Architecture d'un réseau décentralisé d'apprentissage machine

par

Michael DUCHESNE

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE LOGICIEL
M. Sc. A.

MONTRÉAL, LE 16 AOÛT 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Kaiwen Zhang, directeur de mémoire
Département de génie logiciel à l'École de technologie supérieure

M. Aris Leivadeas, président du jury
Département de génie logiciel à l'École de technologie supérieure

Mme. Chamseddine Talhi, membre du jury
Département de génie logiciel à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 11 AOÛT 2023

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Un grand merci à Kaiwen Zhang, mon directeur de recherche pour son expertise, son savoir et sa disponibilité. Vos conseils et votre perspective ont été très précieux tout au long de mon parcours.

J'aimerais aussi remercier tout mon entourage pour vos encouragements et votre support inconditionnel.

Finalement, merci à mes amis du lab Fusée, vous êtes d'une détermination et d'une résilience inspirante.

Architecture d'un réseau décentralisé d'apprentissage machine

Michael DUCHESNE

RÉSUMÉ

Plusieurs avancées majeures dans le monde de l'intelligence artificielle, combiné à l'augmentation massive des performances de nos ordinateurs, ont récemment propulsé l'adoption de l'intelligence artificielle dans la vie de tous les jours. En effet, il s'agit d'une technologie très versatile avec d'innombrables cas d'utilisation. Cependant, le processus d'apprentissage requiert beaucoup de données qui sont souvent collectées et centralisées dans des entrepôts de données hors du contrôle de l'utilisateur. Avec raison, cela cause un certain inconfort chez plusieurs utilisateurs, qui craignent que leurs données soient utilisées à leur détriment. D'un autre côté, certains cas d'utilisation ne peuvent être exploré puisque le partage des données des utilisateurs est régi par certaines lois. Par exemple, les données provenant du milieu médical ou financier ne peuvent être partager librement. Afin de remédier à ces problèmes, l'apprentissage fédéré a été proposé afin de laisser le contrôle sur les données à l'utilisateur. Avec cette approche, les utilisateurs entraînent un modèle local sur leur appareil avec leurs données et partagent le modèle à un serveur central qui fera l'agrégation du modèle de tous les utilisateurs. Il s'agit d'une approche très intéressante et prometteuses, mais qui augmente la complexité de l'entraînement.

Une approche inspirée de l'apprentissage fédéré sera proposée. Cette nouvelle approche, décentralisé et inclusive, cherche à créer un système qui permet à plusieurs entités indépendantes et autonomes de collaborer pour entraîner un modèle. En effet, elle laisse entièrement le contrôle de l'entraînement et de l'aggrégation de modèle à chaque participant. Ce document détaille l'architecture, l'implémentation ainsi que les résultats obtenus par cette solution.

Mots-clés: apprentissage fédéré, apprentissage décentralisé, divergence de poids

Architecture of a Decentralized Network for Machine Learning

Michael DUCHESNE

ABSTRACT

Major advancements in artificial intelligence (AI) combined with the constantly increasing computing power pushed adoption of AI in our everyday life. However, the learning process requires to collect and centralize user's data in datacenters outside of the user's control. This practice raises some concerns from users who fear the misuse of their data. On the other side, some use cases cannot be explored as it requires data that is forbidden to share. For example, financial or medical data. To remedy to these problems, a federated approach was suggested. In this approach, users train a model locally on their device and only have to share the global model with a centralized server. Its role is to aggregate models from all users. This approach inspired a lot new variants.

This paper proposes a new decentralised and inclusive version. It recognizes the sovereignty of the different actors and gives total control on the training and aggregation of models. This document document details the architecture, the implementation and the results of this approach.

Keywords: Federated Learning, Decentralised Learning, weight divergence

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

IA	Intelligence Artificielle
FL	Federated Learning
MCFL	MultiConfederatedLearning

INTRODUCTION

Alors que l'intelligence artificielle devient un incontournable dans la vie de tous les jours et contribue à l'amélioration de notre qualité de vie, elle donne une grande valeur aux données à partir desquelles elle est entraînée. Cela a pour effet de motiver les entreprises à conserver le plus de données possible, même si elles n'ont pas d'utilité à l'instant, afin de monétiser ces données par simplement la revente ou encore par l'entraînement d'intelligence artificielle. Cette pratique a cependant des désavantages pour les utilisateurs qui perdent le contrôle sur leurs données personnelles puisqu'elle se retrouvent centralisées dans des centres de données.

De plus, l'exploitation des données peut être fait au détriment de l'utilisateur qui doit confier des données, parfois confidentielles, à des entreprises qui cherchent principalement à en tirer profit. Il y a donc une certaine déconnexion entre les intérêts des utilisateurs et des entraîneurs.

De plus, certaines données, qui sont régies par des lois, ne peuvent pas être partagées, ce qui empêche l'utilisation de ces données lors d'entraînement de modèles qui pourraient bénéficier de ce type de données. Ils sont alors limités à des jeux de données publics qui sont souvent petits ou incomplets. Un exemple commun est le partage de dossiers médicaux. Ces derniers n'ont pas le droit de quitter leur lieu d'origine.

Cependant, il est impossible de nier l'utilité et les résultats impressionnants qui peuvent être obtenus grâce à une intelligence artificielle bien entraînée. Il est donc primordial de ne pas freiner les développements en intelligence artificielle, mais plutôt de développer les techniques nécessaires pour permettre l'entraînement d'intelligence artificielle. En effet, sans la collecte de données massives, l'IA peut être bénéfique pour tous les acteurs.

Afin de remédier à ce problème, des chercheurs chez Google ont proposé une méthode d'entraînement sans avoir besoin de centraliser les données (Yang *et al.*, 2018). Cette approche est appelée Federated Learning et consiste à entraîner un modèle par utilisateur sur leur propre

appareil puis de faire l'agrégation des modèles sur un serveur centralisé. Non seulement cette approche est plus respectueuse de la vie privée, elle permet aussi de faire de l'apprentissage sur des données qui ne peuvent pas être partagées comme des dossiers médicaux ou financiers, car les données n'ont pas besoin de quitter l'appareil.

Cependant, cette approche a ses propres désavantages. En effet, l'agrégateur est encore un système centralisé qui exerce beaucoup de contrôle sur les différents clients qui font l'entraînement. En effet, il s'agit d'un système opéré par une entité maître qui exploite les données sur les appareils de ses utilisateurs. Il s'agit sans contredit d'une amélioration, comparativement à l'apprentissage centralisé, et a inspiré beaucoup de nouvelles stratégies d'agrégation qui vont essayer de continuer d'améliorer ce système. Notamment des alternatives décentralisées comme le Swarm Learning (Warnat-Herresthal *et al.*, 2021) ou BFLC (Li *et al.*, 2021). Cependant, ses approches décentralisées ne prennent pas en compte la diversité des objectifs, des données et des clients.

En effet, puisque l'entraîneur ne peut pas faire de nettoyage dans les données pour s'assurer que les données sont distribuées uniformément, il n'y a aucune garantie sur la distribution des données sur chaque appareil. Ceci mène alors à avoir des modèles qui peuvent grandement varier d'un appareil à l'autre puisqu'ils s'optimisent pour différentes distributions de données. Ainsi, lorsque l'agrégation de ces modèles est faite, le modèle résultant est moins performant, car ils sont trop différents (Zhao *et al.*, 2018).

De plus, ce type de système tente de plaire à la majorité et de développer un seul modèle qui plait à tous, cependant il est possible que certains entraîneurs n'aient pas la même vision du problème. Ainsi, il est difficile d'obtenir une solution optimale pour tous les entraîneurs. De plus, ce manque d'alignement entre les entraîneurs peut causer l'exclusion de certains entraîneurs lors de l'ajout de techniques comme KRUM (Guo *et al.*, 2021).

Ces problèmes rendent l'implémentation d'un système d'apprentissage fédéré ou décentralisé peu attrayant. Ainsi, ce mémoire propose un système décentralisé d'entraîneurs souverains pour l'apprentissage machine inclusif et respectueux de la vie privée. En effet, en donnant plus de liberté sur le processus aux entraîneurs, il devient plus attractif pour eux d'entraîner leur modèle à l'aide de l'apprentissage fédéré, car ils restent plus en contrôle de l'entraînement qu'il puisse bien répondre à leurs attentes. De plus, en réduisant les impacts négatifs d'une distribution non uniforme, cela peut permettre aux entraîneurs d'obtenir des performances similaires ou meilleurs à un entraînement seul. Ainsi, ce mémoire espère aider à l'adoption de l'apprentissage fédéré en proposant des solutions aux aspects négatifs et en démontrant qu'ils peuvent être implémentés avec succès dans divers scénarios.

Ainsi, les contributions principales de ce mémoire sont les suivantes :

1. Élaboration d'une stratégie générique pour créer des réseaux d'apprentissage machine décentralisé et inclusif. Cette stratégie se base sur l'entraînement de plusieurs modèles spécialisés simultanément ainsi que le transfert d'apprentissage pour obtenir des modèles généralisés.
2. Implémentation d'un système d'apprentissage machine décentralisé basé sur l'architecture proposée. Cette implémentation cherche à augmenter la rapidité de convergence d'un réseau avec des entraîneurs n'ayant pas des données uniformes.
3. Analyse d'impacts des différents paramètres de l'apprentissage fédéré pour justifier les décisions d'architectures et d'implémentations.
4. Évaluation de l'implémentation dans plusieurs scénarios d'apprentissage pour démontrer les capacités à converger rapidement.

Ce mémoire est divisé en six chapitres. Le Chapitre 1 présente la recherche ainsi que les connaissances de base requises pour comprendre la solution. Le chapitre 2 élabore sur certains défis rencontrés par l'apprentissage décentralisé et fait l'analyse des paramètres ayant un impact

sur l'apprentissage. Puis, le chapitre 3 propose une architecture générique donnant les libertés nécessaires aux entraîneurs pour surmonter les défis rencontrés. Le chapitre 4 élabore sur l'implémentation de la solution et les détails plus techniques de la solution. L'implémentation est évaluée dans le chapitre 5 dans plusieurs scénarios différents afin de démontrer la flexibilité de l'architecture. Finalement, le chapitre 6 propose des améliorations et des questions de recherche futures.

CHAPITRE 1

PRÉSENTATION DE LA RECHERCHE

Ce chapitre a pour objectif de situer le lecteur dans le contexte et de mettre ses connaissances à niveau avec la recherche. Dans un premier temps, une revue de la littérature sera faite puis la présentation des concepts de bases requis pour la compréhension de la recherche.

1.1 Revue de la littérature

1.1.1 Données réparties de façon non-identique et non-indépendante

Dans (Zhao *et al.*, 2018), les auteurs réussissent à limiter l'impact d'une distribution non uniforme des données en partageant un jeu de données avec tous les entraîneurs. Le jeu de données peut être constitué d'un échantillon provenant des entraîneurs ou un jeu de données publiques. Dans le premier cas, cela brise le concept de ne pas partager les données et dans le deuxième cas, limite beaucoup les cas d'utilisation.

L'article de (Shoham *et al.*, 2019) présente l'apprentissage fédéré à partir d'un jeu de données distribué non uniformément comme étant un problème d'apprentissage multitâche et applique la technique EWC (Kirkpatrick *et al.*, 2017) qui modifie la fonction d'apprentissage pour mitiger les impacts négatifs.

