 16

3.5 Librairies existantes : des taxonomies incompatibles 16

4 Conception de Krum **18**

4.1 Primitives sans état 18

4.1.1 Un pattern de conception unique 18

4.1.2 Extensibilité 19

4.1.3 Règles d'agrégation 19

4.1.4 Attaques 21

4.1.5 Partitionneurs de données 21

4.1.6 Modèles : wrapper zéro-copie et architectures de la littérature . . . 22

4.2 Simulations fidèles aux protocoles publiés 22

4.3 Orchestration d'expériences . . . 23

5 Validation expérimentale 25

5.1 Reproduction de [BEGS17] (NeurIPS 2017) : MultiKrum contre Average . 25

5.1.1 Paramétrage . . . 25

5.1.2 Résultats . . . 25

5.2 Reproduction de [EGR18] (ICML 2018) : MultiKrum contre Bulyan . . . 26

5.2.1 Paramétrage . . . 26

5.2.2 Résultats . . . 27

5.3 Étude de cas à échelle réduite de [FGGHPS23] (ICML 2023) : MoNNA décentralisé (NNA contre moyenne) . . . 28

5.3.1 Paramétrage . . . 28

5.3.2 Résultats . . . 29

6 Discussion et limites 30

6.1 Forces et compromis . . . 30

6.2 Limites actuelles . . . 31

6.3 Krum face aux autres librairies . . . 32

7 Conclusion et perspectives 36

7.1 Synthèse des contributions . . . 36

7.2 Acquis personnels du stage . . . 37

7.3 Futures directions . . . 38

7.4 Conclusion générale . . . 39

Bibliographie 40

A Catalogue des algorithmes implémentés 44

A.1 Règles d'agrégation . . . 45

A.2 Attaques . . . 46

A.3 Partitionneurs . . . 47

B Exemple de code étendu (orchestration) 48

C Architectures de modèles pré-définis 50

Résumé

En apprentissage distribué, une machine défaillante ou malveillante peut perturber l'entraînement en envoyant des mises à jour arbitraires. Le modèle de menace byzantin décrit ce type de comportement. Les règles d'agrégation robustes visent à en limiter l'influence. Reproduire les résultats publiés reste toutefois difficile, car les codes de recherche diffèrent d'un article à l'autre. Mon stage au Centre de Mathématiques Appliquées de l'École Polytechnique vise à développer une librairie réutilisable pour étudier ces règles.

La librairie Krum, développée en Python, utilise la bibliothèque de calcul PyTorch. Elle réunit onze règles d'agrégation, cinq attaques et quatre méthodes de répartition des données entre participants. Un composant donne accès aux paramètres des modèles sans les copier. Un orchestrateur gère les campagnes d'expériences, vérifie le type des métriques collectées et reprend les calculs échoués ou périmés.

Trois expériences adaptées de protocoles publiés en retrouvent les phénomènes principaux. Sur Spambase, `MultiKrum` résiste à l'inversion du signe des gradients, tandis que la moyenne diverge. Sur MNIST, `Bulyan` filtre une perturbation ciblée qui dégrade les performances de `Krum`. Sans serveur central, le mélange de modèles `NNA` du protocole `MoNNA` poursuit l'apprentissage sous attaque, alors que la moyenne reste au niveau du hasard. Chaque expérience s'exécute en moins de trois minutes sur processeur (CPU). Les adaptations sont documentées. À code et matériel fixés, les mêmes graines, qui déterminent les tirages aléatoires, permettent de répéter les expériences à l'identique.

Ce travail montre comment traduire des formulations mathématiques en composants testés et documentés, puis les évaluer expérimentalement.

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

Chapitre 1 Introduction

En apprentissage distribué, plusieurs machines contribuent à entraîner un même modèle. Certaines peuvent toutefois envoyer des mises à jour qui détournent l'entraînement. Ce mémoire étudie les méthodes qui limitent cet effet et présente Krum, une librairie développée pendant le stage pour les expérimenter.

Ce mémoire présente tout d'abord le contexte et la mission du stage, puis les notions scientifiques utiles à la lecture du mémoire. Il décrit ensuite la conception de la librairie, les expériences menées pour la valider et les limites des résultats.

1.1 Contexte

L'apprentissage supervisé peut se trouver partout : le clavier qui suggère la suite d'un message, un hôpital qui entraîne des modèles avec d'autres établissements. Le point commun est la contrainte : les données restent sur place, pour respecter la vie privée. Le clavier Gboard de Google apprend ainsi sans envoyer le texte, et les hôpitaux partagent un modèle sans échanger les dossiers des patients. Les modèles et les jeux de données grossissent aussi, au-delà de ce qu'une seule machine peut entraîner. L'entraînement se répartit donc sur plusieurs machines : c'est l'apprentissage distribué. Chaque machine, dite worker, calcule sur sa part des données un gradient, c'est-à-dire la direction qui réduit l'erreur du modèle. La descente de gradient stochastique organise ce travail en tours : les workers envoient leur gradient, un serveur de paramètres agrège les vecteurs reçus, les combinant en une mise à jour unique par moyenne, puis diffuse le modèle mis à jour. Cependant, en variante décentralisée, les workers échangent leurs modèles de pair-à-pair, chaque machine dialoguant directement avec un sous-ensemble des autres, sans serveur central. Les deux régimes sont décrits au chapitre 3 (section 3.2).

Le serveur fait confiance à chaque gradient reçu. Pourtant, un worker défaillant ou malveillant peut envoyer n'importe quel vecteur. Le modèle de menace, dit byzantin, autorise ce comportement arbitraire, du nom du problème des généraux byzantins, où des traîtres falsifient des messages (LSP82). Le chapitre 3 pose formellement ce modèle (section 3.1). En réponse, le serveur remplace la moyenne par une règle d'agrégation robuste, qui sélectionne ou filtre les vecteurs reçus pour limiter l'influence des contributions malveillantes (BEGS17 ; CSX18).

Trois familles d'attaques visent cet entraînement distribué. La première, l'empoisonnement

des données, corrompt le jeu d'entraînement lui-même. La deuxième frappe le serveur central, point de défaillance unique dont la compromission fait s'effondrer le protocole. La troisième famille est l'attaque byzantine en apprentissage distribué, où les workers malveillants envoient des gradients forgés, c'est-à-dire fabriqués de toutes pièces. C'est ce dernier cas que couvre la librairie du stage. L'empoisonnement des données et l'attaque du serveur central restent hors de son périmètre.

1.2 Problématique

Depuis 2017 (BEGS17), de nombreuses règles d'agrégation et garanties de convergence contre la menace byzantine ont émergé, ainsi que des attaques pour les mettre à l'épreuve (BEGS17 ; YCRB18 ; XKG18 ; BEGMR21 ; PKH22 ; BBG19 ; EGR18 ; CSX18). Pourtant, reproduire ces résultats reste difficile : chaque papier fait des choix d'implémentation distincts, rarement isolés en composants réutilisables. Le code des règles publiées vit dans des dépôts de recherche, sans interface stable ni tests. Implémenter une règle à la main, ou reprendre un code existant, est donc une tâche pénible pour un autre chercheur ; les reproductions menées de zéro au début du stage l'ont vérifié (chapitre 2). Les protocoles d'évaluation varient aussi d'un papier à l'autre, sur les données, les modèles, les pas ou les attaques. Quant aux librairies existantes, chacune ne couvre qu'une partie de cette chaîne (GGPRST25 ; ZLHWX23 ; LNYJZV24 ; CRY24 ; Byz25). La comparaison détaillée est reportée au chapitre 6 (section 6.3).

Un protocole d'évaluation désigne l'ensemble des conditions d'une expérience : données, modèle, règle d'agrégation, topologie.

Comment concevoir une librairie permettant de reproduire toute la chaîne expérimentale de l'agrégation robuste face aux fautes byzantines, depuis les composants testés jusqu'aux campagnes d'expériences fondées sur des protocoles publiés ?

1.3 Contribution : la librairie Krum

La réponse apportée par le stage à cette problématique est la librairie Krum (chapitre 4). Elle est distribuée sous licence MIT, une licence libre permissive. Elle est installable en une commande depuis PyPI, le dépôt public de paquets Python. Elle est construite sur PyTorch, la librairie de calcul qui entraîne la plupart des modèles d'apprentissage. Le code et les tutoriels sont en accès libre [1].

La librairie s'organise en trois couches. La première, *les primitives*, réunit onze règles

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

d'agrégation, cinq attaques et quatre méthodes de partitionnement des données. Ces composants s'utilisent comme de simples fonctions. Cette couche comprend aussi le composant `Model`, appelé wrapper, qui donne accès aux paramètres du modèle sans les copier (section 4.1.6). La deuxième, *les simulations*, reprend les mécanismes de trois protocoles publiés, avec des adaptations documentées. La troisième, *l'orchestration*, déclare les balayages en code Python. Un balayage désigne un ensemble d'expériences indépendantes qui font varier les paramètres d'un protocole. Cette couche collecte aussi des métriques typées, c'est-à-dire des relevés dont le type est vérifié à chaque ajout. L'orchestrateur accepte les fonctions d'expérience de l'utilisateur, produit des tables prêtes à analyser, et ne relance que les expériences nouvelles, modifiées ou ratées (section 4.3).

La promesse de reproductibilité repose sur ces trois couches : Krum rejoue les protocoles et retrouve leurs phénomènes, à graines fixées, en moins de trois minutes par expérience et sur machine ordinaire. Les graines fixent les tirages aléatoires, de sorte que deux exécutions suivent la même trajectoire (section 6.1). Un papier compagnon, en cours de rédaction, décrit la librairie selon ces trois couches et la compare aux autres librairies (chapitre 6).

Dans ce rapport, vous trouverez le chapitre 2 qui pose le cadre du stage, et le chapitre 3 les bases scientifiques. Le chapitre 4 décrit la librairie en trois couches, validée au chapitre 5 sur trois protocoles publiés. La discussion (chapitre 6) en examine les forces et les limites face aux cinq systèmes comparables, avant le bilan du chapitre 7. Les annexes A à C détaillent le catalogue, un exemple d'orchestration et les architectures.

Chapitre 2 Cadre du stage

Le stage s'est déroulé au Centre de Mathématiques Appliquées (CMAP), au sein du groupe Robust Learning (groupe de recherche sur l'apprentissage robuste). La mission consistait à développer une librairie pour étudier comment combiner les calculs de plusieurs machines lorsque certaines envoient des informations erronées. Ce chapitre présente le laboratoire, l'équipe et les étapes du travail.

2.1 Accueil et mission

Le CMAP et ses axes de recherche. Mon stage s'est déroulé au CMAP de l'École Polytechnique, unité mixte de recherche (UMR 7641) du CNRS et de l'École. Le Centre couvre l'optimisation, les probabilités, les statistiques et le calcul scientifique, avec une ouverture croissante vers l'apprentissage automatique. Mon stage s'inscrit dans ce dernier domaine : l'apprentissage distribué robuste, à la frontière entre optimisation et systèmes distribués. Il conclut mon master Ingénierie Statistique et Financière de l'Université Paris-Dauphine – PSL.

Équipe d'encadrement. Mon maître de stage est El-Mahdi El-Mhamdi, chercheur au CMAP. Il est coauteur de deux articles fondateurs du sujet. Le premier, [BEGS17], introduit une règle d'agrégation byzantine-résiliente. Le second, [EGR18], présente une attaque par vulnérabilité cachée, qui fait passer un vecteur malicieux pour un gradient honnête (chapitre 3, sections 3.3 et 3.4).

Objectifs du stage. 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 devait réunir les agrégateurs, les attaques et les outils nécessaires à l'orchestration de milliers d'expériences. Cet ensemble devait reposer sur des primitives testées (section 4.1). Le chapitre 4 décrit le fonctionnement de l'orchestrateur (section 4.3). Une fois la librairie développée, trois protocoles publiés ont été reproduits pour montrer qu'elle pouvait servir à plusieurs configurations expérimentales (chapitre 5). Il s'agit des protocoles de [BEGS17], de [EGR18] et de [FGGHPS23] (section 4.2).

2.2 Déroulement du stage

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

Une équipe recherche-produit de quatre. Peva Blanchard, coauteur du papier `Krum` (BEGS17), est responsable produit (Product Owner). Il définit la vision du produit, priorise les fonctionnalités et tient le carnet de tickets, c'est-à-dire la liste des tâches à réaliser. Sébastien Rouault, responsable technique (Tech Lead : architecture et revues de code), est coauteur du papier `Bulyan` (EGR18). Son code était déjà repris tel quel dans les codes de recherche publiés : agrégateurs, attaques et composant d'accès aux paramètres sans copie (wrapper, section 4.1). Mohamad-Ammar Said et moi-même assurions le développement.

Pendant mon stage, j'ai pris en charge les agrégateurs et le wrapper de modèle `Model`, qui donne accès aux paramètres sans les copier. Mon travail couvrait aussi les réseaux de neurones convolutifs (CNN), les perceptrons multicouches (MLP) et l'ensemble de la partie centralisée de la librairie.

J'ai rédigé une très grande partie de la documentation de la librairie pour faciliter son utilisation et la reproduction des expériences, ainsi que les tests associés aux composants développés. Afin d'automatiser les vérifications et la publication du logiciel, j'ai écrit les fichiers de configuration de l'intégration et du déploiement continus (CI/CD).

J'ai aussi choisi les librairies retenues pour la comparaison et rédigé les notes qui en documentent les caractéristiques. J'ai réalisé les décomptes utilisés dans le tableau 6.1, présenté au chapitre 6.

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. Cette organisation m'a permis d'apprendre à prioriser les fonctionnalités, à justifier les choix d'architecture et à soumettre mes contributions à une revue de code. La conclusion revient sur ces acquis personnels (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 réunissant cinq personnes, avec El-Mahdi El-Mhamdi, traitait l'organisation générale et la revue des demandes de fusion. Une fois le code stabilisé, elle est passée à 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.

Jalons. Le calendrier se lit dans le dépôt KRUM : plus de 650 commits et 29 demandes de fusion déposées au total au 5 octobre 2026, dont 26 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 semblait clair : les chercheurs peinaient à intégrer ce code. Pour le réutiliser, ils copiaient dans leurs propres dépôts les agrégateurs, les attaques et le wrapper, le composant qui donne accès aux paramètres sans les copier. La liste des idées de la librairie est née de cet audit. 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, mené de la mi-août à fin septembre) et les corrections de fidélité (section 6.1). En octobre, l'orchestration est réécrite en version 2 (n° 66). Les résultats sont stockés par job, et seuls les jobs périmés sont rejoués. La librairie sort alors en version 0.9.0, publiée sur PyPI (n° 67). Parallèlement, la rédaction du papier compagnon commence et se poursuit en octobre. 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.

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

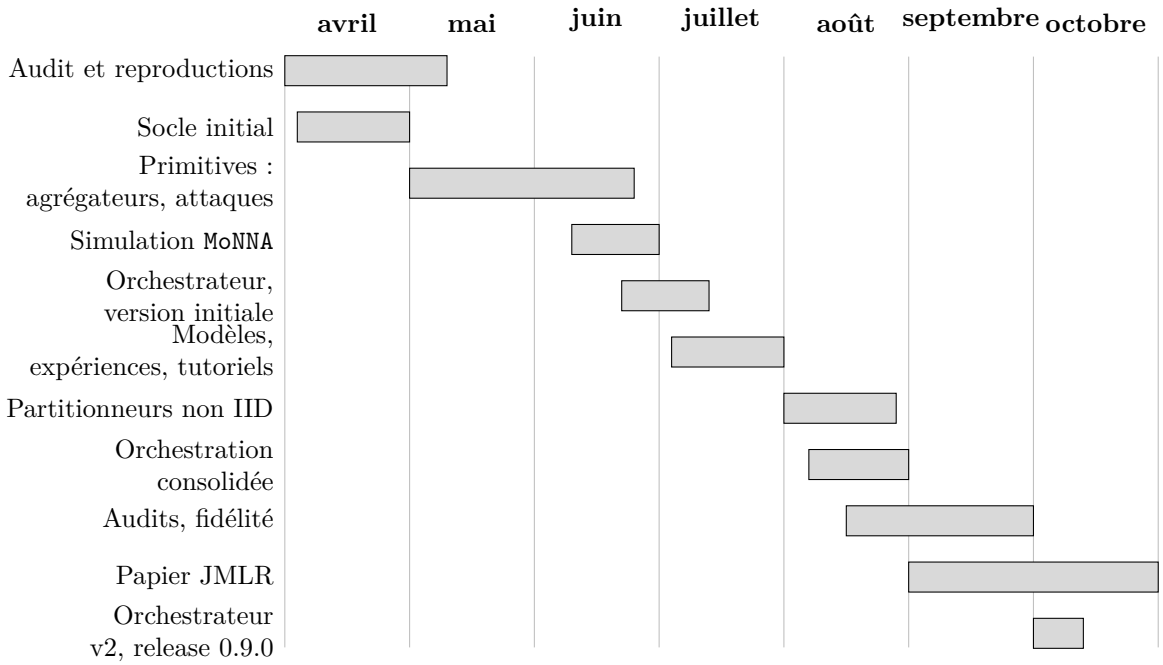


FIGURE 2.1 – Chronologie du stage, d'avril à octobre 2026.

Chapitre 3 État de l'art

Plusieurs machines peuvent participer à l'entraînement d'un modèle, et certaines peuvent envoyer des mises à jour malveillantes. Ce chapitre explique le fonctionnement de l'apprentissage distribué et présente les méthodes étudiées pour limiter l'effet de ces mises à jour.

3.1 Terminologie minimale

Cette section définit le vocabulaire nécessaire à la lecture de la suite, regroupé en trois blocs : les acteurs et les protocoles, les données et le modèle, l'optimisation.

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 et f celui des malveillants.

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 malicieux reçus.

Byzantin. Le modèle de menace byzantin autorise un comportement arbitraire : écart au protocole, messages contradictoires, action malveillante. N'importe quel worker peut être concerné. On suppose toutefois qu'au plus f workers peuvent adopter ce comportement parmi les n participants. Un worker honnête est celui qui suit exactement le protocole

prescrit. Le worker malveillant exploite cette liberté et soumet des gradients malveillants, dits aussi malicieux, malhonnêtes ou adversariaux, par opposition aux gradients honnêtes des workers honnêtes. Le nom vient du problème des généraux byzantins (LSP82) : des généraux de l'empire byzantin doivent convenir d'un plan commun en ne communiquant que par messagers, alors que des traîtres parmi eux peuvent falsifier les messages.

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, Multi-Layer Perceptron) empile des couches linéaires suivies d'activations non linéaires.

CNN. Un réseau de neurones convolutif (CNN, Convolutive Neural Network) applique des filtres convolutifs partagés en chaque point de l'image, puis combine les motifs extraits par des couches linéaires. Il est adapté aux images.

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 (GB10) pour stabiliser le début de l'entraînement.

Dirichlet. Dirichlet tire les proportions de chaque classe par worker selon une loi de Dirichlet (HQB19), et un α petit donne un déséquilibre fort.

3.1.3 Optimisation

Conditions de Robbins-Monro. Un pas décroissant satisfait les conditions de Robbins et Monro (RM51) lorsque la somme des pas diverge et que la somme de leurs carrés converge. 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 de [EGR18].

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

3.2 Apprentissage distribué byzantin-résilient

Le modèle de menace est simple. La descente de gradient suit pas à pas la direction qui fait baisser la perte pour approcher un minimum. Chaque worker calcule cette direction sur sa part locale. Les workers honnêtes suivent le protocole tandis que les workers malveillants proposent des vecteurs arbitraires pour détourner la trajectoire. Le pire scénario est le plus destructeur, non le plus complexe. Il combine deux capacités supplémentaires. La **collusion** permet aux malveillants de communiquer entre eux et de construire leurs vecteurs ensemble. L'**omniscience** leur donne accès à l'état complet du système, y compris aux gradients honnêtes, au sens de [EGR18]. Chaque attaquant ne soumet toutefois qu'un vecteur par tour. Les garanties visent ce pire cas : une règle qui y résiste résiste a fortiori à un attaquant isolé ou moins informé. 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, comme au lemme 1 de [BEGS17] ; le problème vient alors du fait qu'un seul gradient malveillant non borné suffit à déplacer l'agrégat arbitrairement loin, si bien que la moyenne n'offre aucune garantie. L'architecture est résumée figure 3.1.

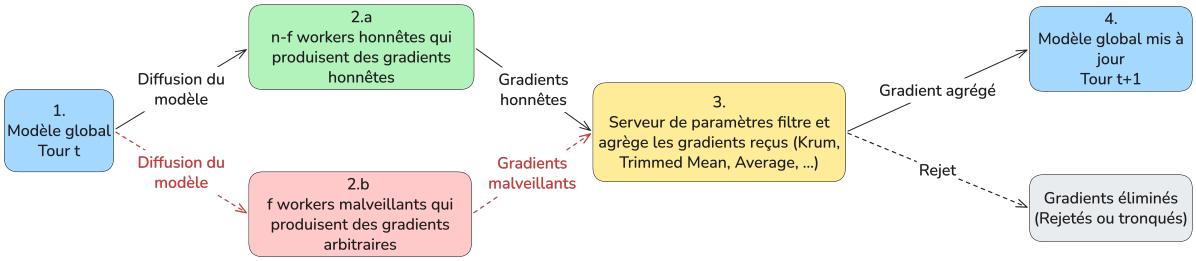


FIGURE 3.1 – Architecture de l'apprentissage fédéré centralisé en présence byzantine, du tour t au tour $t+1$. Les $n-f$ workers honnêtes calculent des gradients sur leurs données locales et les envoient au serveur de paramètres. Les f workers malveillants voient le modèle et envoient des vecteurs arbitraires,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

En opposition, le protocole décentralisé supprime le serveur central. 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 adversariaux par tour. C'est le régime du gossip (section 3.1), repris par le protocole `MoNNA` de [FGGHPS23]. La figure 3.2 illustre ce tour décentralisé.