Bien que ces stratégies aient un certain succès, elles ont besoin d'une connaissance partielle des données, ce qui ne satisfait pas nos objectifs. En effet, même la technique EWC, qui ne requiert pas d'échantillon de données, a tout de même besoin d'une certaine connaissance du domaine des données pour s'assurer que le modèle soit surparamétrisé pour permettre l'apprentissage des multiples tâches à l'intérieur du même modèle sans avoir d'impact négatif sur les autres tâches.

Cependant, la stratégie Diviser pour mieux régner implémenté dans (Chandran, Bhat, Chakravarthi & Chandar, 2021) dont l'objectif est de limiter la divergence des poids entre les mises à jour démontre l'intérêt et les gains de performance lorsque l'algorithme est pensé en fonction de

limiter la divergence. La solution proposée dans ce mémoire s'inspire de cette stratégie et tente de limiter la divergence pour accélérer la convergence.

1.1.2 Apprentissage fédéré

La méthode la plus simple pour l'apprentissage fédéré est le FedAvg, qui fait simplement une moyenne des paramètres du réseau de neurones. Ce dernier requiert généralement que les participants aient terminé leur entraînement avant d'en faire l'agrégation. FedProx (Li *et al.*, 2020) est une itération de FedAvg qui tolère cependant les mises à jour qui ne sont pas complètes et limite le changement des poids par ronde. Scaffold (Karimireddy *et al.*, 2021) modifie la fonction de perte en ajoutant un biais pour tenter d'aligner différents entraîneurs qui auraient des données non uniformes. Ces techniques sont intéressantes et pourraient être utilisées de façon complémentaire à l'approche proposée.

Le groupement des clients est une technique qui a été essayée à plusieurs reprises notamment avec HypClusters (Mansour, Mohri, Ro & Suresh, 2020) qui démontre des résultats bénéfiques avec des modèles qui sont plus personnalisés. Dans (Ghosh, Chung, Yin & Ramchandran, 2021), l'orchestrateur regroupe les entraîneurs à l'aide de k-means sur les paramètres des modèles. Quant à Hierarchical Clustering (Briggs, Fan & Andras, 2020), l'orchestrateur regroupe les entraîneurs à l'aide de la distance euclidienne ou de la distance cosine entre les modèles. Finalement, des approches utilisant des graphes acycliques ont été proposées dans (Beilharz *et al.*, 2021) et FedMC (Kopparapu & Lin, 2020). Dans la première approche, les mises à jour sont enregistrées dans un DAG dans lequel deux branches peuvent être agrégées. Dans le la seconde approche, les branches du graphe peuvent être fusionnées en utilisant EWC.

Dans la même optique, l'approche proposée cherche à former des groupes pour obtenir des modèles spécialisés et simultanément réduire la divergence des poids. De plus, les entraîneurs ne sont pas limités à un seul groupe, ils peuvent librement contribuer à autant de groupes qu'ils le souhaitent. De plus, ce mémoire donne aussi une stratégie pour obtenir un modèle plus généralisé à partir de plusieurs modèles spécialisés.

1.1.3 Apprentissage décentralisé

L'apprentissage décentralisé est une adaptation de l'apprentissage fédéré sans serveur centralisé pour orchestrer l'entraînement. Puisqu'il n'y a pas de serveur centralisé, les nœuds sont connectés en pair-à-pair (P2P) comme dans BrainTorrent (Roy, Siddiqui, Pölsterl, Navab & Wachinger, 2019). La chaîne de bloc est aussi utilisée comme méthode de communication dans (Li *et al.*, 2021) dans lequel un comité évalue les nouvelles itérations du modèle avant de les ajouter à la chaîne. Des contrats intelligents peuvent aussi être utilisés comme c'est le cas du Swarm Learning (Warnat-Herresthal *et al.*, 2021).

L'approche proposée connecte les nœuds en pair-à-pair, mais varie des approches étudiées en permettant aux nœuds de s'autoassigner à des sous-groupes pour limiter la divergence des poids. De plus, les nœuds peuvent entraîner plusieurs modèles simultanément.

1.2 Notions de base

1.2.1 Apprentissage fédéré

L'apprentissage fédéré est une technique émergente d'apprentissage machine qui fait l'agrégation de modèles plutôt que des données. En effet, les modèles sont entraînés localement sur les appareils où les données se situent. Ainsi, le serveur central n'a pas d'accès direct aux données. Il agit cependant comme d'un coordonnateur et va contrôler chaque ronde de l'entraînement. Il commence par faire une sélection aléatoire des appareils qui participeront à la ronde. Puis, les appareils sélectionnés vont commencer leur entraînement local sur le modèle global. Les appareils doivent avoir complété et envoyer leur mise à jour avant la fin du temps alloué pour la ronde. Lorsque la ronde est terminée, l'agrégateur applique un algorithme d'agrégation pour obtenir le nouveau modèle global et recommence une nouvelle ronde.

L'algorithme d'agrégation le plus simple est FedAvg A I-1. Dans cette équation, on retrouve K entraîneurs participant à la ronde n entraînant le modèle W_n. L'agrégateur détermine un poids $w?i$ pour chacun des entraîneurs selon le nombre d'échantillon que contient leur jeu de données.

Ainsi, l'équation fait la moyenne des poids tout en pondérant l'importance de la mise à jour basé sur le nombre d'échantillons utilisé pour l'entraînement.

$$W_{n+1}^{g} = \sum_{i=1}^{K} \frac{W_n^i}{w^i} \tag{1.1}$$

L'apprentissage fédéré est plutôt demandant au niveau des coûts de communications dépendamment de la taille du modèle entraîné, mais aussi en fonction du nombre d'appareils participants à l'entraînement dans la ronde. En effet, dans un réseau composé de K clients avec un taux de sélection r, l'agrégateur recevra K*r modèles. En effet, le nombre de modèle reçu augmente linéairement avec le nombre de client. Pour ce qui est du client, il reçoit au maximum un modèle par ronde et envoie un modèle seulement 1/r rondes.

1.2.2 Apprentissage décentralisé

L'apprentissage décentralisé retire le serveur centralisé d'agrégation et connecte les nœuds à l'aide d'un réseau pair-à-pair ou d'une chaine de bloc. Ainsi, l'apprentissage décentralisé hérite généralement des propriétés de l'apprentissage fédéré. De plus, il est possible d'y appliquer des stratégies d'agrégation similaires à l'apprentissage fédéré.

Cependant, le retrait du serveur centralisé implique généralement plus de communication, puisque tous les nœuds doivent connaître tous les modèles, afin d'en faire l'agrégation et vérifier que l'agrégation soit correcte. Par exemple, dans un réseau d'apprentissage décentralisé dans lequel tous les K nœuds font l'entraînement et l'agrégation, alors K2 modèles seront envoyés sur le réseau.

L'apprentissage fédéré et décentralisé peut être exécuté dans deux variantes : entre appareils ou entre silo. Dans la première variante, il s'agit d'un entraînement entre appareils dans laquelle on attend plusieurs milliers d'appareils. Dans la deuxième variante, c'est plutôt un entraînement entre organisations, ce qui réduit considérablement le nombre de participants

(Huang, Huang & Liu, 2022). De plus, on considère que les participants ont des connexions stables et serveurs d'entraînement puissants.

1.2.3 Données non indépendantes distribuées non uniformément

Les entraîneurs ne peuvent pas considérer que les données sont distribuées de façon uniforme autant dans l'apprentissage fédéré que décentralisé. En effet, puisque les entraîneurs sont distribués, il est fort probable que leurs données ne soient pas générées de la même façon, ce qui peut mener à différentes distributions. De plus, il est impossible de connaître la distribution des données puisqu'aucun partage de données ne peut être fait. Il a été démontré par (Zhao *et al.*, 2018) que les poids de modèles ayant été entraînés sur différentes distributions tendent à plus diverger. Ainsi, lors de l'agrégation, certaines mises à jour de paramètres viendront s'annuler. Cela a comme résultat de réduire les performances du modèle global de façon significative.

Il existe différentes variantes de biais dans les données qui peuvent rendre le jeu de données non uniforme. Premièrement, si le jeu de données ne contient pas le même nombre d'échantillons pour chaque classe, cela mène à un jeu de données non balancé ce qui peut biaiser l'entraînement (Zhu, Xu, Liu & Jin, 2021). Il est aussi possible que certains entraîneurs aient une préférence d'étiquetage, c'est-à-dire qu'ils ne seront pas d'accord avec les autres entraîneurs sur l'étiquette d'une donnée.

C'est pour cette raison qu'il est presque impossible de différencier un acteur avec un biais dans ses données d'une attaque d'empoissonnement (Jeong, Son, Lee, Hyun & Chung). En effet, un attaquant pourrait simplement échanger les étiquettes entre les échantillons de deux classes et faire son entraînement normalement pour empoisonner un modèle de façon très subtile. De plus, comme les entraîneurs ne connaissant pas toujours le domaine des données, il n'est pas nécessairement possible d'exclure certains entraîneurs basés seulement sur quelques heuristiques. Par exemple, un approche comme dans (Guo *et al.*, 2021) peut exclure des entraîneurs légitimes et ainsi priver les réseaux de leurs connaissances. En effet, ses données potentiellement uniques dans le réseau pourraient peut-être permettre d'améliorer considérablement le modèle. Ainsi,

il ne faut pas prendre à la légère l'exclusion d'un modèle. Notre approche isole cet entraîneur dans une branche, mais le laisse tirer avantage des autres mises à jour du réseau. Les autres entraîneurs seront libres de rejoindre cet entraîneur si le modèle s'avère plus performant.

1.2.4 Apprentissage multitâches

L'apprentissage fédéré d'un jeu de données distribuées non uniformément est considéré comme un apprentissage multitâche. En effet, prenons par exemple deux entraîneurs qui entraînent un classificateur. Le premier possède seulement des photos de chat et de chien alors que le deuxième a des photos de serpent et d'oiseau. Les deux entraîneurs apprennent une tâche différente, mais elles partagent des ressemblances. Ainsi, certaines connaissances acquises d'une tâche peuvent être appliquées à l'autre tâche. Il peut donc être plus rapide et donner de meilleures performances d'utiliser un modèle déjà entraîné pour une tâche connexe plutôt que d'entraîner deux modèles distincts.

L'apprentissage de multiples tâches dans le même réseau peut s'avérer difficile, car lorsqu'on essaie d'entraîner un modèle sur une tâche de façon séquentielle, une tâche après l'autre, le réseau de neurones tend à oublier comment remplir les tâches précédentes Kirkpatrick *et al.* (2017). De façon similaire, les modèles globaux peuvent aussi oublier lorsqu'ils sont « fine tuned » localement (Wang *et al.*, 2019). Ceci a pour effet que le modèle local perd de sa capacité à généraliser, mais devient meilleur sur les données locales.

1.3 Conclusion

L'arrivée de l'intelligence artificielle dans la vie de tous les jours vient aussi avec la collecte de données massive. L'apprentissage fédéré est une solution pour éviter cette collecte, elle vient cependant avec son lot de problèmes. En effet, l'impossibilité de partager les données crée beaucoup de problèmes qui empêchent l'adoption de ces techniques respectueuses de la vie privée. En effet, des problèmes propres à l'apprentissage fédéré comme la divergence des poids

sont difficiles à résoudre. Le milieu doit donc développer ces propres techniques efficaces et respectueuses des exigences des entraîneurs.

CHAPITRE 2

ANALYSE

L'objectif de ce chapitre est de donner un fondement aux décisions architecturales de la solution. Dans un premier temps, ce chapitre aborde la décision de structurer l'évolution des modèles à l'aide d'un graphe. Dans un deuxième temps, une analyse d'une variété de scénarios d'apprentissage fédéré qui permet de mieux comprendre les relations entre les différents paramètres.