D'autre part, la résilience (α, f) de [BEGS17] donne un sens précis à « robuste ». Elle demande deux choses à la règle d'agrégation. D'abord, son espérance doit pointer dans

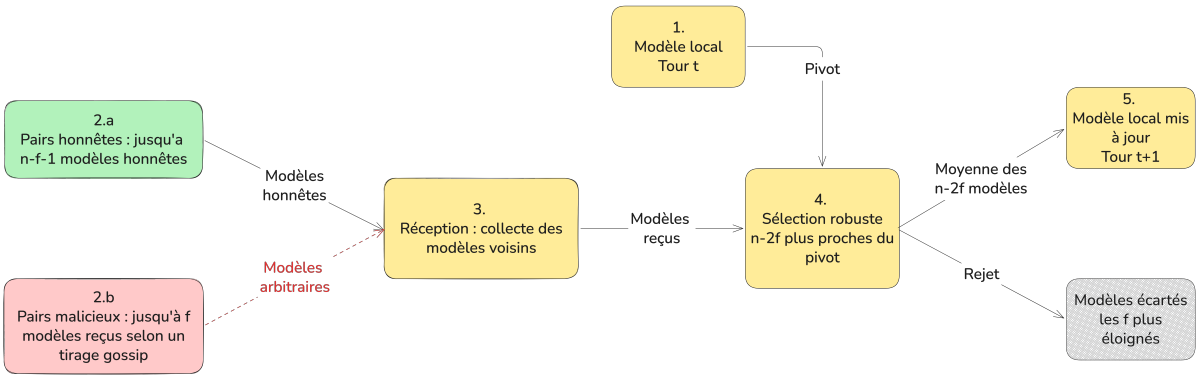


FIGURE 3.2 – Tour décentralisé, vu d'un worker honnête, du tour t au tour $t+1$. Le worker collecte les modèles de ses pairs, dont 0 à f modèles malveillants tirés au hasard selon le gossip. Il conserve les $n-2f$ modèles les plus proches de son modèle local, pris comme pivot, écarte les f plus éloignés,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

la 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) [BEGS17])**.** Soient $0 \le \alpha < \pi/2$ et $0 \le f \le n$ un entier. Soient $\boldsymbol{V}_1, \dots, \boldsymbol{V}_n$ 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 $\boldsymbol{B}_1, \dots, \boldsymbol{B}_f$ 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 $1 \le j_1 < \cdots < j_f \le n$, le vecteur

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

vérifie les deux conditions suivantes :

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

(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 workers malveillants, pourvu que le bruit honnête reste petit devant le gradient. L'article [CSX18] établit en ce sens la convergence de la descente de gradient byzantine sous de telles règles.

Ce mémoire distingue deux degrés de défaillance :

1. La définition d'origine (BEGS17) 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.

— une règle est « non robuste » lorsqu'elle n'offre aucune garantie en présence de workers malveillants, comme la moyenne simple ;
— une règle est « non résiliente » lorsqu'elle échoue au seul critère (α, f) qui précède, comme le médoïde de la définition 3.3.8.

Les règles médianes de la section 3.3 ont pour leur part des garanties statistiques, sous majorité stricte, d'une autre nature que cette condition géométrique.

Les chapitres suivants utilisent des pas fixes puis décroissants. En régime décroissant, la convergence suppose un pas au sens de Robbins et Monro (RM51) : la somme des pas diverge et la somme de leurs carrés converge. Sans elles, la garantie géométrique de la règle ne produit aucune convergence. L'article [EGR18] utilise 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 face aux workers malveillants, elle sert de baseline, c'est-à-dire de référence témoin, dans tout le mémoire.

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 [BEGS17])**.** 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 $2f + 2 < n$.

Il s'agit ici de la règle `Krum`, à ne pas confondre avec la librairie Krum du même nom décrite au chapitre 4.

Le coût est $O(n^2(d + \log n))$.

Définition 3.3.2 (MultiKrum [BEGS17])**.** 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

département 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 [EGR18])**.** Soit A une règle (α, f)-résiliente, avec $n \geq 4f + 3$ gradients reçus. $\texttt{Bulyan}(A)$ 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. $\texttt{Bulyan}(A)$ 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 [YCRB18])**.** 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 [YCRB18])**.** Pour $\beta \in [0, 1/2)$ tel que βn soit un entier, et des vecteurs $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$ 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 triant les n valeurs k-ièmes puis en retirant les βn plus petites et les βn plus grandes. Avec $\beta = f/n$, la règle retire exactement f valeurs de chaque côté et moyenne les $n - 2f$ restantes ; $\beta = 0$ redonne la moyenne. C'est la forme implémentée dans Krum, voir l'annexe A.

La garantie de [YCRB18] 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 malveillante 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. Des frameworks fédérés récents bâtissent encore sur la moyenne tronquée (WZL25).

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 [FGGHPS23])**.** 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`, qui prend le modèle local comme pivot, chaque nœud agrège ainsi les $n-f-1$ vecteurs reçus avec le sien en éliminant les f plus éloignés.

3.3.4 Méthodes géométriques et énumératives

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

Définition 3.3.7 (Médiane géométrique [Wei37 ; PKH22])**.** 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 workers malveillants tolérable, vaut $1/2$, la valeur optimale : elle résiste tant que $f < n/2$. Sa garantie est d'ordre statistique.

Définition 3.3.8 (Médoïde [XKG18])**.** Le médoïde restreint la médiane géométrique aux vecteurs soumis : il renvoie celui qui minimise la somme des distances aux autres,

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

Le médoïde choisit sa sortie parmi les vecteurs soumis, alors que la médiane géométrique peut se trouver en dehors de cet ensemble. Des contributions malveillantes coordonnées peuvent influencer ce choix. Il ne satisfait pas le critère (α, f).

Définition 3.3.9 (AKSEL [BEGMR21])**.** 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 [EGR18])**.** 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. Elles restent dans les limites du pire cas défini à la section 3.2, collusion et omniscience. Les attaques se rangent en deux groupes : génériques et attaques de la vulnérabilité cachée. Dans tous les cas, le but est le même : faire sélectionner un vecteur mal orienté tout en ressemblant aux gradients honnêtes.

3.4.1 Attaques génériques

Définition 3.4.1 (SignFlip [XKG20])**.** Dans la classe des manipulations de produit scalaire de [XKG20], l'instanciation locale `SignFlip` renvoie la négation de la moyenne honnête, répétée f fois : tous les workers malveillants envoient $-\texttt{scale} \cdot \bar{g}$, 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 [BBG19])**.** 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 contributions honnêtes nécessaires pour former une majorité avec les contributions malveillantes. Avec Φ la fonction de répartition normale centrée réduite, $z^{\max} = \max\{z : \Phi(z) < (n-s)/n\}$. $z^{\max}$ est donc le quantile d'ordre $(n-s)/n$ de la normale centrée réduite : par coordonnée, environ s gradients honnêtes seulement s'écartent de μ_j de plus de $z^{\max}\sigma_j$. Les copies malveillantes restent ainsi proches des valeurs honnêtes, ce qui les rend moins faciles à repérer comme valeurs aberrantes. ALIE fait envoyer à tous les workers malveillants $(p_{\text{mal}})_j = \mu_j \pm z^{\max}\sigma_j$ 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. L'attaquant est dit agnostique : il n'observe ni les gradients honnêtes ni leurs statistiques, contrairement à l'attaquant oracle ou omniscient. C'est la menace la plus faible, et elle sert de baseline dans ce mémoire.

3.4.2 Attaques de la vulnérabilité cachée

Définition 3.4.4 (FullGradientNegation [EGR18])**.** La négation du gradient complet est l'instanciation locale de l'attaquant omniscient de [EGR18] : 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 [EGR18])**.** 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 workers malveillants 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 section 5.2 le montre : ressembler aux honnêtes en norme ne suffit pas, car une coordonnée empoisonnée passe les règles de type `Krum`, et ce décalage d'un seul point sur une seule coordonnée illustre l'effet de la grande dimension.

3.5 Librairies existantes : des taxonomies incompatibles

Cinq systèmes fédérés et byzantins sont retenus comme points de comparaison : ByzFL (GGPRST25), FedLab (ZLHWX23), Blades (LNYJZV24), FL-Byzantine-Library (CRY24 ; OOKG24) et ByzPy (Byz25), sans vocabulaire ni protocole d'évaluation communs. Le critère de rétention est la comparabilité : chacun propose du code ouvert d'agrégation byzantine avec une taxonomie explicite de ses primitives. De nombreuses autres librairies fédérées existent, mais sans volet byzantin, sans code réutilisable ou sans protocole d'évaluation décrit.

Le nombre de primitives proposées varie fortement selon les librairies, dans les versions examinées en août 2026. FedLab ne propose que 2 règles et aucune attaque, contre 36 règles et 22 attaques dans FL-Byzantine-Library. Les pré-agrégateurs, ces filtres préalables avant l'agrégation, n'existent que dans deux systèmes. Quand il existe, le wrapper de modèle aplatit par copie. Pour le partitionnement, les librairies proposent de deux à neuf schémas, à l'exception d'un système qui n'en fournit aucun. Les méthodes de partitionnement

reposent principalement sur le tri par étiquette, le tirage de proportions selon une loi de Dirichlet et le découpage des données. Elles n'ont pas d'interface commune.

Les simulations se répartissent en trois catégories. Les simulations fidèles reprennent les mécanismes d'un protocole publié pour en retrouver le phénomène, avec des adaptations documentées (section 6.1). Les simulations génériques proposent un cadre configurable pour comparer plusieurs méthodes. Les suites de baselines regroupent des méthodes de référence dans un même cadre d'évaluation. Les cinq systèmes examinés proposent des simulations génériques ou des suites de baselines, jusqu'à dix baselines chez FedLab. Seul ByzPy propose aussi le pair-à-pair, dans un cadre générique. Aucun de ces cinq systèmes ne propose de reproduction fidèle au sens retenu dans ce mémoire.

Pour l'orchestration, balayage et déploiement restent confondus. La spécification passe par fichiers, gestionnaires de serveur, ligne de commande ou graphes de tâches. Les métriques partent dans des fichiers sur disque, et la relance après échec reste partielle ou absente. Le parallélisme s'organise par configuration ou par réglage distribué. Pour le déploiement, la plupart restent sur une seule machine. FedLab propose trois modes, ByzPy cinq moteurs et Blades s'appuie sur un exécuteur distribué. La section 6.3 justifie ce choix parmi les nombreuses librairies existantes et détaille la comparaison en trois blocs.

Au fil de ce chapitre, la menace s'est précisée : le pire cas destructeur combine collusion et omniscience, et les garanties visent ce scénario. Les règles se rangent en quatre familles, de la sélection par distances au filtrage coordonnée par coordonnée, avec la moyenne simple comme référence sans garantie. Pour les attaques, tout reste dans ces hypothèses, avec des stratégies génériques puis ciblées sur la vulnérabilité cachée des règles de sélection. Les librairies, elles, perdent le suivi : vocabulaires et protocoles incompatibles, aucune reproduction fidèle, orchestration confondue avec le déploiement. Il manque donc une interface unifiée, des protocoles fidèles et une orchestration reproductible. C'est ce manque que la librairie du chapitre 4 vient combler. La conception de son orchestrateur s'appuie sur l'expérience de recherche de Sébastien Rouault, qui lui a permis d'en préciser les besoins.

Chapitre 4 Conception de Krum

La librairie rassemble les outils nécessaires pour étudier ces méthodes et organiser des expériences. Ce chapitre décrit les fonctions de calcul, les expériences informatiques qui reprennent des protocoles publiés, puis le lancement et le suivi des campagnes d'expériences.

Réunir les primitives sous une interface commune répondait à un besoin évident de réutilisation du code existant. La conception de l'orchestrateur vient de la réflexion de Sébastien Rouault sur ses propres projets de recherche. Dans ses projets, il n'avait jamais disposé d'un tel outil. Les difficultés rencontrées lui avaient permis d'en définir les besoins et d'en penser le fonctionnement en détail. Nous avons pu nous appuyer sur ses idées pour conceptualiser rapidement l'orchestrateur pendant le stage.

4.1 Primitives sans état

Les primitives sont les briques de calcul de Krum. Elles transforment des tenseurs sans garder de mémoire d'un appel à l'autre. Cette section présente d'abord le pattern commun qui les unit et ce qu'il permet d'étendre, puis les trois familles de briques, agrégateurs, attaques et partitionneurs, et les modèles, avec le wrapper zéro-copie et les architectures reprises de la littérature.

4.1.1 Un pattern de conception unique

Les briques de base de Krum suivent le même schéma de conception, appelé pattern. Chaque brique est définie par une classe, qui décrit le composant logiciel. Son calcul passe par une seule méthode de classe, une fonction que l'on appelle directement sur cette classe. Il est donc inutile de créer au préalable un objet à partir de la classe, opération appelée instanciation. La brique ne conserve aucun état propre 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. Il s'agit des gradients des n workers en vues plates pour une règle d'agrégation, et des gradients honnêtes pour une attaque. Les hyperparamètres spécifiques sont nommés explicitement, comme le nombre de malicieux f pour les règles d'agrégation. Dans tout le chapitre, n et f suivent la terminologie minimale (section 3.1).

Ce choix supprime la configuration d'objets, source classique d'erreurs silencieuses : sans état à initialiser, aucun oubli d'initialisation n'est possible. La méthode ne conserve pas

d'état propre entre les appels. Elle peut toutefois écrire son résultat dans une zone de mémoire, appelée tampon ou buffer, fournie par l'utilisateur. Cette organisation facilite les tests et la combinaison des composants. Les agrégateurs et les attaques sont appelés à chaque pas d'entraînement, y compris dans les balayages de milliers d'exécutions. Leur méthode accepte donc un tampon `out` réservé à l'avance, transmis 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, sans autre allocation dans la boucle.

4.1.2 Extensibilité

Ce pattern a une conséquence directe : étendre la librairie demande une seule 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 fonctionne aussitôt avec les couches supérieures, sans les modifier. Une nouvelle règle d'agrégation tourne 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, avec le même cache et les mêmes métriques. Krum sert ainsi de base d'expérimentation.

4.1.3 Règles d'agrégation

Krum embarque onze règles d'agrégation, prises dans 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	$f \leq (n-3)/2$ (BEGS17)
`MultiKrum`	moyenne des m gradients sélectionnés	$f \leq (n-3)/2$, $m \leq n - 2f - 3$ (BEGS17)
Voisinage		
`NNA`	moyenne des k plus proches voisins d'un pivot ($k =$ `num_closest`)	politique de l'appelant ($n - 2f$ pour `MoNNA`) (FGGHPS23)
Filtrage coordonnée par coordonnée		
`Average`	moyenne simple	baseline non robuste
`Median`	médiane coordonnée par coordonnée	$f \leq (n-1)/2$ et bruit contrôlé (YCRB18)
`TrimmedMean`	moyenne tronquée par coordonnée	$f \leq (n-1)/2$ et $\beta \geq f/n$ (YCRB18)
`Bulyan`	`MultiKrum` puis `TrimmedMean` coordonnée par coordonnée	$f \leq (n-3)/4$ (EGR18)
Méthodes géométriques et énumératives		
`GeoMed`	médiane géométrique par itérations de Weiszfeld lissées (Wei37 ; PKH22)	$f \leq (n-1)/2$ (PKH22)
`Medoid`	médoïde (médiane restreinte aux vecteurs soumis)	baseline non résiliente (BEGS17) ; définition (XKG18)
`Aksel`	pivot médian, temps linéaire en $O(nd)$	$f \leq (n-1)/2$ (point de rupture optimal) (BEGMR21)
`Brute`	énumération des sous-ensembles	exacte ($f \leq (n-1)/2$)

TABLE 4.1 – Règles d'agrégation implémentées dans KRUM et leurs garanties, comme définies à la section 3.3. « Baseline » signale une règle incluse comme point de comparaison. m et β sont les paramètres des définitions 3.3.2 et 3.3.5.

`TrimmedMean` correspond à la définition au point $\beta = f/n$. `MultiKrum` prend $m = n - 2f - 3$, la borne maximale admissible, et `GeoMed` lisse les itérations de Weiszfeld avec $\nu = 0{,}1$. Le catalogue de l'annexe A les recense.

4.1.4 Attaques

Évaluer une règle d'agrégation suppose de savoir l'attaquer. KRUM fournit les cinq attaques de la section 3.4, également sans état. Chacune implémente `Attack.generate` à partir des gradients honnêtes et du nombre de malhonnêtes f. La librairie fixe les réglages par défaut : `SignFlip` applique `scale` à 1, la négation exacte de la moyenne honnête ; `ALIE` retient la variante oracle, avec $z = z^{\max}$ et la direction négative, soit z écarts-types sous la moyenne honnête coordonnée par coordonnée ; `Gaussian` part de $\mu = 0$ et $\sigma = 200$; `FullGradientNegation` prend $\kappa = 100$; `SmallPerturbation` opère en norme 2, sur la coordonnée de plus forte variance honnête, et borne sa recherche à $\gamma_{\max} = 10^6$.

`SmallPerturbation` est la seule de ces attaques à exploiter finement la frontière de sélection des règles de type `Krum`. Celles-ci admettent des vecteurs proches du groupe des gradients honnêtes mais mal orientés, un effet typique de la grande dimension. Inclure dans la librairie l'attaque qui met en défaut ses propres règles rend l'évaluation honnête. Le chapitre 5 utilise exactement cette configuration.

4.1.5 Partitionneurs de données

Le passage à des données non IID sert à reproduire les conditions 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 avec un sous-jeu de données par worker, à partir d'un jeu d'entraînement et d'une graine.

`IidPartitioner` mélange le jeu puis le segmente. `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 $\lambda = 0$, chaque worker ne voit qu'une seule classe. `DirichletPartitioner` distribue les classes selon le paramètre α (section 3.1) (HQB19). Un α petit produit un déséquilibre extrême, jusqu'au cas limite d'un worker sans échantillon d'une classe. La librairie accepte ces jeux partiels ou vides. 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 tampon contigu, alloué une seule fois. La lecture et l'écriture se font sans copie.

Sans ce wrapper, chaque accès à un gradient complet demande 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. Le coût de l'interface devient alors négligeable devant le calcul. Parmi les librairies comparées au chapitre 6, KRUM est la seule à offrir cette garantie.

Le paquet `models` comprend le wrapper et six architectures prêtes à l'emploi, dont cinq reprises des publications reproduites : `Krum2017MLPMnist`, `Krum2017MLPSpambase` et `Krum2017CNN` pour les protocoles centralisés.

Pour le protocole décentralisé, `Monna2023CNNMnist` et `Monna2023CNNCifar10` reprennent l'appendice de la publication `MoNNA`. Une variante plus légère, `Monna2023SmallMnist`, complète ces architectures. Elle a été ajoutée pour les expériences rapides et ne provient pas de la publication. Ces six architectures sont détaillées en annexe C.

4.2 Simulations fidèles aux protocoles publiés

La couche *simulations* rejoue les protocoles publiés. KRUM en embarque trois, implémentés d'après les publications : le serveur de paramètres centralisé de [BEGS17] (NeurIPS 2017), le protocole de [EGR18] (ICML 2018), et le protocole décentralisé pair-à-pair `MoNNA` de [FGGHPS23] (ICML 2023). « Fidèle » signifie ici conserver les mécanismes du protocole publié nécessaires à l'étude du phénomène visé. Les adaptations du modèle, des données, des règles et des paramètres sont documentées. Le sens précis et borné est détaillé à la section 6.1. Les expériences du chapitre 5 le confirment.

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 malicieux, le paramètre `byzantine_reach` distingue deux régimes de diffusion. Avec `all`, chaque modèle malveillant atteint chaque worker, le pire cas. Avec `sampled`, chaque worker reçoit entre 0 et f modèles malveillants, selon le gossip (section 3.1). 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, deux exécutions sont bit à bit identiques (section 6.1) : la graine globale est passée au constructeur et exploitée par `setup`, et chaque worker dispose de son propre générateur aléatoire. 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.

4.3 Orchestration d'expériences

Le fonctionnement présenté ici reprend les idées de Sébastien Rouault, issues de son expérience dans l'organisation et l'exécution des expériences de ses projets de recherche.

Pour évaluer une règle d'agrégation, on lance des milliers d'expériences en faisant varier les paramètres d'un protocole de simulation. Une expérience correspond à une combinaison de paramètres et à la collecte de métriques associée. Chaque appel à `Orchestrator.run` déclare une expérience et la place dans une file d'attente. L'utilisateur écrit des boucles Python pour parcourir les combinaisons de paramètres à explorer. L'ensemble des expériences ainsi déclarées constitue un balayage. Leur exécution effective est déclenchée par un appel à `drain` ou à `get`. Les paramètres sont appariés à la signature de la fonction d'expérience, c'est-à-dire aux noms et valeurs par défaut de ses arguments. L'orchestrateur construit une clé à partir des paramètres de l'expérience. Cette clé lui permet d'identifier les résultats déjà enregistrés et de déterminer s'ils peuvent être réutilisés.

Dans une exécution (*run*), l'utilisateur déclare des métriques typées avec `Metric(name, dtype)`. Le `dtype` impose le type des valeurs, vérifié à chaque ajout. Il s'agit du type

attendu, par exemple `float` pour un nombre réel. Il les alimente par `push(step, value)`. Chaque point est automatiquement étiqueté avec les paramètres de l'exécution. La récupération se fait par `get(name)`, qui rend une `MetricTable`. Une `MetricTable` est une table simple sans dépendance externe. Elle porte `step` puis `value`, puis les paramètres du balayage et une clé de job, dite `job_key`. Cette clé nomme le dossier où le résultat est stocké. La conversion par `to_pandas()` rend un tableau pandas, la librairie des tables de données en Python. Filtrage, agrégation (`groupby`) et fusion (`merge`) passent par cette conversion. L'exemple canonique, un balayage croisant n, f, règles d'agrégation et attaques avec la baseline honnête, est détaillé en annexe B.