2.1 Données non-uniforme et acteurs malicieux

Il est généralement très difficile de faire de la prévention contre les attaques d'empoisonnement puisque les entraîneurs doivent généralement défendre à l'aveugle, car ils n'ont pas accès aux données des autres entraîneurs. Par exemple, l'attaquant peut entraîner un modèle à partir de données vraies, mais mal étiquetées pour empoisonner le modèle. Cependant, cet attaquant pourrait aussi être un simple entraîneur ayant une gamme d'échantillons très différente ou même dont le reste du réseau ignore qu'elles puissent exister. Par exemple, des applications d'entraînement collectent des données de gens très sportifs qui collaborent pour entraîner un modèle de prédiction. Puis, un hôpital collecte le même type de données lors de tests médicaux et décide de participer. Ainsi, ses mises à jour de modèles pourraient sembler suspectes puisque son modèle est entraîné sur une distribution très différente.

Le réseau se retrouve face à un dilemme : inclure ou exclure sa mise à jour à l'agrégation. D'un côté, le réseau inclut la mise à jour qui peut potentiellement être malicieuse et réduire les performances du modèle. De l'autre côté, le réseau se prive de nouvelles connaissances qui pourraient potentiellement améliorer son modèle.

L'apprentissage fédéré laisse cette décision à un serveur qui n'a pas connaissance des données, alors qu'il semblerait plus intuitifs de laisser les entraîneurs déterminer de l'aggrégation puisqu'ils possèdent les données. Ainsi, ils peuvent déterminer quels sont les meilleures mises à jour et modèle à l'aide de leur données. Cependant, entraîner un seul modèle global n'est probablement

pas en mesure de satisfaire l'ensemble des entraîneurs, puisque l'exclusion de modèles de l'agrégation risque d'exclure de l'information importante ou de ralentir l'entraînement pour les autres entraîneurs.

2.2 Modèles biaisés

Les modèles biaisés sont intéressants puisqu'ils apprennent simplement à optimiser leur réponse à partir des données. Ainsi, il est possible de dire que ce ne sont pas les modèles qui sont biaisés, mais plutôt les données.

Prenez par exemple le cas suivant de trois classificateurs binaires entraînés avec des données inconnues. Lorsqu'ils sont testés sur votre jeu de données, ils obtiennent les précisions suivantes : 1%, 50% et 90%. Il serait instinctif de dire que le pire modèle est celui obtenant 1% de précision, cependant le pire modèle est le deuxième puisqu'il obtient le même taux qu'une classification aléatoire. Alors que le premier classeur a seulement un biais dans son étiquetage, car si on inverse ses réponses, il obtiendra une précision de 99%, ce qui est supérieur au troisième modèle.

Cet exemple fictif sert simplement à démontrer que les modèles biaisés peuvent tout de même être utiles tant qu'ils sont meilleurs qu'une solution aléatoire.

2.3 Analyse d'impacts des paramètres d'apprentissage fédéré

Afin de mieux comprendre les défis qui entourent l'apprentissage fédéré, il est important d'établir les relations entre les différents paramètres et leurs impacts dans le processus d'apprentissage. Pour ce faire, une évaluation a été conduite afin d'éclairer et appuyer les décisions de conceptions de la solution.

Les paramètres étudiés sont le nombre d'échantillons dans le jeu de données de chacun des entraîneurs, le nombre d'époques d'entraînement locales, le nombre d'entraîneurs participant à une ronde et la distribution des données.

Les évaluations ont été conduites avec le modèle LeNet sur le jeu de données MNIST. Il s'agit d'un problème de classification de chiffres écrits à la main.

2.3.1 Distribution uniforme

Les premières évaluations ont été faites en distribuant le jeu de données de façon uniforme aux entraîneurs. 48 sessions d'entraînement fédéré ont été complétées avec les paramètres suivants :

- Nombre d'échantillons : 500, 1000, 2000, 5000.
- Nombre d'entraîneurs : 2, 8, 16, 32.
- Nombre d'époques : 1, 5, 10.

Les résultats sont très pertinents. En effet, la figure 2.1 illustre l'évolution de la précision du modèle en fonction de la taille du jeu de données de chacun des entraîneurs. L'interprétation de ce graphique est qu'il est mieux que chaque entraîneur ait un gros jeu de données pour converger plus rapidement. Cependant, ce graphique peut porter à confusion puisqu'il utilise le numéro de ronde à l'axe des x et que plus un entraîneur a de données, plus la ronde sera longue. Ainsi, en analysant la figure 2.2 qui illustre l'évolution de la performance selon le temps, l'impact de la quantité d'échantillons n'est pas très remarquable. Les réseaux semblent converger au même rythme vers le modèle final et les entraîneurs avec les plus gros jeux de données sont en mesure d'atteindre de meilleures performances finales, car les modèles développent une meilleure capacité à généraliser.

Le graphique 2.3 illustre l'évolution de la précision en fonction du nombre d'époques d'entraînement locales. Plus l'entraîneur fait des époques d'entraînement, plus l'entraînement sera long, mais les performances seront meilleures. Cependant, encore une fois, puisque le temps pour chacune des rondes augmente avec le nombre d'époques, il est plus représentatif d'analyser la figure 2.4 qui illustre l'évolution de la précision en fonction du temps écoulé. Il est possible d'observer que le nombre d'époques n'a pas beaucoup d'influence sur la rapidité de la convergence du modèle. Ainsi, dans un contexte où les données sont distribuées uniformément,

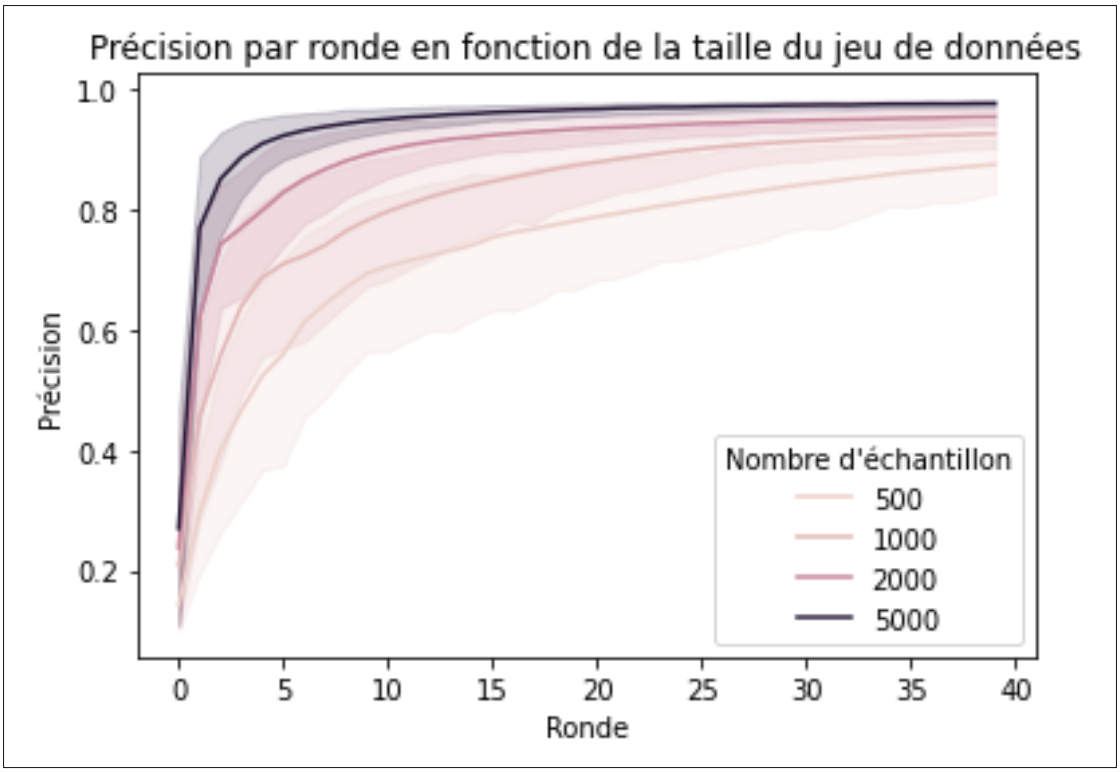

Figure 2.1 Évolution de la précision dans un réseau uniforme en fonction de la ronde pour chaque quantité d'échantillons

il est judicieux d'avoir un plus grand nombre d'époques par ronde afin de réduire les besoins de communications des entraîneurs.

Finalement, la dernière analyse dans le contexte de données distribuées uniformément porte sur le nombre d'entraîneurs présents. La figure 2.5 illustre les résultats de la performance en fonction du nombre de rondes. Contrairement aux expériences précédentes, le graphique par ronde est représentatif, car l'entraînement se déroule en parallèle sur tous les entraîneurs. Les performances sont similaires à travers toutes les expériences. Ainsi, la vitesse de convergence dépend plutôt du travail fait séquentiellement plutôt qu'en parallèle. Cependant, il est intéressant de remarquer que les réseaux avec un plus grand nombre d'entraîneurs ont une plus grande déviation de la moyenne. Cette diférence est probablement introduite, car chacun des entraîneurs introduit un peu de bruit qui s'additionne et devient plus important lorsque le nombre d'entraîneur devient plus grand.

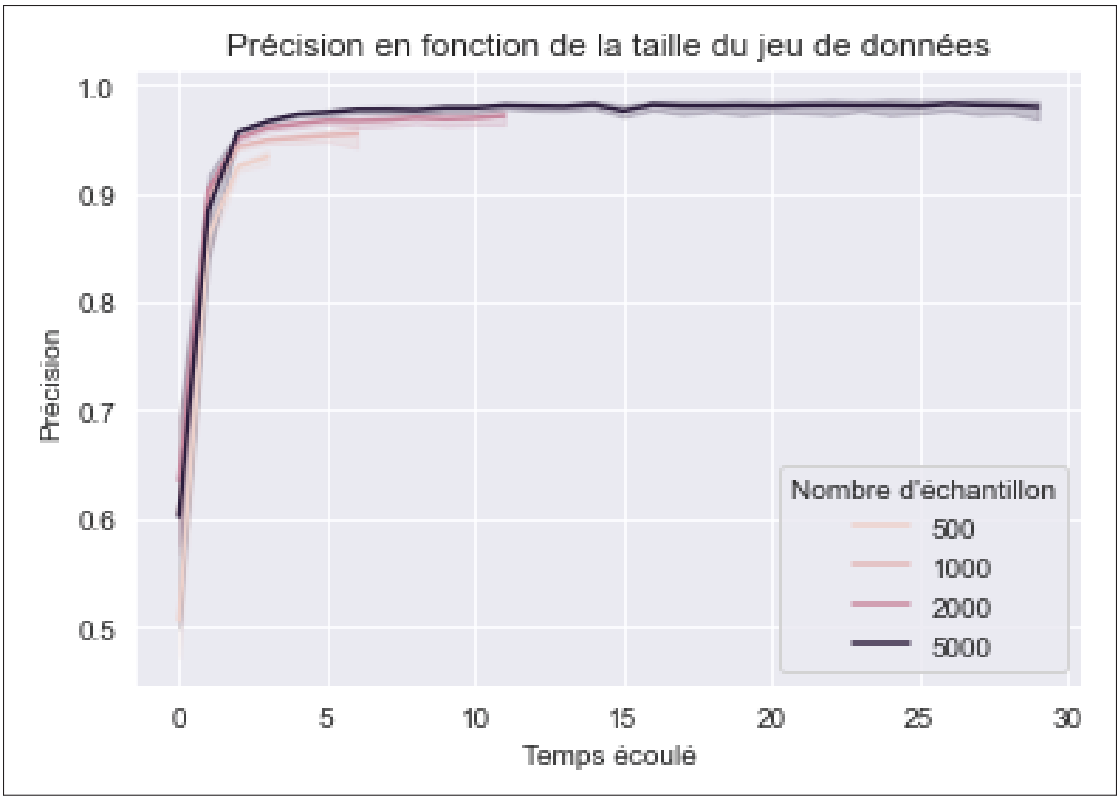

Figure 2.2 Évolution de la précision dans un réseau uniforme en fonction du temps pour chaque quantité de d'échantillons

2.3.2 Distribution non uniforme

Les analyses précédentes ont été refaites 2n distribuant les échantillons selon une distribution gaussienne avec une déviation standard de un afin de créer une distrubution non-uniforme. Ces nouvelles analyses démontrent l'impact de la différence des poids discutés précédemment.

Les différentes valeurs suivantes ont été testées pour un total de huit entraînements :

- Nombre d'entraîneurs : 8 et 32
- Nombre d'échantillons : 500 et 2000
- Nombre d'époques local : 1 et 10

Premièrement, les résultats concernant le nombre d'échantillons sont similaires avec les résultats obtenus avec la distribution uniforme. En effet, comme il est possible de l'observer dans les

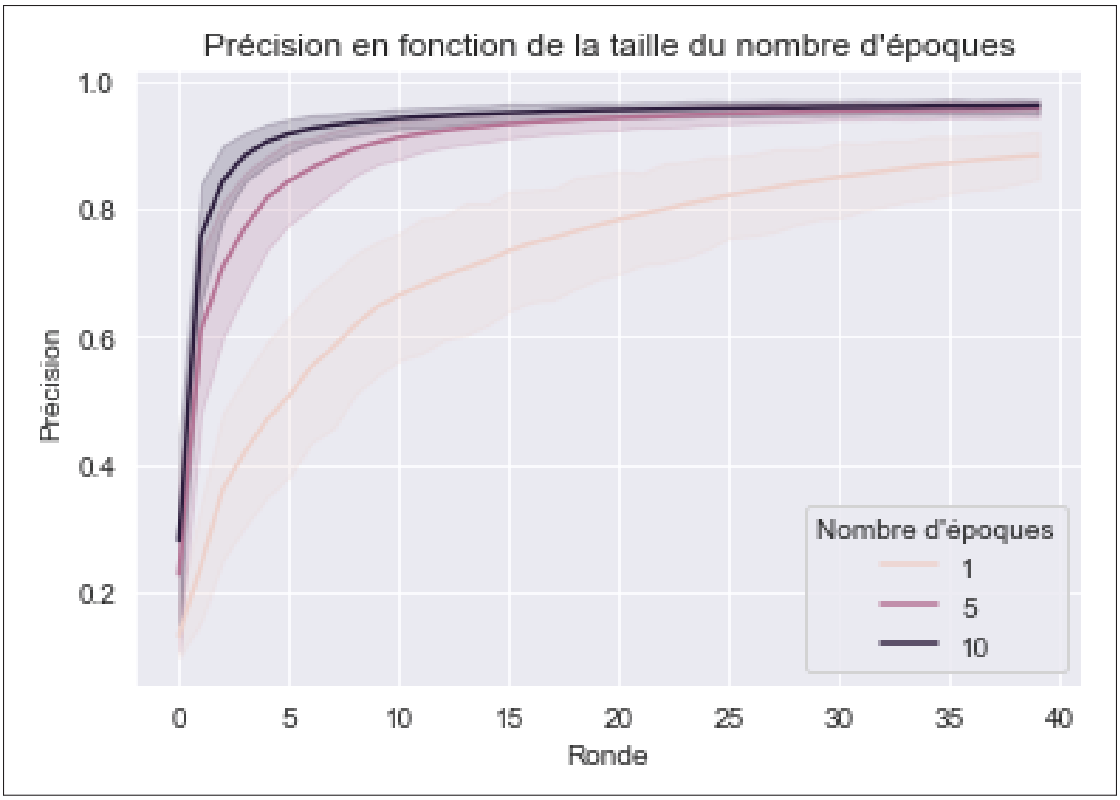

Figure 2.3 Évolution de la précision dans un réseau uniforme en fonction de la ronde pour chaque nombre d'époque d'entraînement local

graphiques 2.6 et 2.7, la quantité de données n'influence pas la rapidité de convergence, mais plutôt le résultat final. Ce qui est normal puisque le réseau peut apprendre sur plus de données et obtenir une connaissance plus générale.

Les données pour évaluer l'impact du nombre d'époques des figures 2.8 et 2.9 sont particulièrement intéressantes. En effet, il est possible de réellement percevoir l'impact négatif de la divergence de poids avec ces graphiques. Puisque les entraîneurs ont des distributions divergentes, plus l'entraîneur travaille son modèle local pendant de nombreuses époques, plus il sera différent des autres entraîneurs. Ainsi, lors de l'agrégation, les mises à jour sont destructrices. Cet effet n'est pas observé dans les graphiques 2.4 avec une distribution uniforme, car les modèles s'optimisent tous pour la même distribution. En réduisant le nombre d'époques d'entraînement, ils font moins de travail, donc il y a moins de divergence entre les modèles. De plus, puisque les

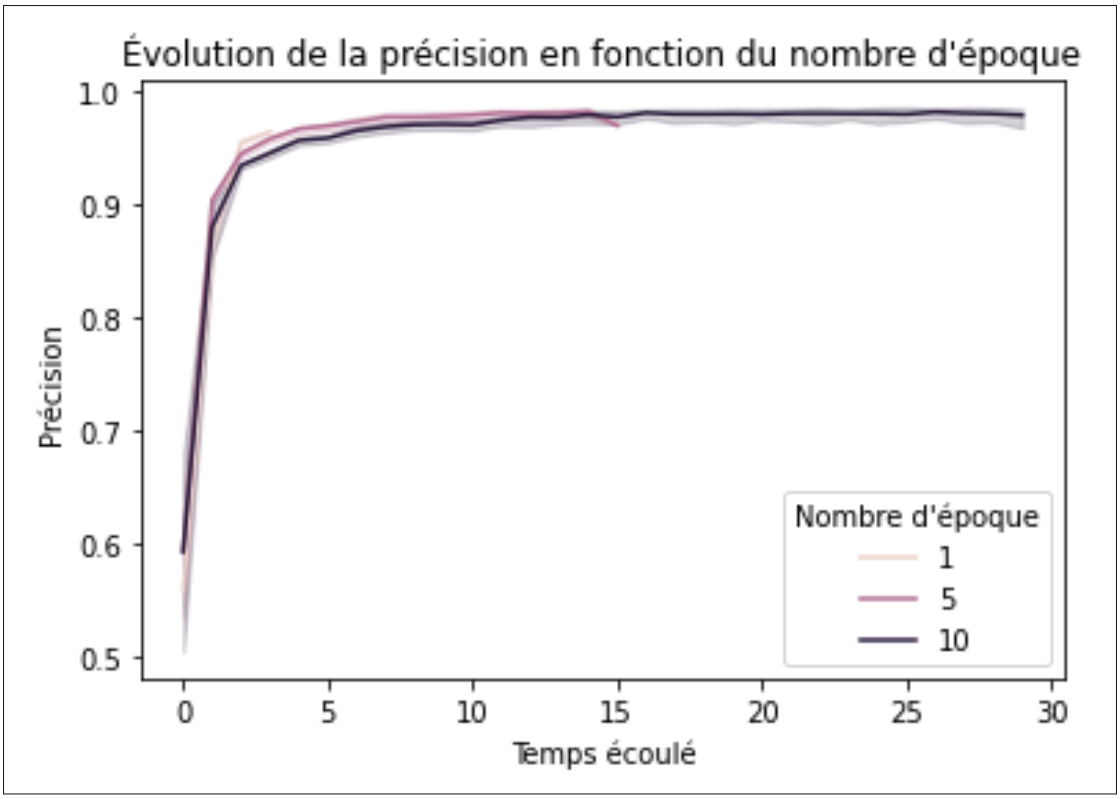

Figure 2.4 Évolution de la précision dans un réseau uniforme en fonction du temps pour chaque nombre d'époque d'entraînement local

entraîneurs synchronisent leurs poids plus fréquemment, il y a moins de mise à jour destructrice et il semble plus facile de trouver une solution commune. Ainsi, comme les expériences l'ont démontré, la divergence des poids ralentit beaucoup la rapidité de convergence. Donc, limiter la divergence des poids est une bonne stratégie pour obtenir une convergence plus rapide.

Finalement, le nombre d'entraîneurs dans un réseau non uniforme ne semble pas avoir d'impact sur la rapidité de convergence tout comme il a pu être observé dans un réseau uniforme. La figure 2.10 illustre les résultats de performance obtenus en fonction de la ronde pour 8 et 32 entraîneurs.

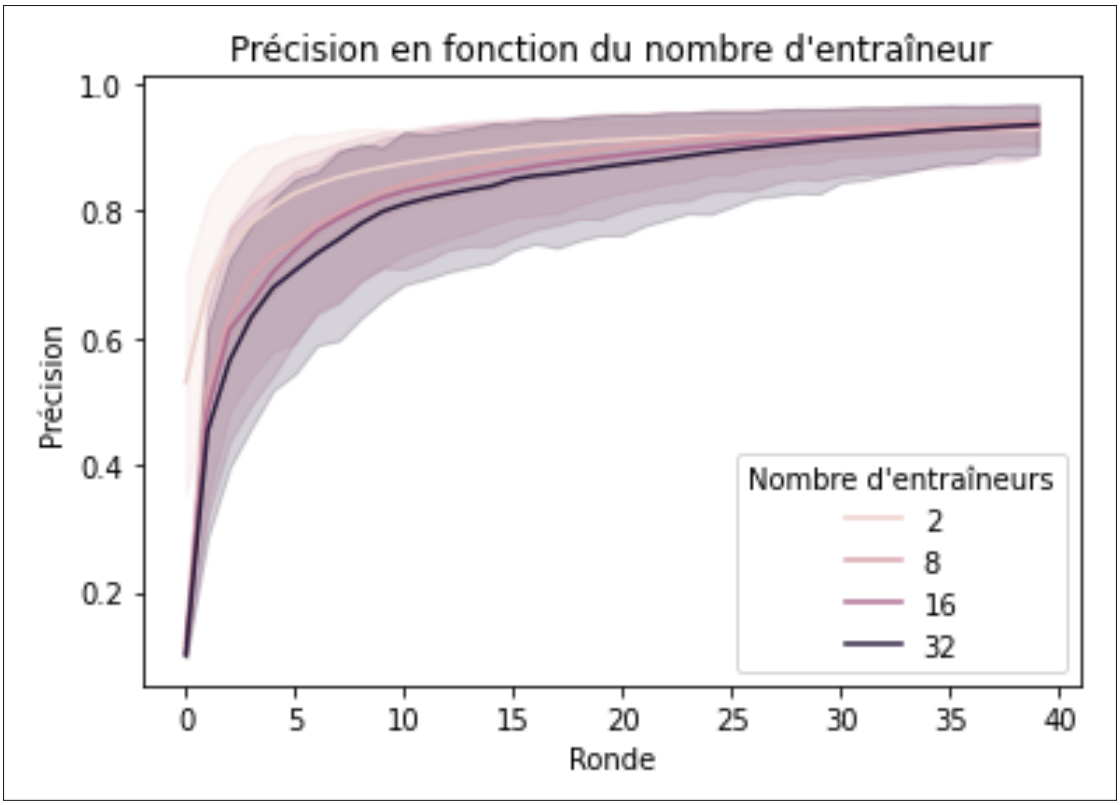

Figure 2.5 Évolution de la précision dans un réseau uniforme en fonction de la ronde dans un réseau de différentes tailles

2.4 Conclusion

L'entraînement de modèle dans un réseau fédéré a son lot de problèmes causés par l'incapacité des entraîneurs de connaître le domaine des données. Ceci rend donc la tâche difficile de différencier les mauvais acteurs des bons acteurs, car il n'est pas possible de différencier avec certitude un acteur malicieux intelligent d'un entraîneur honnête. De plus, la distribution des données non uniforme est un problème majeur qui ralentit la convergence du modèle global. Ainsi, un acteur malicieux doit être géré de la même façon qu'une distribution non uniforme pour s'assurer que le modèle converge de façon efficace.