L'orchestrateur prend en charge ces balayages massifs. `run` place les expériences dans une file d'attente. `drain` ou `get` exécute les expériences en attente. L'exécution est séquentielle et à échec rapide (*fail fast*) : toute expérience qui échoue interrompt le balayage en nommant la clé courte du job et ses raisons, plutôt que de laisser une courbe vide passer inaperçue. Un job est une exécution mise en file. La relance ne rejoue que les expériences nouvelles, ratées ou devenues périmées. Un résultat devient périmé après un changement de paramètres, de code ou d'environnement, par exemple la version de Python ou le fichier de dépendances. La lecture se limite aux jobs du balayage courant. Aucune table de résultats ne mélange ainsi deux versions du code. Avec `isolate=True`, chaque job tourne dans un interpréteur frais, toujours séquentiellement et sans changer le contrat `push`/`get`. Le parallélisme reste un travail futur. 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.

Krum est testée, avec une suite de tests automatiques `pytest` qui couvre les garanties des règles. Elle est documentée [1] et publiée sur PyPI sous licence MIT, avec une compatibilité assurée pour Python 3.12 à 3.14. L'installation s'obtient en une seule commande, `pip install krum`, et les mises à jour ultérieures par `pip install -upgrade krum`, sans téléchargement manuel ni configuration particulière. La chaîne complète, primitives, protocoles fidèles et balayage reproductible, est donc livrée prête à l'emploi.

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

Chapitre 5 Validation expérimentale

La librairie a été utilisée pour reproduire trois expériences décrites dans la littérature. Les paramètres ont été adaptés pour que chacune s'exécute en moins de trois minutes sur CPU. Ce chapitre présente ces choix et les résultats obtenus, puis précise dans quelle mesure ils correspondent aux résultats publiés.

5.1 Reproduction de [BEGS17] (NeurIPS 2017) : MultiKrum contre Average

5.1.1 Paramétrage

Cette expérience s'inspire du protocole Spambase de [BEGS17], en régime centralisé. Elle vise le phénomène rapporté plutôt que la recopie des réglages : la divergence de `Average` sous attaque contre la convergence de `MultiKrum`. 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. La fonction de perte est la cross-entropie usuelle en classification ; ce choix s'applique aux trois expériences du chapitre. 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 $f \in \{0, 6\}$ malhonnêtes. 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. `MultiKrum` tourne avec $m = n - f$, le choix du papier pour sa figure 6 : $m = 14$ sous attaque ($f = 6$) et $m = 20$ sans attaque, où `MultiKrum` coïncide avec `Average`. Ce réglage sort de la plage de garantie, qui exige $m \leq n - 2f - 3 = 5$; le papier le pose lui-même hors borne. Ce réglage conserve davantage de gradients qu'une sélection respectant la borne théorique. Il permet d'étudier l'effet de l'attaque avec une agrégation plus proche de la moyenne simple.

5.1.2 Résultats

La figure 5.1 croise les deux règles avec et sans attaque. Sans malveillant, `Average` et `MultiKrum` coïncident ($m = n$) et convergent ensemble : la perte test décroît de 0,81 à 0,44 et l'exactitude atteint 0,82. Sous attaque ($f = 6$), leurs trajectoires divergent : la perte de `Average` s'envole entre les tours 30 et 45 (poids devenus indéfinis, *NaN*, à partir

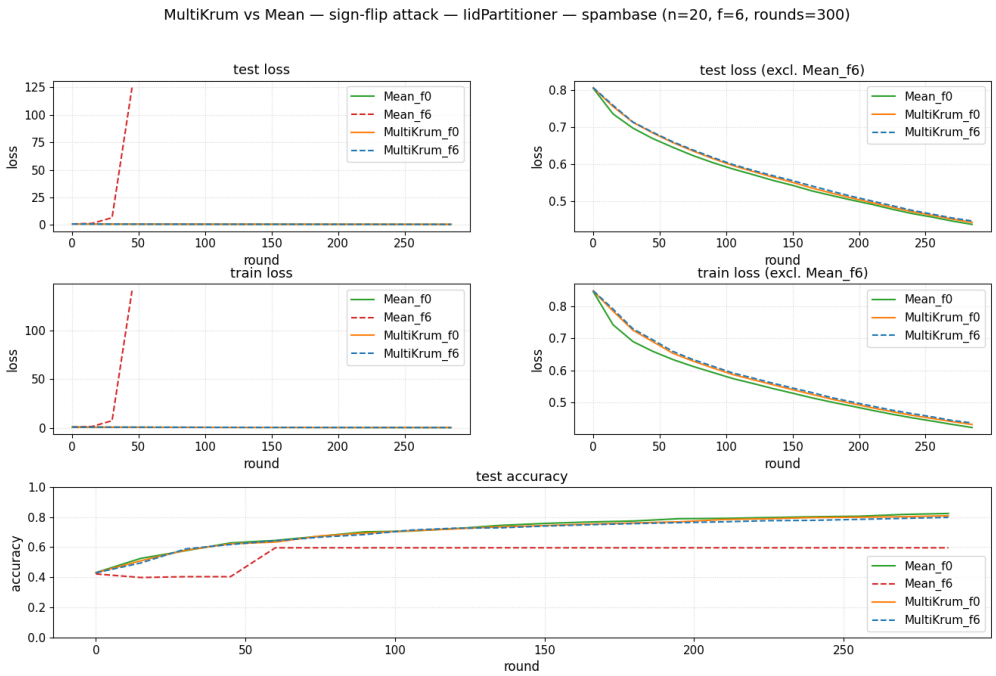


FIGURE 5.1 – Reproduction du protocole Spambase sous `SignFlip` d'amplitude 10, avec $n = 20$, $f = 6$, $m = n - f = 14$, 300 tours et graine 42. Les panneaux montrent les pertes test et d'entraînement, avec puis sans la courbe divergente de `Average` attaqué. L'exactitude test de `MultiKrum` attaqué atteint environ 0,77.

du tour 60), tandis que `MultiKrum` suit sa trajectoire, autour de 0,77 d'exactitude. Les panneaux de droite, qui excluent la courbe divergente, permettent de comparer `MultiKrum` sous attaque à la référence sans attaque. Dans cette expérience, l'exactitude de `MultiKrum` sous attaque est inférieure d'environ cinq points à celle de cette référence.

La courbe d'exactitude de `Average` attaqué suit un régime à part : après un début autour de 0,40, elle saute à un palier de 0,60 vers le tour 60. C'est un artefact de bascule sur la classe majoritaire ($\approx 60\,\%$ des exemples) une fois les poids indéfinis. Le phénomène du papier est reproduit : `MultiKrum` encaisse six malveillants sur vingt, soit 30 %, sans décrocher, là où `Average` s'effondre.

5.2 Reproduction de [EGR18] (ICML 2018) : MultiKrum contre Bulyan

5.2.1 Paramétrage

En régime centralisé également, la deuxième expérience reprend le protocole de [EGR18]. Elle vise la vulnérabilité cachée de `MultiKrum` et le filtrage de `Bulyan`, sans recopier les 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 $f \in \{0, 9\}$ malicieux. Ce compte satisfait $n = 4f+3$, la borne de `Bulyan`. `MultiKrum` tourne en variante $m = 1$, soit `Krum`. L'attaque est `SmallPerturbation` à $\gamma = 4$ fixé. L'entraînement dure 300 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 malhonnêtes. L'attaque vise la norme 2 sur la coordonnée de plus forte variance honnête. Elle est maintenue pendant les 300 tours, sans arrêt en cours d'entraînement. `Average` sans attaque sert de référence saine.

Pour alléger l'expérience, γ est fixé au lieu de rechercher la frontière de sélection à chaque tour. La valeur vient d'une mesure au tour 0 : avec le voisinage de score à $n - f - 2$ pairs (définition 3.3.1), `Krum` ne sélectionne le vecteur malicieux $B(\gamma)$ que si son score reste sous le plus petit score honnête, frontière mesurée à $\gamma_m \approx 6{,}3$ pour ce protocole. Au tour initial, une amplitude supérieure à cette frontière entraîne le rejet du vecteur malveillant. À $\gamma = 280$, valeur arbitraire, le score de `Krum` du vecteur malicieux dépasse $1{,}5 \cdot 10^6$, contre $\approx 1{,}2 \cdot 10^3$ pour le meilleur score honnête, si bien que l'attaque reste sans aucun effet. Dans l'expérience présentée, la valeur constante $\gamma = 4$ reste sous la frontière de sélection au fil de l'entraînement. Le vecteur malveillant continue ainsi d'être sélectionné à chaque tour. L'attaque éloigne le modèle de l'optimum. Les gradients honnêtes restent dispersés, ce qui maintient la frontière au-dessus de 4.

5.2.2 Résultats

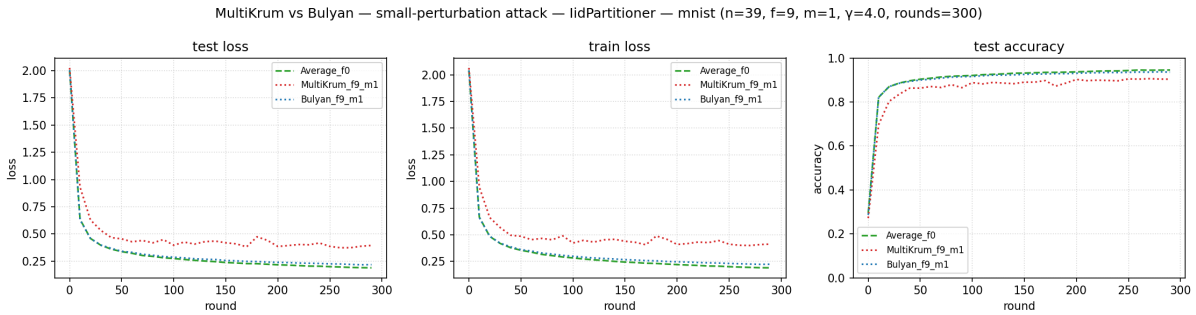


FIGURE 5.2 – Vulnérabilité cachée sous `SmallPerturbation` à $\gamma = 4$ fixé, avec MNIST, $n = 39$, $f = 9$, $m = 1$, 300 tours et graine 42. Les panneaux montrent les pertes test et d'entraînement ainsi que l'exactitude test. `MultiKrum` à $m = 1$ plafonne autour de 0,90, cinq points derrière la référence, tandis que `Bulyan` reste proche de la référence sans attaque.

La figure 5.2 montre trois trajectoires, aux tours 0 à 290. Sans attaque, `Average` converge :

perte test $\approx 0{,}19$, exactitude $\approx 0{,}95$. Sous attaque, `MultiKrum` paraît apprendre puis plafonne : son exactitude se stabilise autour de 0,90, cinq points derrière la référence ($\approx 0{,}90$ contre 0,95), et sa perte test, stable autour de 0,40, reste deux fois plus haute que celle de la référence. `Bulyan`, frappé par la même attaque, colle à la référence sur les trois panneaux (perte $\approx 0{,}21$, exactitude $\approx 0{,}94$).

Le mécanisme du papier est reproduit : à $m = 1$, la sortie de `MultiKrum` est le vecteur malicieux 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, ce qui montre qu'une ressemblance aux honnêtes ne garantit pas la robustesse. La « divergence » désigne ici un apprentissage qui se stabilise à un niveau de performance dégradé. `MultiKrum` semble apprendre et sa perte se stabilise, ce qui peut masquer l'échec. Il faut comparer ses performances à celles de la référence sans attaque pour voir la dégradation, d'où le nom de vulnérabilité cachée.