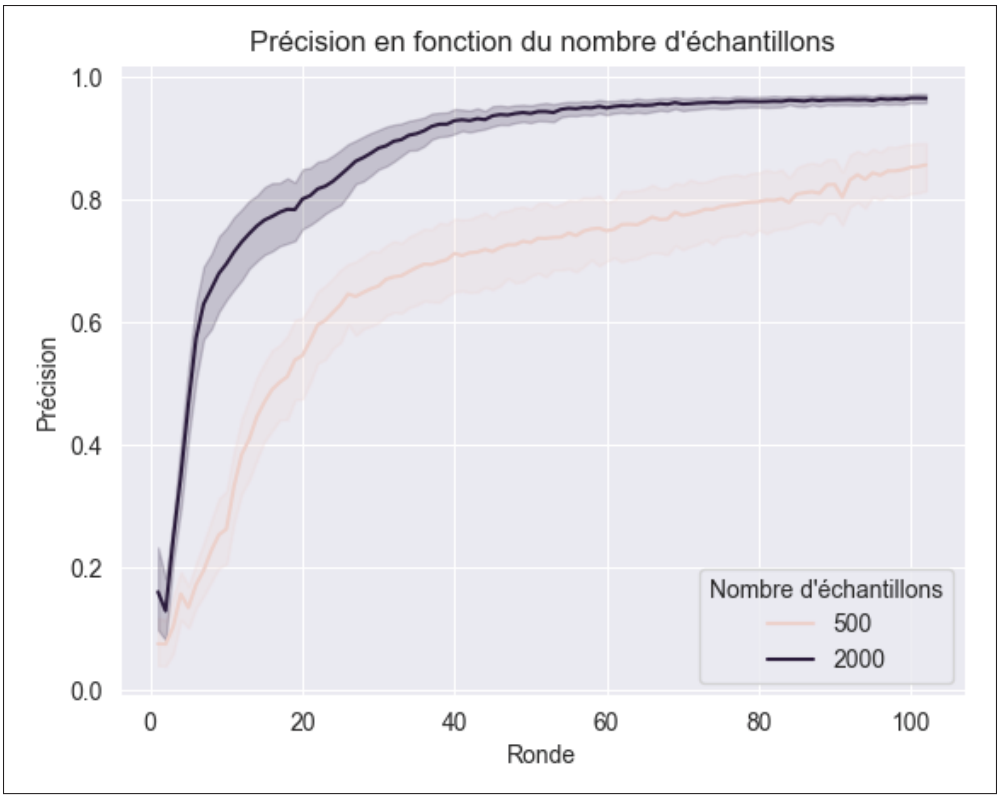

Figure 2.6 Évolution de la précision dans un réseau non uniforme en fonction de la ronde pour chaque quantité d'échantillons

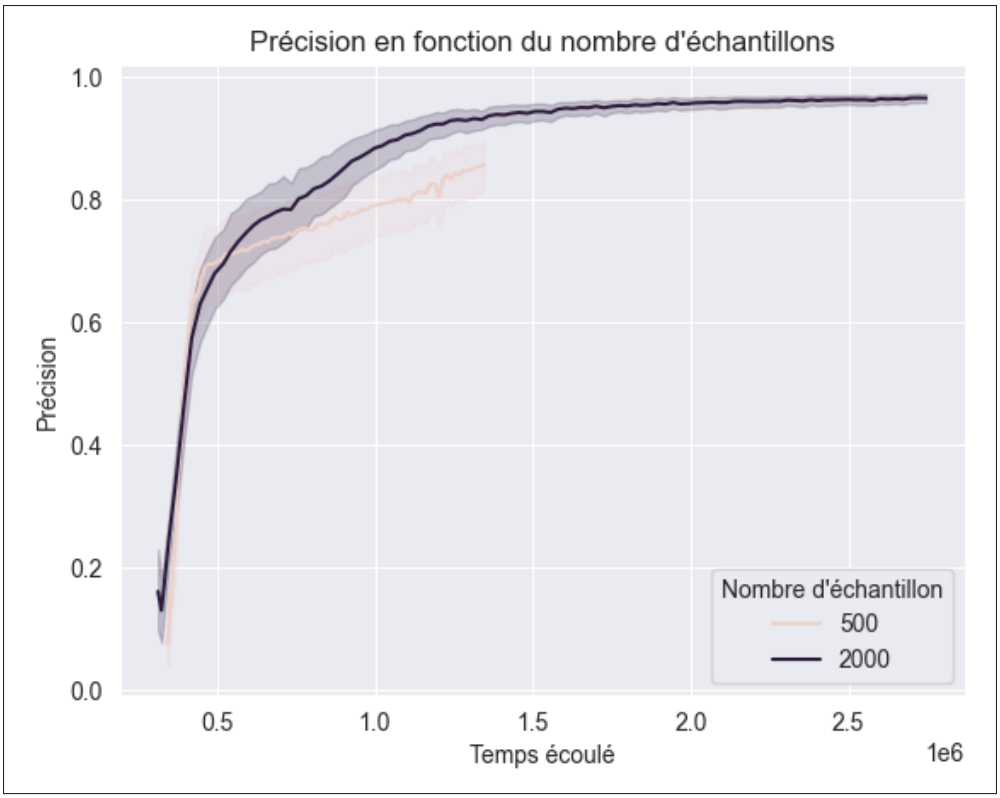

Figure 2.7 Évolution de la précision dans un réseau non uniforme en fonction du temps pour chaque quantité de d'échantillons

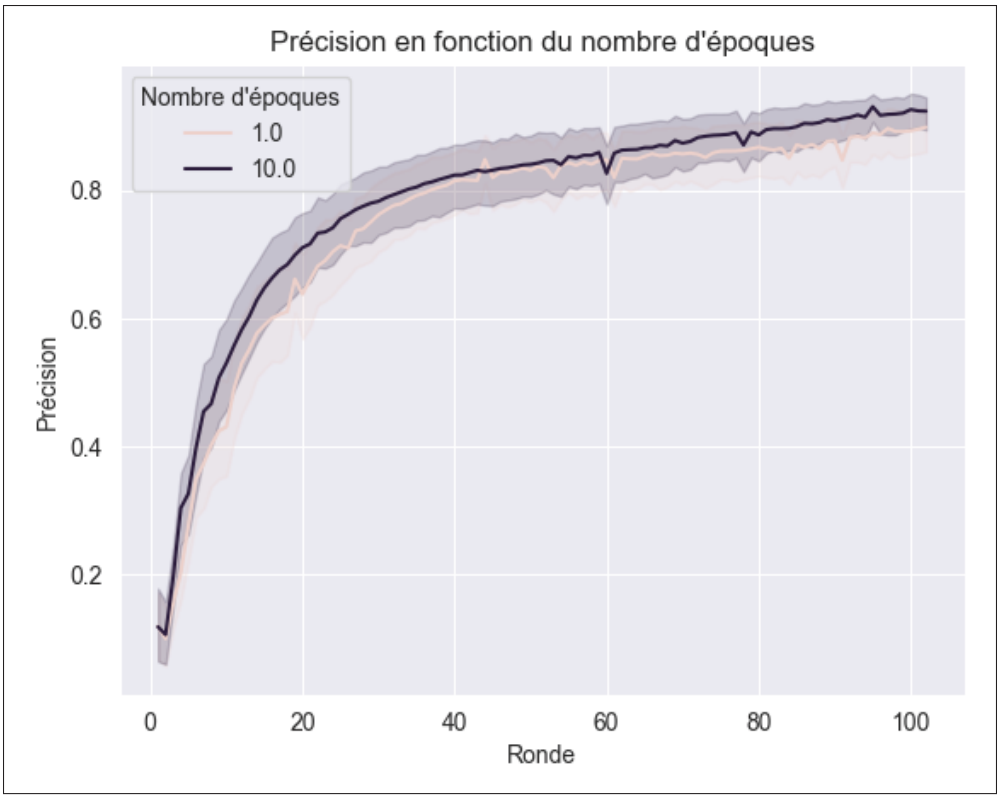

Figure 2.8 Évolution de la précision dans un réseau non uniforme en fonction de la ronde pour chaque nombre d'époque d'entraînement local

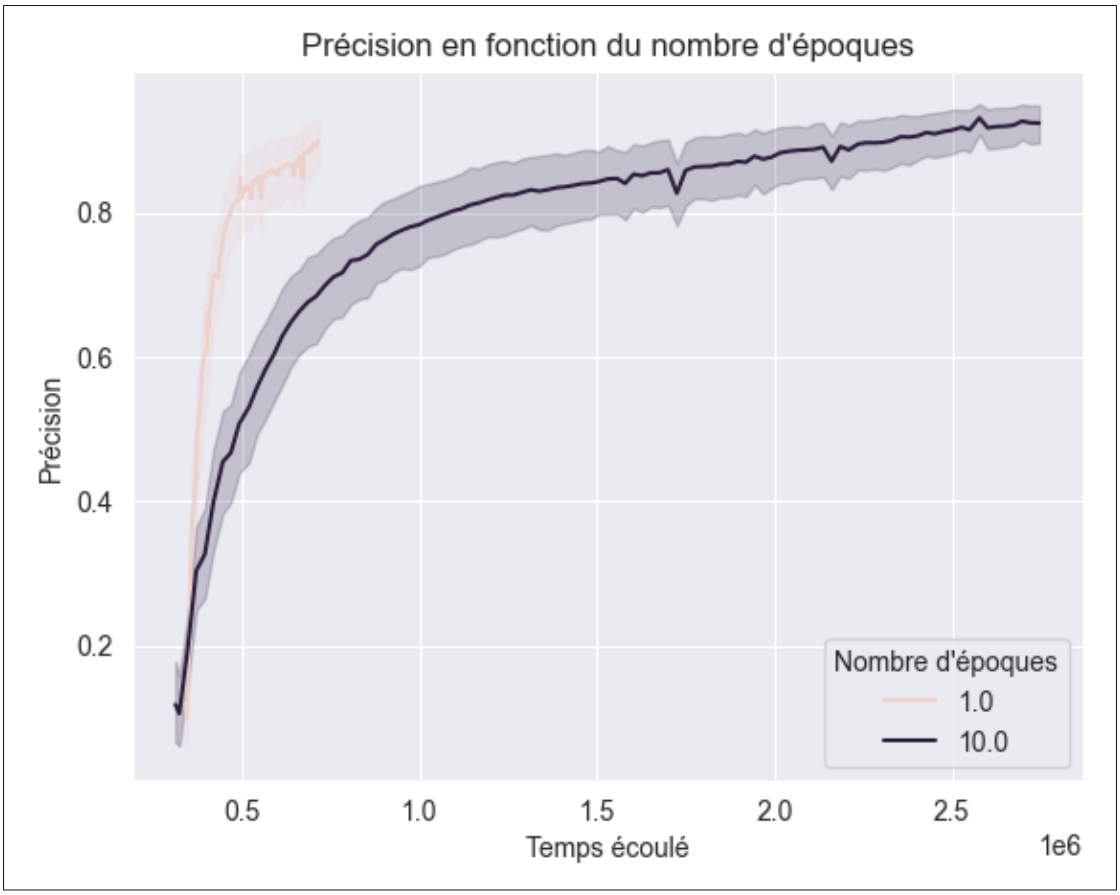

Figure 2.9 Évolution de la précision dans un réseau non uniforme en fonction du temps pour chaque nombre d'époque d'entraînement local