5.3 Étude de cas à échelle réduite de [FGGHPS23] (ICML 2023) : MoNNA décentralisé (NNA contre moyenne)

5.3.1 Paramétrage

À échelle réduite et en régime décentralisé, les conditions gardent les briques du protocole de [FGGHPS23] : mélange `NNA`, momentum de Polyak, hétérogénéité Dirichlet. Le modèle est réduit, avec moins de nœuds et moins de tours. 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 $f \in \{0, 5\}$ malhonnêtes. L'expérience compare le mélange `NNA` à `Average`. Sous attaque ($f = 5$), chaque worker honnête agrège $n - f = 11$ candidats : son modèle local, cinq modèles honnêtes provenant d'autres workers et les cinq modèles malveillants. Les cinq autres workers honnêtes sont tirés au hasard à chaque tour. `NNA` conserve les $n - 2f = 6$ modèles les plus proches du modèle local, pivot compris, tandis que `Average` moyenne les onze candidats. L'attaque est `SignFlip` d'amplitude 1. Chaque modèle malveillant atteint chaque worker honnête. Ce régime correspond au paramètre `byzantine_reach` réglé sur `all`, sa valeur par défaut. Sans attaque ($f = 0$), chaque worker agrège les seize modèles et les deux règles coïncident. 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.

5.3.2 Résultats

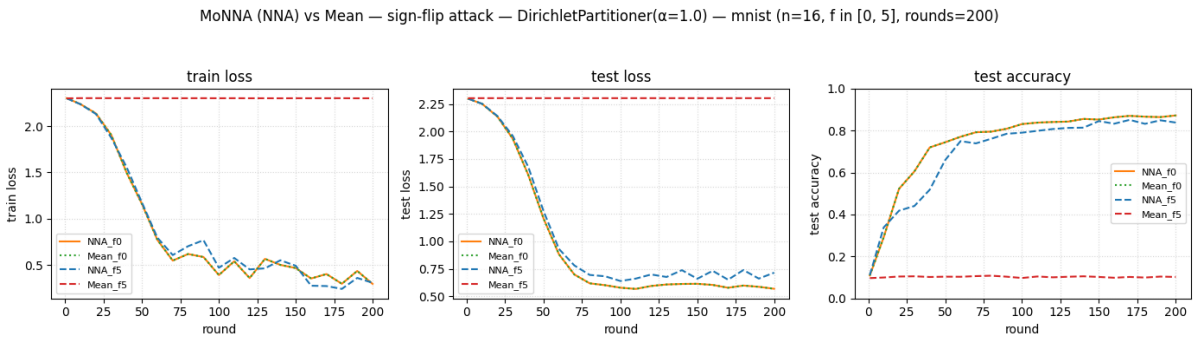


FIGURE 5.3 – Protocole MoNNA décentralisé sous `SignFlip`, avec MNIST, $n = 16$, $f = 5$, 200 tours, graine 0 et Dirichlet $\alpha = 1$. Les panneaux montrent les pertes d'entraînement et test ainsi que l'exactitude test. Sans attaque, `NNA` et `Average` convergent ensemble. Sous attaque, `NNA` rejoint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

La figure 5.3 oppose les deux règles avec et sans attaque. Sans malveillants, `NNA` et `Average` convergent ensemble vers $\approx 0{,}87$ d'exactitude. Sous attaque ($f = 5$, près d'un tiers de malveillants), `Average` ne décolle jamais, car son exactitude est 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 d'exactitude entre `NNA` sous attaque et `NNA` sans attaque atteint environ vingt points entre les tours 45 et 60, puis diminue. `NNA` sous attaque finit autour de $\approx 0{,}84$, à trois points des courbes saines.

Le phénomène MoNNA est reproduit à échelle réduite : tant que les modèles adversariaux restent minoritaires parmi les modèles reçus, le mélange aux plus proches voisins les isole et l'apprentissage suit sa trajectoire saine. La règle `Average`, elle, est ramenée au hasard. On y retrouve sous forme décentralisée la leçon de la section 5.1 : sans sélection des modèles reçus, l'agrégation ne résiste pas.

Les trois protocoles des sections 5.1 à 5.3 retrouvent leur phénomène à échelle réduite, en moins de trois minutes sur CPU. Chaque expérience conserve les mécanismes du protocole nécessaires à l'étude du phénomène visé, avec des adaptations documentées et des graines fixées. Deux exécutions sont identiques à code, graines et matériel fixés.

Chapitre 6 Discussion et limites

Les expériences du chapitre précédent reprennent les protocoles publiés avec des paramètres adaptés et dans un environnement simulé. Ce chapitre examine les écarts avec les publications, ce que les résultats permettent de conclure et la place de la librairie parmi les outils existants.

6.1 Forces et compromis

Un premier atout de la librairie est la conception sans état de la section 4.1. Chaque brique s'appelle comme une simple fonction, sans mémoire entre deux appels. Ce choix écarte l'oubli d'initialisation, erreur silencieuse classique, et rend chaque règle testable isolément, composable dans n'importe quel protocole. De plus, étendre la librairie demande une seule méthode pour les agrégateurs, les attaques ou les partitionneurs. Le wrapper zéro-copie prolonge ce soin : il expose les paramètres en vues qui partagent la mémoire du modèle, sans copie. Sans lui, chaque pas paierait une copie par worker. Cela représente près de 12 000 copies intégrales par expérience pour le seul protocole 2018, avec 39 workers et 300 tours. L'interface est exigeante à implémenter, mais rend le coût d'accès négligeable devant le calcul. Aucun réglage ne persiste entre appels : les balayages passent les hyperparamètres en arguments nommés.

La fidélité des protocoles est un autre atout (section 4.2). Krum reprend les mécanismes de trois protocoles publiés, avec des adaptations documentées (BEGS17 ; EGR18 ; FGGHPS23). Les graines sont fixées et le calcul est déterministe bit à bit à code et matériel fixés. Cette fidélité concerne les mécanismes conservés et le phénomène retrouvé, au sens précisé ci-dessous. Elle n'implique pas l'identité numérique des courbes. La couverture reste étroite : trois protocoles contre dix baselines génériques chez certains concurrents (ZLHWX23). Pour reproduire les phénomènes de ces publications, ces trois protocoles sont plus utiles qu'une suite de dix baselines sans rattachement à un papier. Une telle suite sert davantage à comparer rapidement de nombreuses méthodes.

Dans ce mémoire, une reproduction dite fidèle conserve les mécanismes du protocole publié nécessaires à l'étude du phénomène visé. Les adaptations du modèle, des données, des règles et des paramètres sont documentées. Dans l'étude de cas MoNNA, un perceptron remplace le CNN du papier, mais le mélange `NNA`, le momentum et le partitionnement de Dirichlet sont conservés. D'autres adaptations portent sur l'attaque, le découpage des données, le pas, la taille de lot ou le nombre de tours. Les graines valent 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. Ce déterminisme assure leur répétabilité dans ces conditions. La fidélité concerne les mécanismes conservés, le phénomène observé et les conditions documentées ; elle ne suppose pas l'identité numérique avec les résultats des publications.

Côté orchestration, la section 4.3 apporte le troisième atout : le balayage se déclare en code exécutable, et une campagne entière tient dans un seul script, qui déclare, exécute puis analyse. L'orchestrateur accepte le propre code de l'utilisateur, pour réutiliser une simulation existante ou écrire un scénario inédit. Les métriques restent libres, et les tables qu'elles alimentent s'exploitent directement, par filtrage et fusion, sans suivi manuel des paramètres. La contrepartie est le déploiement mono-hôte, sur une seule machine, un choix cohérent avec la vocation de Krum : une librairie de recherche pour le balayage simulé, non une infrastructure de production distribuée.

6.2 Limites actuelles

La première limite de la librairie est l'absence de pré-agrégateurs, ces filtres préalables appliqués avant l'agrégation : le bucketing, le clipping ainsi que la pondération robuste. Le bucketing regroupe les gradients en paquets préalablement moyennés pour réduire la variance (KHJ20). Le clipping écrête les normes au-delà d'un seuil pour borner l'influence d'un vecteur aberrant (OOKG24). La pondération robuste des clients ajuste le poids de chaque worker selon une mesure de fiabilité (PTH22). Ces trois filtres récents assainissent les gradients avant l'agrégation, en réduisant bruit et variance. Aucun des trois protocoles rejoués n'en emploie, donc la validation du chapitre 5 n'en avait pas besoin : ne pas les implémenter est un choix assumé. La librairie vise l'évaluation des règles d'agrégation face aux attaques, pas le déploiement sur réseau réel. Pour généraliser à des protocoles qui les emploient, ces filtres seront nécessaires. Ils suivront la conception sans état de la section 4.1, pour rester cohérents avec l'architecture, et figurent parmi les travaux futurs du chapitre 7.

Autre limite, l'exécution sur un seul hôte. FedLab propose trois modes de déploiement (ZLHWX23) et ByzPy cinq moteurs d'exécution (Byz25). Krum exécute chaque expérience séquentiellement sur la même machine. Les balayages de milliers d'expériences simulées y tiennent, mais aucun protocole ne tourne en conditions réseau réelles. Les résultats valent donc pour le simulé : le comportement sur réseau réel reste hors de portée. C'est le prix du mono-hôte, acceptable pour des balayages de recherche. Le balayage parallèle viendra ensuite.

La comparaison avec les publications comporte quatre limites supplémentaires : le matériel utilisé, les simplifications expérimentales, le nombre de protocoles couverts et les variantes d'implémentation.

Le matériel utilisé diffère de celui des publications. L'identité bit à bit des poids avec leurs résultats n'est donc pas garantie.

Les simplifications expérimentales constituent la deuxième limite : $\gamma = 4$ fixé au lieu de la recherche de frontière, petit MLP, moins de nœuds et 200 tours pour MoNNA. Elles répondent à l'objectif fixé par l'équipe : chaque expérience doit produire ses résultats en moins de trois minutes sur CPU. Le matériel d'exécution (*device*) est détecté automatiquement, avec repli CPU, pour tenir sur une machine ordinaire.

La troisième limite est le périmètre des simulations, restreint aux trois protocoles rejoués au chapitre 5.

La quatrième concerne les variantes d'implémentation. `Bulyan` sélectionne $\theta = n - 2f - 2$ vecteurs au lieu de $n - 2f$ (EGR18), `Aksel` moyenne les $n - f$ gradients les plus proches du pivot au lieu du seuil médian du papier (BEGMR21), et `MultiKrum` moyenne les m plus petits scores en une seule passe au lieu du retrait itératif (BEGS17). L'écart concernant `Bulyan` est conservé sur l'avis des auteurs, afin de maintenir la continuité avec les comparaisons antérieures. Il est explicitement documenté. L'annexe A recense ces trois variantes.