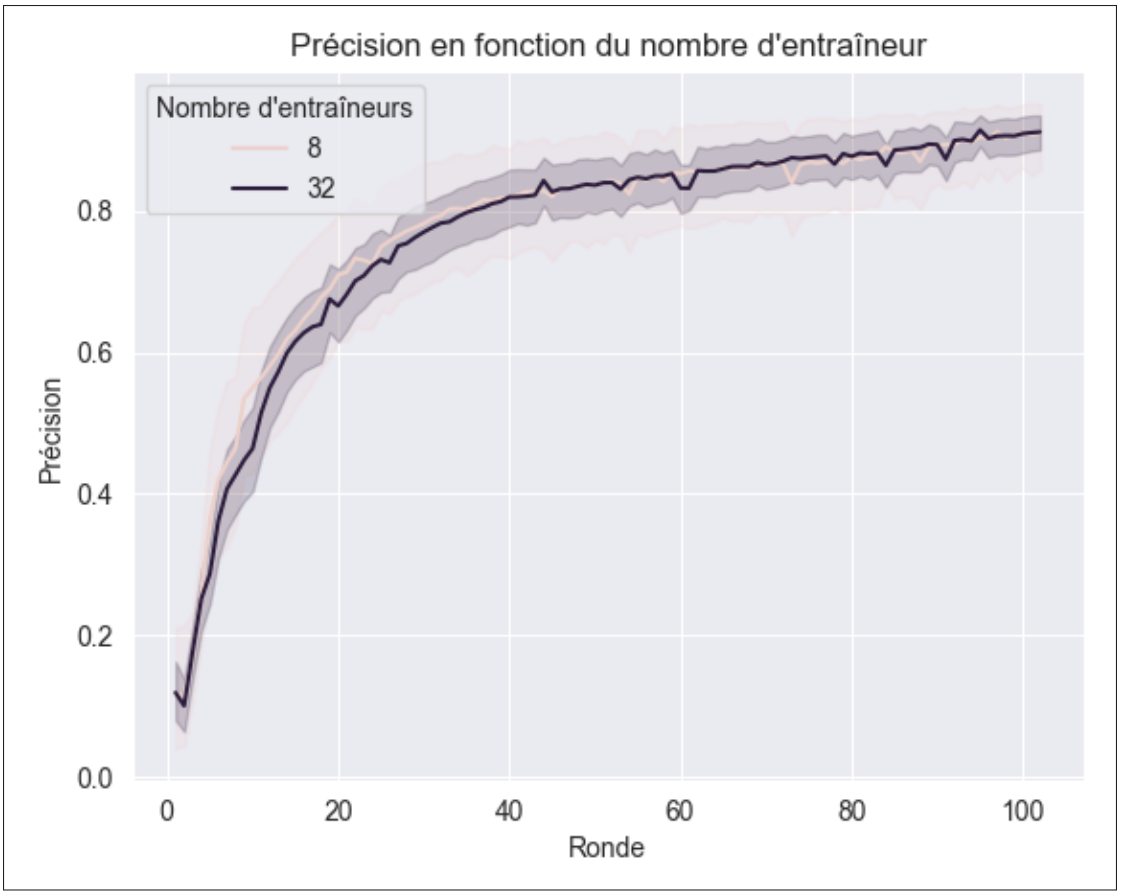

Figure 2.10 Évolution de la précision dans un réseau non uniforme en fonction de la ronde dans un réseau de différentes tailles

CHAPITRE 3

ARCHITECTURE DU SYSTÈME

Ce chapitre aborde l'architecture de la solution. Dans un premier temps, un survol de la solution sera fait, puis les différentes parties principales seront expliquées en détail par la suite.

3.1 Survol de la solution

La solution est un système décentralisé connecté en pair-à-pair dans lequel chaque nœud a le rôle d'entraîneur et d'agrégateur. Pour cette raison, cette solution s'applique principalement à des cas entre silos, car elle requiert beaucoup de bande passante pour communiquer tous les modèles. Les nœuds forment des groupes et chaque groupe entraîne un modèle global. Les groupes sont modélisés dans le système à l'aide d'un graphe acyclique. De plus, les entraîneurs sont libres de joindre autant de groupes qu'ils le souhaitent.

Puisqu'il n'y a pas de serveur central pour l'agrégation, les nœuds doivent s'autoassigner un ou des groupes. Pour ce faire, les nœuds sont libres d'établir leurs propres filtres de sélection de groupe. Les nœuds devront participer à l'entraînement du modèle du groupe, ils doivent donc choisir judicieusement les modèles qui seront les plus bénéfiques pour eux. Les nœuds s'assignant à plusieurs groupes permettent de faire du transfert de connaissance d'un groupe à l'autre menant à des modèles plus généralisés. La figure 3.1 illustre un réseau avec quatre entraîneurs créant trois groupes. Les entraîneurs qui font partis de plusieurs groupes permettent de faire du transfert d'apprentissage d'un groupe à l'autre.

Chaque nœud fait l'agrégation des mises à jour à l'aide de filtres de sélection des mises à jour et publie sa sélection sur le réseau. Chaque combinaison publiée par les nœuds devient une nouvelle branche qui pourra être sélectionnée pour l'entraînement dans les rondes suivantes.

La solution proposée ne considère pas les acteurs malicieux au niveau architectural puisque les nœuds ont la liberté d'exclure des mises à jour en créant une nouvelle branche. Ainsi, il s'agit plutôt d'un détail d'implémentation dans les filtres de sélection qui peut varier d'un entraîneur

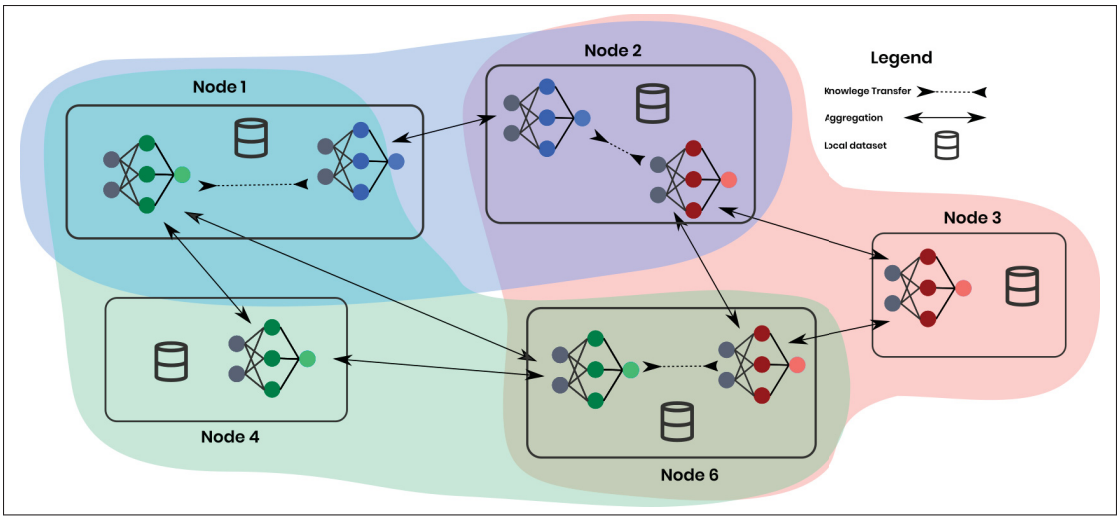

Figure 3.1 Architecture du réseau

à l'autre. En effet, l'architecture est assez flexible pour permettre tous les cas possibles. Par exemple, si un entraîneur est exclu unanimement par le groupe, alors il se retrouvera isolé dans sa branche.

Le processus est divisé en trois étapes :

1. Les entraîneurs sélectionnent les modèles du groupe auxquels ils souhaitent contribuer.
2. Ils entraînent et partagent le modèle avec leurs pairs.
3. La phase d'agrégation consiste à choisir quelles mises à jour seront incluses dans la prochaine version du modèle. Chaque combinaison devient une nouvelle branche.

Il est possible de voir une représentation visuelle du processus dans la figure 3.2. Les trois entraîneurs décident d'entraîner le modèle A. Suite à la sélection du modèle, les entraîneurs vont l'entraîner, puis partager la mise à jour. Lors de la phase de sélection des mises à jour, l'entraîneur 3 décide d'exclure la mise à jour de l'entraîneur 1. Ainsi, deux combinaisons existent, ce qui a pour effet de créer deux branches au modèle A.

La solution sera référée par le terme d'apprentissage multiconfédéré ou MultiConfederated Learning (MCFL) en anglais. Le terme confédération a été préféré, car il est plus représentatif de la solution puisque tous les acteurs sont indépendants, mais se regroupent pour s'entraider à

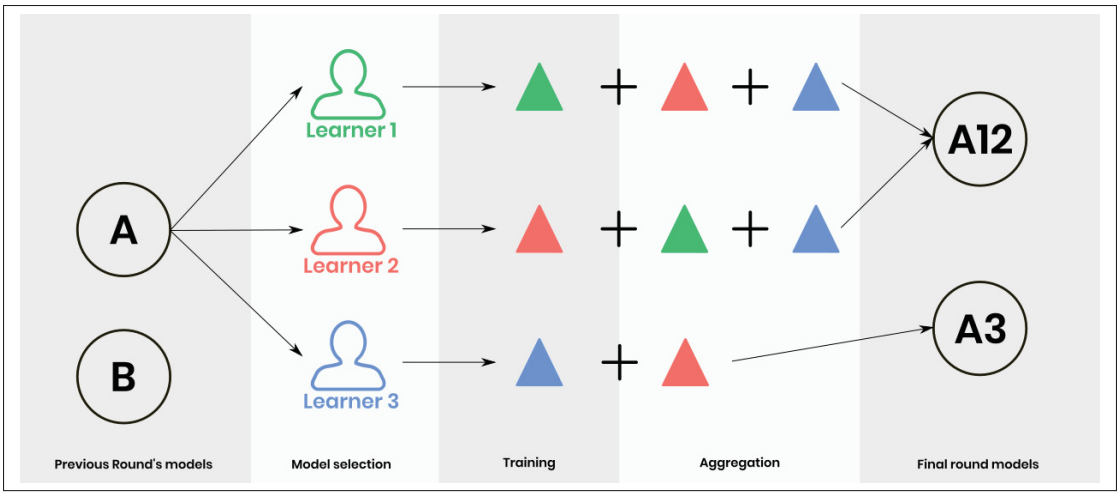

Figure 3.2 Étapes d'un ronde d'entraînement

atteindre un objectif commun sans sacrifier leur indépendance et leur autonomie. Le préfixe multi est ajouté, car la solution orchestre plusieurs apprentissages simultanément.

3.2 Entraînement de plusieurs modèles en simultané

Alors que l'entraînement fédéré se concentre sur un seul modèle qui évolue à chaque ronde, l'architecture proposée structure les modèles dans un graphe afin que plusieurs modèles puissent évoluer en même temps. Un graphe acyclique a été choisi comme structure de données pour faire le suivi de l'évolution puisqu'il s'agit d'une structure très flexible. La figure 3.3 illustre la différence entre le modèle classique et l'approche proposée. De plus, il est relativement simple de traverser un graphe en suivant certaines règles pour sélectionner les modèles à entraîner.

Puisque plusieurs modèles évoluent en même temps, les entraîneurs ne sont pas limités à entraîner un seul modèle. En effet, les entraîneurs peuvent participer à l'entraînement d'autant de modèles qu'ils le souhaitent. Ceci a pour effet de développer certaines spécialités dans des branches tout en étant capable de développer des branches qui sont plus généralisées.

De plus, en adoptant le suivi de modèle par DAG, le réseau n'a plus besoin de prendre une décision binaire dans le cas d'un modèle suspecté d'empoisonnement. En effet, à partir du

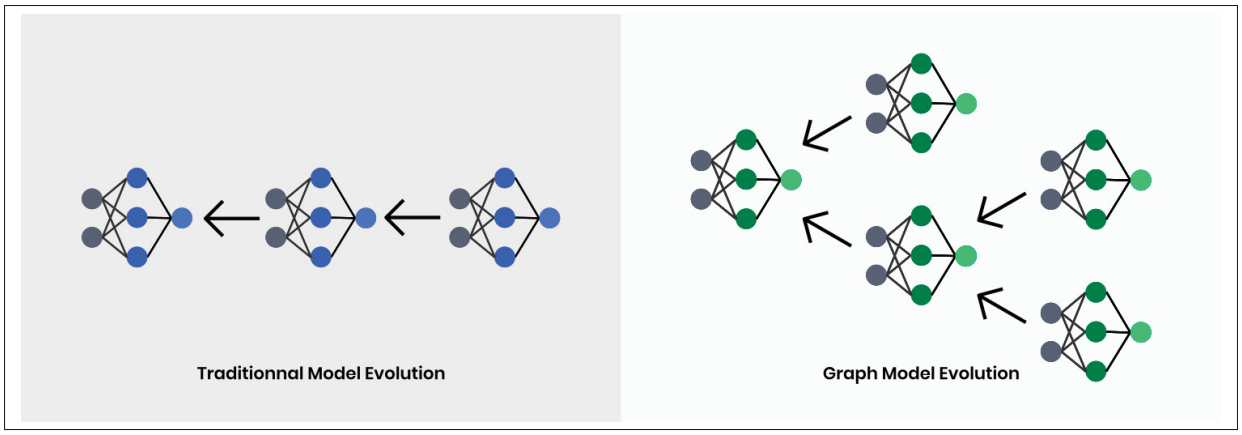

Figure 3.3 Évolution d'un modèle en apprentissage fédéré traditionnel et de l'approche proposée

modèle parent, le réseau peut créer deux branches du modèle. Une première branche inclut le modèle suspect et une deuxième branche n'inclut pas le modèle suspect. Les entraîneurs ont par la suite le choix de contribuer à une seule ou les deux branches.

Cet aspect est important pour garantir que tous les entraîneurs puissent être inclus dans l'entraînement. En effet, l'entraîneur exclu par les autres peut toujours utiliser les mises à jour des autres tout en ajoutant la sienne dans sa propre branche.

Éventuellement, un modèle peut se démarquer comme étant supérieur et cela aura pour effet de faire converger les entraîneurs vers la même branche. Il est aussi possible qu'ils ne convergent pas vers la même branche si les branches ont chacune développé une spécialité qui détériore les performances de l'autre.

Les entraîneurs peuvent appliquer du transfert d'apprentissage, comme dans l'apprentissage multitâche en entraînant une branche qui n'est pas leur favorite. Ainsi, dans un entraînement avec suivi par DAG, les entraîneurs devraient être encouragés à participer à l'entraînement d'autres branches afin de rendre le modèle plus générique. De plus, plus un modèle est générique, plus les entraîneurs convergeront vers cette branche.

3.3 Filtres de sélection

Il est important de considérer tous les entraîneurs comme étant souverains. C'est-à-dire qu'ils sont indépendants et devraient être les seuls en mesure de prendre des décisions les concernant puisqu'ils sont les seuls à avoir connaissance de leurs données. C'est pour cette raison que l'architecture de la solution propose la mise en place de filtres de sélection modulable qui peuvent être modifiés et ajustés aux besoins spécifiques de chacun des entraîneurs. De plus, les filtres peuvent être additionnés pour combiner certains effets.

Deux types de filtres ont été établis pour donner la flexibilité requise :

1. Filtre de sélection de groupe : ces filtres sont appliqués lors du parcours du graphe pour déterminer quels sont les modèles auxquels l'entraîneur devrait participer.
2. Filtre de sélection de mises à jour : ils sont appliqués lors de l'agrégation aux mises à jour reçues. Les mises à jour passant le filtre sont par la suite agrégées.

3.4 Processus d'entraînement

L'entraînement se déroule en plusieurs étapes. Premièrement, le nœud doit sélectionner le ou les modèles qu'il souhaite entraîner en appliquant ses règles de sélection de modèles. Une fois qu'il a déterminé quels modèles sont à entraîner, il peut procéder à l'entraînement. L'architecture ne considère pas la méthode d'entraînement et laisse le choix à l'entraîneur de l'implémentation à utiliser. Lorsque l'entraînement est terminé, l'entraîneur peut partager la mise à jour avec son groupe. En parallèle à l'entraînement, le nœud écoute et télécharge les mises à jour envoyées à son groupe. Lorsqu'il a récolté les mises à jour et est prêt à faire l'agrégation, il applique les filtres de sélection des mises à jour. Avec les mises à jour restantes, il applique l'algorithme d'agrégation et publie la combinaison des mises à jour. Les nœuds ajoutent alors le nouveau modèle résultant au graphe. Un diagramme de flux illustre le processus d'entraînement à la figure 3.4.

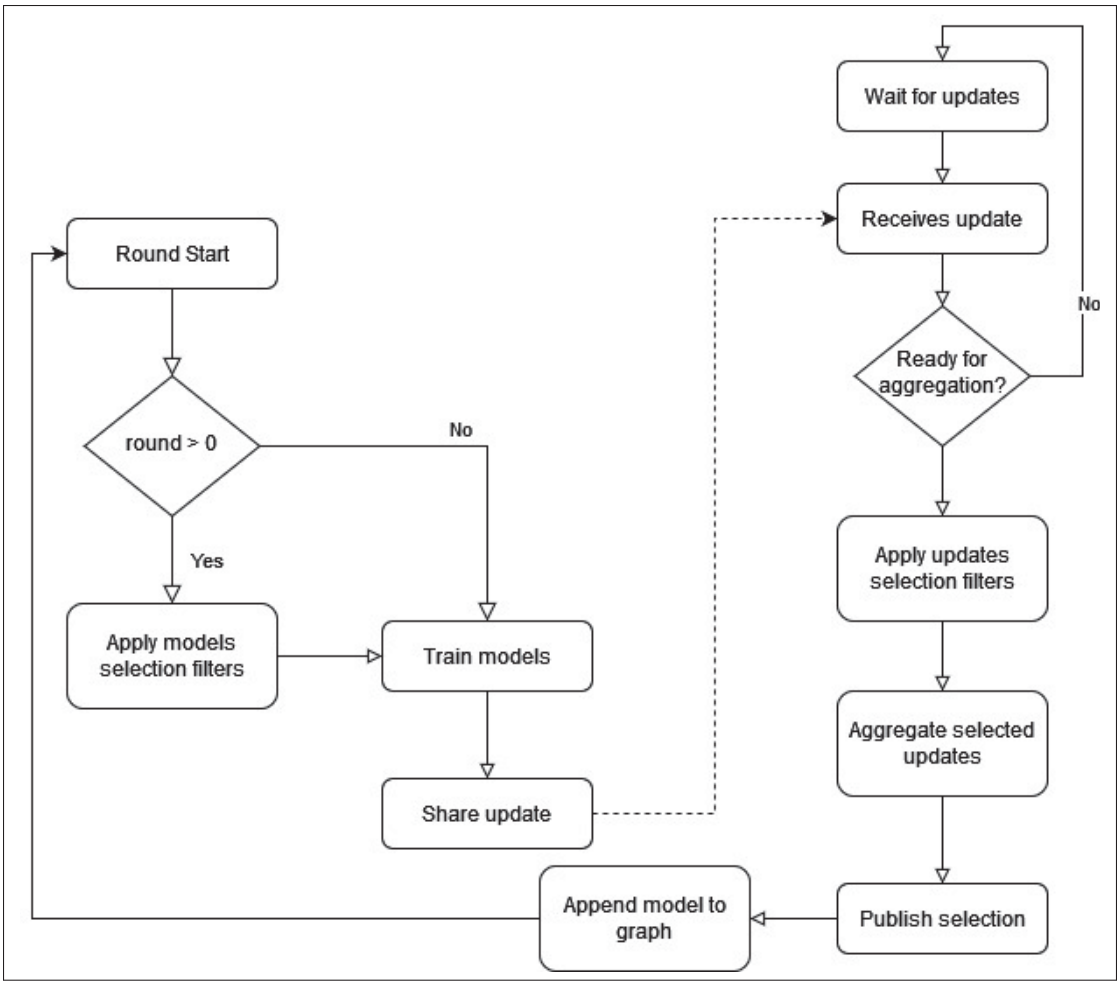

Figure 3.4 Processus d'entraînement

3.5 Réutilisation des modèles pour un apprentissage multitâches

Les différentes branches qui se développent au fil de l'entraînement ont certains biais puisqu'ils ne sont pas entraînés par tous les entraîneurs. Il s'agit d'une décision consciente faite dans l'optique de fournir des modèles spécialisés qui pourront mieux performer et atteindre leur sommet de performance plus rapidement. De plus, il est possible de réutiliser des modèles pour créer de nouveaux modèles plus généralisés. Avec cette vision, il faut considérer les modèles spécialisés comme de petits modules pour composer de plus grands modèles. Par exemple, la figure 3.5 propose une architecture pour un classificateur qui utilise les prédictions de trois autres classificateurs pour faire sa propre classification. Il est possible de créer ce genre de

classificateur à deux niveaux en utilisant de simples algorithmes comme un arbre de décision ou un SVM qui requiert moins de données à entraîner qu'un réseau de neurones.