Une contre-expérience justifie le réglage large de la section 5.1. Cette section fait tourner `MultiKrum` à $m = 14$, hors de la borne de garantie $m \leq 5$. À la borne exacte $m = 5$, l'entraînement diverge plus vite que la référence non robuste : la perte test dépasse 23000 dès le tour 15 et les poids deviennent indéfinis, dits *NaN*, dès le tour 30. À titre de comparaison, `Average` attaqué diverge plus lentement, avec une perte autour de 125 au tour 45 et des poids indéfinis à partir du tour 60. Il mélange le malicieux aux honnêtes à chaque tour, d'où une dérive progressive. L'interprétation reste prudente : à sélection stricte, le vecteur malicieux sélectionné pèse lourd dans une moyenne courte, tandis qu'à $m = 14$ il se dilue parmi quatorze vecteurs. Ce réglage hors borne est celui du papier lui-même, avec $m = n - f$. S'y tenir isole le comportement malicieux, là où la borne exacte $m = 5$ le concentre. C'est ce réglage, assumé, que retient le chapitre 5.

6.3 Krum face aux autres librairies

1. Le papier compagnon est en cours de rédaction. Il décrit Krum selon ses trois couches, primitives, simulations et orchestration, et la compare aux autres librairies. Il vise la rubrique Machine Learning Open Source Software (logiciels libres) du Journal of Machine Learning Research (section 2.2).

	Krum papier compagnon[1]	ByzFL arXiv 2025 (GGPRST25)	FedLab **JMLR 2023** (ZLHWX23)	Blades IoTDI 2024 (LNYJZV24)	FL-Byz-Lib TIFS 2024 (OOKG24)	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	Server-Handler	YAML + Ray	CLI dataclass	DAG sched.
Collecte des métriques	**`Metric` typé**	fichiers	fichiers	fichiers	fichiers	fichiers
Relance après échec ou modification	✓	partiel	—	—	—	—
Parallélisme (balayage)	séquentiel (parallélisme futur)	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.

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

Cinq systèmes ont été retenus comme les plus pertinents : ByzFL (GGPRST25), FedLab (ZLHWX23), Blades (LNYJZV24), FL-Byzantine-Library (CRY24 ; OOKG24) et ByzPy (Byz25). Le critère de rétention est la comparabilité : chacun propose du code ouvert d'agrégation byzantine avec une taxonomie explicite de ses primitives. De nombreuses autres librairies fédérées existent, mais sans volet byzantin, sans code réutilisable ou sans protocole d'évaluation décrit. Les points suivants reprennent en miroir les blocs de la section 3.5.

Parmi les cinq systèmes comparés à Krum dans le tableau 6.1, aucun ne combine des protocoles fidèles et une orchestration reproductible. FL-Byzantine-Library propose ainsi 36 règles et 22 attaques, mais un seul protocole générique. Dans cette comparaison, Krum est également la seule librairie à proposer à la fois une relance après modification et des métriques typées. Le pair-à-pair est couvert par Krum et ByzPy. Seule Krum le propose dans une reproduction fidèle, avec `MoNNA`.

En primitives, Krum est la seule à réunir `Bulyan` et `Brute` (EGR18), `Aksel` (BEGMR21) et les deux attaques de la vulnérabilité cachée, `FullGradientNegation` et `SmallPerturbation` (EGR18) Les comptes bruts restent inférieurs, avec 11 règles et 5 attaques contre 36 et 22 au maximum. Krum est aussi la seule avec un wrapper zéro-copie, qui expose les paramètres en vues partageant la mémoire du modèle. Les autres aplatissent par copie, dite flatten, ou n'offrent aucun wrapper. Le partitionnement compte quatre schémas, dont Dirichlet et le mélange paramétré, contre deux à neuf ailleurs. Tous reposent sur les mêmes mécaniques, tri par étiquette, proportions de Dirichlet et découpage. Au total, le catalogue mise sur les références exigeantes et l'interface unifiée plutôt que sur le volume.

En simulations, seul Krum propose des protocoles fidèles, au nombre de trois. Les autres rejouent des moyennes fédérées génériques ou des suites de baselines sans rattachement à une publication précise. Seuls Krum, avec `MoNNA` fidèle, et ByzPy, avec une topologie pair-à-pair générique, couvrent à la fois le serveur de paramètres et le pair-à-pair. Sans ce cadre fidèle, des attaques ciblées comme la vulnérabilité cachée, calibrées au seuil de la règle, restent inévaluables.

En orchestration, Krum déclare les balayages en Python et collecte des métriques typées. Les autres configurent par fichiers JSON ou YAML, par gestionnaires de serveur ou par arguments en ligne de commande, et journalisent dans des fichiers sur disque. Sur la relance, Krum rejoue les expériences nouvelles, ratées ou devenues périmées après changement de paramètres, de code ou d'environnement, tandis que ByzFL saute seulement les expériences terminées, sans invalidation. Côté parallélisme, Krum reste séquentiel sur un seul hôte, contre un vivier de processus par configuration, un réglage

d'hyperparamètres distribué ou un vivier d'acteurs pour graphes de tâches ailleurs. Seuls FedLab et ByzPy dépassent l'hôte unique, avec trois modes et cinq moteurs respectivement. Les orchestrateurs concurrents servent le déploiement et l'échelle. Celui de la librairie sert la re-exécution à l'identique.

FL-Byzantine-Library reste la meilleure pour un benchmark large et rapide, avec 36 règles et 22 attaques, au prix de protocoles génériques sans attache aux publications. FedLab convient mieux pour explorer le panorama fédéré, avec dix baselines, neuf partitionneurs et trois modes de déploiement dans un cadre publié au JMLR. ByzPy sert la recherche sur l'exécution elle-même, avec sa topologie pair-à-pair et ses cinq moteurs. La librairie du stage l'emporte là où la fidélité aux publications prime sur la largeur.

Ce positionnement résume le choix initial : couvrir toute la chaîne, des briques testées aux protocoles fidèles jusqu'au balayage massif.

Chapitre 7 Conclusion et perspectives

Le stage a abouti à une librairie et à la reproduction de trois expériences décrites dans des publications. Ce chapitre récapitule le travail réalisé, les compétences acquises et les pistes qui le prolongent.

7.1 Synthèse des contributions

Krum s'organise en trois couches, décrites au chapitre 4. Les primitives réunissent onze règles, cinq attaques et quatre partitionneurs, appelés comme de simples fonctions, sans mémoire entre deux appels (section 4.1). Le wrapper `Model` complète cet ensemble : il expose les paramètres en vues qui partagent la mémoire du modèle, sans copie, soit près de 12 000 copies évitées par expérience sur le protocole 2018 (section 4.1.6). Les simulations reprennent les mécanismes de trois protocoles publiés, avec des adaptations documentées (section 4.2). L'orchestration déclare enfin les balayages d'expériences en boucles Python et collecte des relevés typés (section 4.3). Au regard des autres librairies (chapitre 6), Krum couvre ainsi toute la chaîne, des primitives testées au balayage massif. Son catalogue de primitives est moins étendu que celui de certains concurrents. Elle associe des protocoles fidèles, dont un en pair-à-pair, une relance automatique des expériences et des métriques typées. Pour vérifier les phénomènes rapportés dans ces publications, les trois protocoles fidèles sont plus utiles que dix baselines génériques. Les écarts aux papiers, variantes assumées et réglage hors borne, sont documentés.

Une suite de tests automatiques vérifie les garanties des règles, et la documentation est publiée en ligne. La librairie s'appuie sur PyTorch et est distribuée sous licence MIT, disponible sur PyPI en version 0.9.0.

Le chapitre 5 a établi trois phénomènes. Sur Spambase, `MultiKrum` conserve sa convergence sous `SignFlip` alors que `Average` diverge (section 5.1). Sur MNIST, `Bulyan` filtre `SmallPerturbation` là où `MultiKrum` plafonne sous la référence (section 5.2). En régime décentralisé, `NNA` poursuit son apprentissage sous `SignFlip` quand `Average` reste au niveau du hasard (section 5.3). La fidélité de ces résultats concerne les mécanismes conservés, les phénomènes observés et les conditions documentées. Les expériences sont répétables à code, graines et matériel fixés (section 6.1). Les protocoles, leurs conditions d'exécution et les outils de relance donnent ainsi un contenu concret à l'objectif de reproductibilité posé dans l'introduction (chapitre 1).

Un papier compagnon, en cours de rédaction, reprend cette description en trois couches et la comparaison aux autres librairies. J'ai assuré l'essentiel de sa rédaction, puis El-Mahdi El-Mhamdi m'a aidé à corriger et à améliorer le texte. Le lieu de publication visé est détaillé au chapitre 2 (section 2.2).

7.2 Acquis personnels du stage

Ce stage m'a permis d'acquérir une méthode de travail pour développer un logiciel en équipe. L'équipe recherche-produit de quatre (section 2.2) suivait les usages du logiciel libre : le suivi passait par des tickets, une fiche par tâche, et les contributions par des demandes de fusion, relues avant intégration. L'intégration continue contrôlait chaque envoi : style, formatage, vérification des types et tests. Cette revue systématique a fixé des critères explicites d'acceptation et m'a appris la priorisation des fonctionnalités et l'exigence d'architecture et de revue de code.

J'ai aussi appris à relier les choix de conception aux besoins des chercheurs. Le stage a commencé par un audit du code de recherche existant ; la liste des idées de Krum en est née. Des reproductions menées de zéro ont ensuite révélé les tâches les plus coûteuses pour un chercheur. Les trois couches de la librairie répondent à ce constat. Chaque choix y est documenté contre sa publication d'origine, avec ses écarts assumés.

Le stage m'a enfin permis de mettre en pratique les acquis de la formation ISF, Ingénierie Statistique et Financière. J'ai mobilisé les statistiques et l'optimisation pour comprendre les garanties des règles (chapitre 3). La programmation m'a servi à structurer les tests (chapitre 4). La rigueur de rédaction m'a permis de documenter les conditions de fidélité (section 6.1). La rédaction du papier compagnon a prolongé cet apprentissage vers une reproductibilité vérifiable par le code, puisque la revue visée évalue le code autant que les résultats.

Dans l'ensemble, je suis très satisfait de ce stage, aussi bien sur le plan professionnel que personnel. J'ai progressé dans l'implémentation des méthodes d'agrégation et dans la façon de tester et de documenter le code. Travailler au sein d'une équipe de recherche m'a aussi appris à maintenir une bonne organisation dans mon travail. J'ai beaucoup apprécié les échanges avec les personnes qui m'ont accompagné, dont les conseils m'ont beaucoup aidé.

7.3 Futures directions

Ma thèse se déroulera dans le cadre de la chaire ATLAS, qui débutera en novembre 2026. ATLAS signifie Adversarial Techniques for Learning and AI Security, soit techniques adversariales pour l'apprentissage et la sécurité de l'intelligence artificielle. Cette chaire de recherche de l'École polytechnique, anciennement dénommée SRAI (Secure and Reliable AI, intelligence artificielle sûre et fiable), est soutenue par le mécénat de TotalEnergies, c'est-à-dire par un financement d'entreprise. Elle porte sur la sécurité et la fiabilité des modèles d'apprentissage automatique, et son premier axe cite l'agrégation robuste de gradients, sujet du stage, parmi les méthodes défensives étudiées. Le critère d'impact retenu est la publication des travaux dans les trois conférences internationales d'apprentissage automatique (NeurIPS, ICLR et ICML) ou dans des revues de même niveau, comme le Journal of Machine Learning Research, revue visée par le papier compagnon (section 2.2).

Les travaux futurs suivent quatre axes : compléter les primitives, approfondir les garanties théoriques dans le cadre de la thèse, étudier les attaques portant sur les données et étendre les capacités d'exécution.

Court terme. La priorité porte sur les pré-agrégateurs déjà présentés au chapitre 6 (section 6.2) : le bucketing (KHJ20), le clipping (OOKG24) et la pondération robuste des clients (PTH22). Ces trois filtres garderont la conception sans état de la section 4.1 et rejoindront les primitives testées.

Théorie. Le projet de thèse prolonge la théorie. L'article [BBFE26] introduit $\kappa^\star$, le coefficient de robustesse optimal d'une règle : l'écart entre la sortie de l'agrégateur et la moyenne réelle des gradients honnêtes, normalisé par la variance de ces derniers, dans le pire cas. Un $\kappa^\star$ fini donne un taux de convergence explicite ; un $\kappa^\star$ infini supprime toute garantie de vitesse. Il faut maintenant le calculer pour chaque règle de la librairie : un nombre par agrégateur qui quantifie sa vitesse de convergence sous fautes byzantines.

`signSGD` servira de premier cas d'étude et rejoindra la librairie comme douzième règle : chaque worker n'envoie que le signe de son gradient et le serveur tranche par vote majoritaire (BZAA19). L'article [MMSE25] en prouve la tolérance face à des attaquants omniscients et collusifs, avec une fraction tolérée qui croît avec la taille de lot. Une campagne expérimentale fera varier la fraction d'attaquants et la taille de lot sur MNIST, pour observer la réaction de $\kappa^\star$ et de la convergence face à la borne théorique.

Données. La librairie traite actuellement les attaques par envoi de gradients forgés (chapitre 1). Les perspectives portent sur l'empoisonnement des données et les portes dérobées, où des échantillons corrompus ou un déclencheur biaisent l'entraînement sans toucher au protocole (BNL12 ; GDG17 ; BVHES20). Les taggants utilisent des exemples spécialement conçus pour laisser une trace détectable et bénigne dans le comportement du modèle. Ces exemples sont hors distribution, c'est-à-dire qu'ils diffèrent des données habituellement rencontrées. Cette trace permet d'auditer le modèle en observant ses réponses, sans accéder à ses paramètres internes. Ce mode d'observation est appelé boîte noire. La détection de la trace fait l'objet d'une certification statistique (BUE25 ; BEU25). Le plan prévoit une primitive de partitionnement empoisonnée et des métriques sensibles au déclencheur, pour comparer, avec la machinerie de balayage et sur une même échelle, la robustesse au niveau gradient et au niveau jeu de données.

Échelle. D'autres protocoles rejoindront les trois rejoués et l'exécution dépassera la machine unique, vers le parallélisme des balayages puis des conditions réseau réelles comme FedLab et ByzPy (ZLHWX23 ; Byz25). L'ensemble forme un premier projet de thèse dans la continuité du stage.

7.4 Conclusion générale

Pendant ce stage, nous avons réuni dans Krum des composants testés, des protocoles de simulation et des outils pour lancer des balayages d'expériences. Les études de cas retrouvent les phénomènes décrits dans les publications, dans les configurations documentées. Ces outils peuvent être réutilisés pour reproduire les expériences présentées et mener de nouvelles comparaisons.

J'ai appris à passer d'une formulation mathématique à du code, puis à en évaluer le comportement dans des expériences. Le travail en équipe et la rédaction de la documentation m'ont aussi appris à mieux expliquer et justifier les choix réalisés. Pendant la thèse, j'approfondirai l'étude des garanties de robustesse des règles et de leur effet sur la convergence.

Bibliographie

[BVHES20] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin et Vitaly Shmatikov. « How To Backdoor Federated Learning ». In : *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*. T. 108. PMLR, 2020, p. 2938-2948.

[BBFE26] Gilles Bareilles, Wassim Bouaziz, Julien Fageot et El-Mahdi El-Mhamdi. « Byzantine Machine Learning : MultiKrum and an optimal notion of robustness ». In : *arXiv preprint arXiv :2602.03899* (2026).

[BBG19] 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.

[BZAA19] Jeremy Bernstein, Jiawei Zhao, Kamyar Azizzadenesheli et Anima Anandkumar. « signSGD with Majority Vote is Communication Efficient And Fault Tolerant ». In : *International Conference on Learning Representations (ICLR)*. 2019. URL : https://arxiv.org/abs/1810.05291.

[BNL12] Battista Biggio, Blaine Nelson et Pavel Laskov. « Poisoning Attacks against Support Vector Machines ». In : *Proceedings of the 29th International Conference on Machine Learning (ICML)*. Omnipress, 2012, p. 1467-1474.

[BEGS17] 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.

[BEU25] Wassim Bouaziz, El-Mahdi El-Mhamdi et Nicolas Usunier. « Targeted Data Poisoning for Black-Box Audio Datasets Ownership Verification ». In : *2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2025, p. 1-5. DOI : 10.1109/ICASSP49660.2025.10888515. URL : https://arxiv.org/abs/2503.10269.

[BUE25] Wassim Bouaziz, Nicolas Usunier et El-Mahdi El-Mhamdi. « Data Taggants : Dataset Ownership Verification Via Harmless Targeted Data Poisoning ». In : *Proceedings of the 13th International Conference on*

Learning Representations (ICLR). 2025. URL : https://openreview.net/forum?id=6ldD8Y4gBQ.

[BEGMR21] 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.

[Byz25] ByzPy. *ByzPy : Byzantine-Robust Learning Runtime*. https://github.com/Byzpy/byzpy. 2025.

[CSX18] 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.

[CRY24] CRYPTO-KU. *FL-Byzantine-Library*. https://github.com/CRYPTO-KU/FL-Byzantine-Library. Code for [OOKG24]. 2024.

[FGGHPS23] 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.

[GB10] 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.

[GGPRST25] 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.

[GDG17] Tianyu Gu, Brendan Dolan-Gavitt et Siddharth Garg. *BadNets : Identifying Vulnerabilities in the Machine Learning Model Supply Chain*. 2017.

[HQB19] 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).

[KHJ20] Sai Praneeth Karimireddy, Lie He et Martin Jaggi. « Byzantine-robust learning on heterogeneous datasets via bucketing ». In : *arXiv preprint arXiv :2006.09365* (2020).

[LSP82] Leslie Lamport, Robert Shostak et Marshall Pease. « The Byzantine Generals Problem ». In : *ACM Transactions on Programming Languages and Systems* 4.3 (1982), p. 382-401.

[LNYJZV24] 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.

[MMSE25] Emanuele Mengoli, Luzius Moll, Virgilio Strozzi et El-Mahdi El-Mhamdi. « On the Byzantine Fault Tolerance of signSGD with Majority Vote ». In : *arXiv preprint arXiv :2502.19170* (2025).

[EGR18] 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.

[OOKG24] 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.

[PKH22] 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.

[PTH22] Amit Portnoy, Yoav Tirosh et Danny Hendler. « Towards federated learning with byzantine-robust client weighting ». In : *Applied Sciences* 12.17 (2022), p. 8847.

[RM51] 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.

[WZL25] 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.

[Wei37] 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.

[XKG18] Cong XIE, Oluwasanmi KOYEJO et Indranil GUPTA. « Generalized Byzantine-tolerant SGD ». In : *arXiv preprint arXiv :1802.10116* (2018).

[XKG20] 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.

[YCRB18] 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.

[ZLHWX23] 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, hyperparamètres exposés dans la signature (`aggregate`, `generate` et `partition`), garanties et publication d'origine. Il couvre les règles d'agrégation (section A.1), les attaques (section A.2) et les partitionneurs (section A.3). 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	[YCRB18]
`TrimmedMean`	retire les f plus petites et f plus grandes valeurs par coordonnée, moyenne le reste	f	$f \leq (n-1)/2$, $\beta \geq f/n$	[YCRB18]
`Krum`	gradient minimisant la somme des distances au carré aux $n-f-2$ plus proches pairs (cas $m=1$)	n, f	$f \leq (n-3)/2$	[BEGS17]
`MultiKrum`	moyenne des m gradients aux plus petits scores de `Krum`	n, f, m ($=n-2f-3$ par défaut)	$f \leq (n-3)/2$; $m \leq n-2f-3$	[BEGS17]
`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)	$f \leq (n-3)/4$	[EGR18]
`Brute`	énumère les $\binom{n}{n-f}$ sous-ensembles, moyenne du plus compact	n, f	exacte ($f \leq (n-1)/2$)	[EGR18]
`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)	$f \leq (n-1)/2$	[Wei37 ; PKH22]
`Medoid`	vecteur soumis minimisant la somme des distances aux autres	n, f (ignoré)	aucune (baseline non résiliente)	[XKG18 ; BEGS17]
`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)	[BEGMR21]
`NNA`	moyenne des k plus proches voisins d'un pivot ($k =$ `num_closest`)	`num_closest`, `pivot`	politique de l'appelant ($n-2f$ pour `MoNNA`)	[FGGHPS23]

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 malicieux, 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 utilisant le paramètre 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 discutés en section 6.2.

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)	[XKG20]
`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)	[BBG19]
`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)	[EGR18]
`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})	[EGR18]

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 malicieux, 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()`. La récupération produit deux tables exploitables, fusionnées sur les paramètres du balayage, le pas et la clé de job.

```
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)


with Orchestrator("krum_byzantine_study") as orch:
    for n, f in [(13, 2), (25, 5)]:
        for aggregator in [Average, Krum, Bulyan]:
            # attack=None : baseline honnête, sans gradients malveillants.
            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 (noms courts en table).
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, le pas et la clé de job.
merged = (loss_df.rename(columns={"value": "loss"}).merge(
    acc_df.rename(columns={"value": "accuracy"}),
    on=["step", "n", "f", "aggregator", "attack", "seed", "job_key"]))
```

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

Chaque `MetricTable` porte `step` puis `value`, puis les paramètres du balayage et `job_key`. La conversion par `to_pandas()` permet le filtrage, l'agrégation (`groupby`) et la fusion (`merge`) ci-dessus, sans suivi manuel des paramètres. Les classes et fonctions passées en paramètres y figurent par leur nom court, par exemple `"Krum"`. Relancer le script après une panne ou une modification ne rejoue que les expériences nouvelles, ratées ou devenues périmées.

Chapitre C Architectures de modèles prédéfinis

Les six architectures embarquées, dont cinq reprises des publications reproduites, peuvent être utilisées dans les simulations fournies ou comme modèles de référence dans les expériences de l'utilisateur. Leur création ne nécessite aucun argument. Le composant `Model` permet ensuite d'accéder à leurs paramètres sous forme de vecteurs, sans copie (section 4.1).

Modèle	Architecture	Origine
`Krum2017MLPMnist`	perceptron multicouche $784 \to 100 \to 10$ (ReLU), $\approx 8 \times 10^4$ paramètres	[BEGS17] (MNIST)
`Krum2017MLP Spambase`	perceptron multicouche $57 \to 20 \to 20 \to 2$ (ReLU)	[BEGS17] (Spambase)
`Krum2017CNN`	2 convolutions (16 et 64 filtres) + 3 couches denses (384, 192, 10), ReLU et max-pooling	[EGR18] (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	[FGGHPS23] (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)	[FGGHPS23] (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 avec les publications sont assumés. 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.