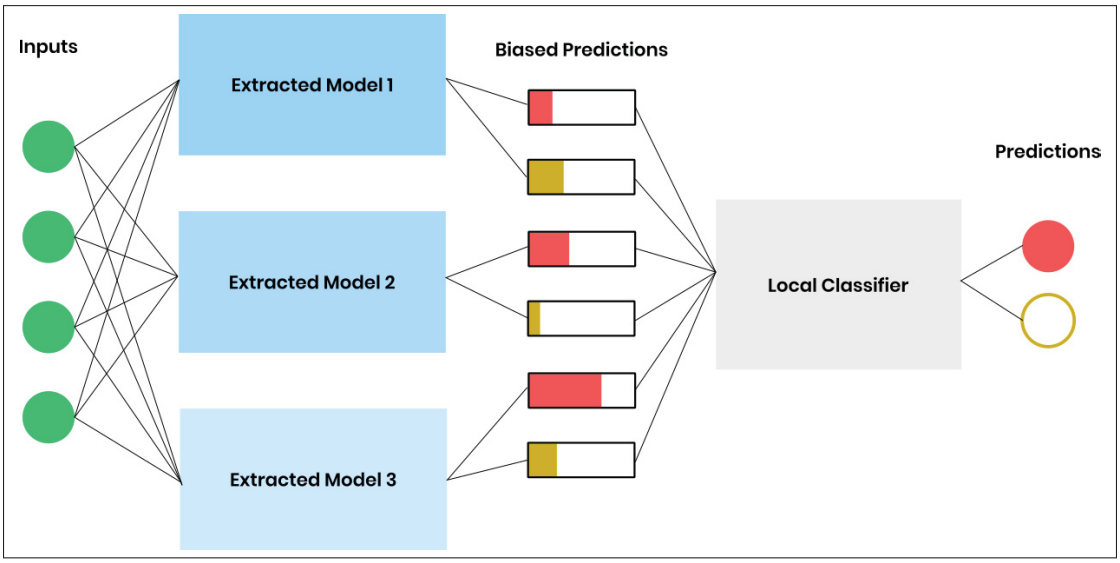

Figure 3.5 Exemple d'architecture d'un classificateur utilisant plusieurs modèles

Pour entraîner ce type de modèle, il suffit de faire l'inférence sur les entrées avec tous les modèles puis de concaténer ces sorties qui deviendront les entrées du modèle classificateur local. L'entraînement se déroule comme à l'habitude avec un jeu de données étiqueté. PCA peut être utilisé pour réduire les dimensions si plusieurs modèles sont extraits.

De plus, comme il n'y a pas d'entraînement fait sur les modèles biaisés, ils n'oublient rien de leur apprentissage d'origine en plus de fournir une redondance qui pourrait être bénéfique à la prédiction finale.

3.6 Architecture modulable basée sur les évènements

Dans l'optique d'apporter la flexibilité requise pour tous les entraîneurs, l'architecture a été développée autour du patron de conception de l'observateur. En effet, un module gérant les évènements est initialisé et permet aux autres modules de s'enregistrer et d'observer les évènements qui sont émis par les différents modules. Lorsqu'un évènement est observé, le module peut exécuter une fonction.

De plus, cette approche simplifie la gestion du multithreading en ayant seulement besoin de protéger le module d'évènements pour les exécutions concurrentes. Le diagramme de séquence de la figure 3.6 est un exemple de cette gestion. Le module d'entrées et sorties, qui doit être sur son propre fil pour rester connecté avec ses pairs, envoie les messages reçus comme un évènement. En même temps, l'entraînement a lieu sur un autre fil d'exécution et envoie simplement un évènement lorsque l'entraînement est terminé. Ainsi, le fil principal n'est jamais bloqué et peut toujours traiter les évènements et continuer de communiquer librement. Les évènements sont traités de façon séquentielle en ordre d'arrivée à l'aide d'une queue d'évènements.

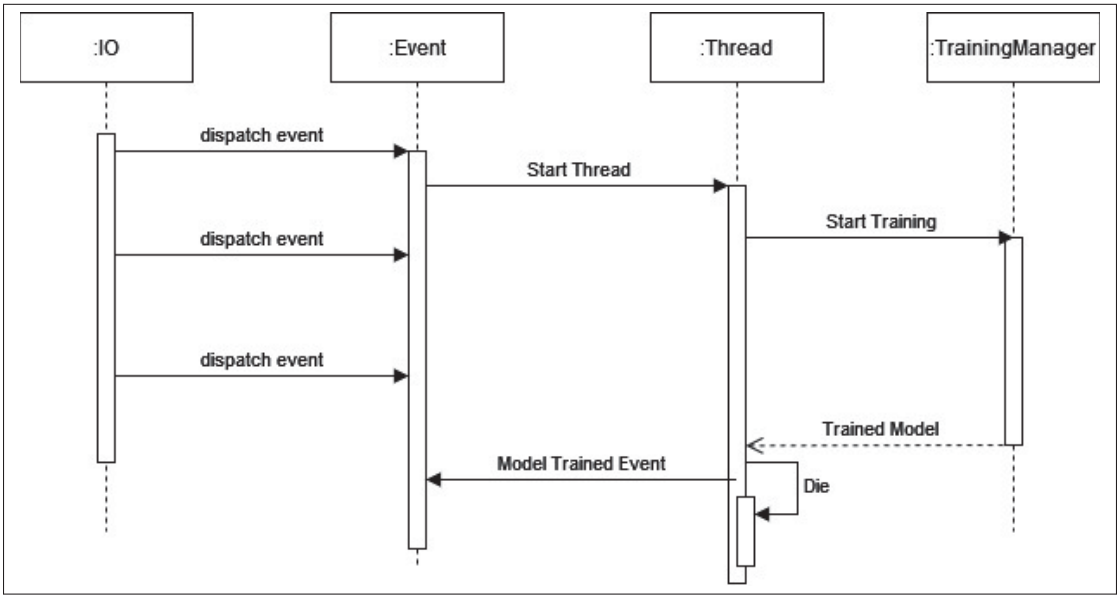

Figure 3.6 Diagramme de séquence d'évènements illustant la communication de modules avec le module d'évènements

L'architecture est composée de plusieurs modules qui forment le diagramme de dépendances de la 3.7.

Module d'état (States) : Il s'agit du module traitant les évènements. Il possède aussi un état dans lequel les différents modules peuvent enregistrer de l'information. L'état permet aussi de garder des références aux objets pour que les différents modules aient accès aux informations enregistrées dans les autres modules.

Module de configuration (Config) : Le module de configuration permet de configurer le nœud. Elle peut être lue à partir des arguments de la ligne de commande, de l'environnement, de l'OS ou d'un fichier .env. Par la suite, tous les modules peuvent lire les configurations à partir du module de configuration.

Module de données (Datasets) : Le module de jeu de données s'occupe de charger le jeu de données requis pour l'entraînement. Le jeu de données chargé dépend de la configuration.

Module de communication (Client) : Ce module est responsable de communiquer avec les autres nœuds. Il dépend du module de configuration pour déterminer comment les nœuds vont communiquer.

Module de ligne de commande (CLI) : La ligne de commande permet de connaître l'état du nœud en exécutant des commandes. Module de journalisation (Logging) : La journalisation permet de garder un historique de l'exécution utile pour le déverminage et l'extraction de données.

Module d'entraînement (Training) : Le module d'entraînement s'occupe d'entraîner les modèles.

Module FedAvg : Ce module implémente l'algorithme d'agrégation FedAvg de façon décentralisée. Il s'agit de l'implémentation de référence à laquelle sera comparée la solution proposée.

Module MultiConfederatedLearning : Il remplace le module FedAvg, ce module implémente la solution proposée.

3.7 Conclusion

L'architecture flexible proposée permet de créer un réseau d'entraîneurs souverains. Cette architecture donne la liberté requise à tous les entraîneurs de prendre leurs propres décisions pour répondre à leurs besoins en priorité. En effet, le suivi de l'entraînement à l'aide d'un

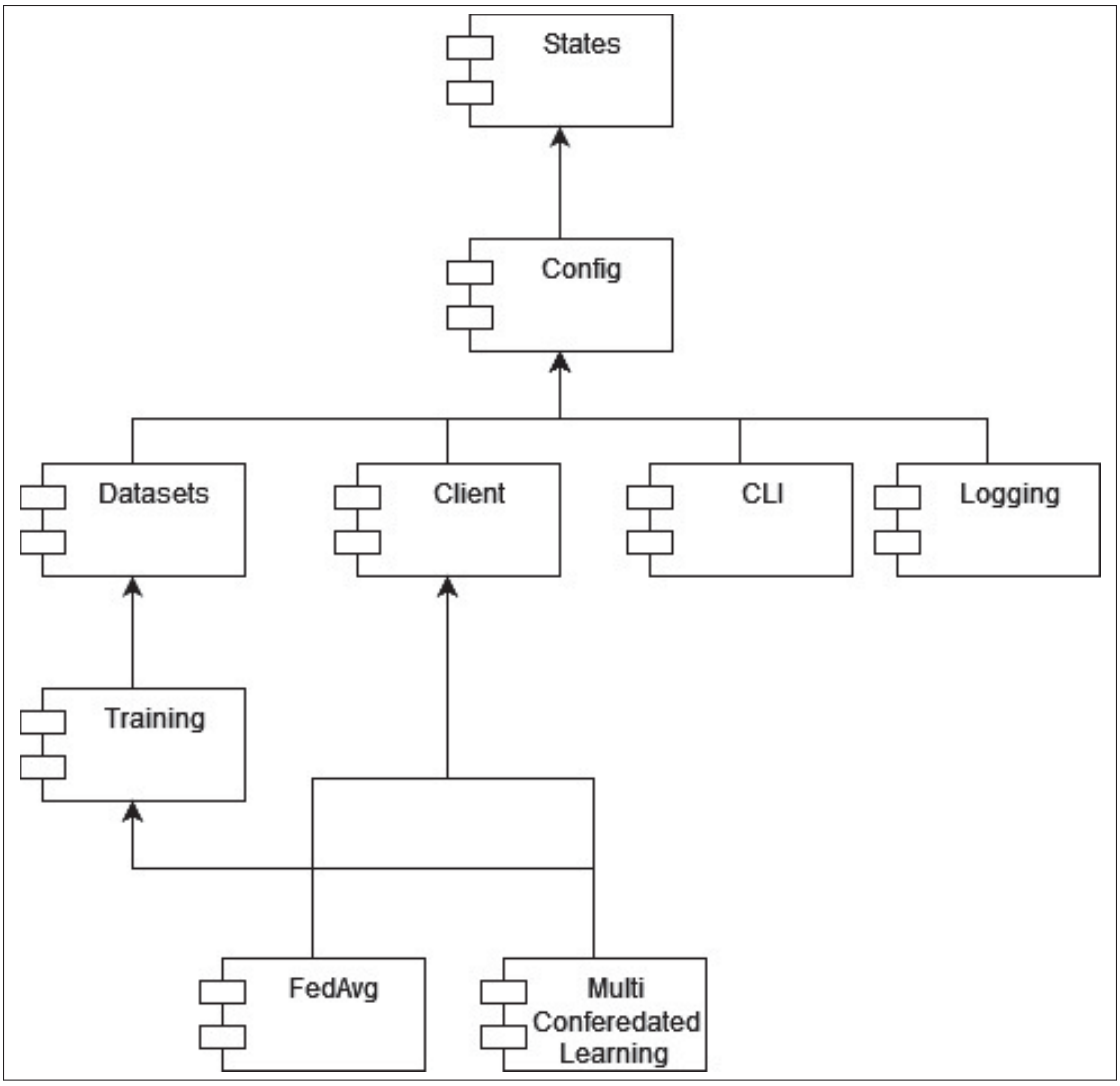

Figure 3.7 Diagramme de modules

graphe combiné avec les filtres permet à tous les modèles de co-exister sur le réseau. De plus, l'approche modulaire rend plus facile l'adaptation du nœud aux méthodologies de l'entraîneur.

CHAPITRE 4

IMPLÉMENTATION

Alors que le chapitre précédent se consacrait à l'architecture soit les mécaniques et structure de la solution, ce chapitre est dédié aux détails d'implémentations. En effet, les thèmes de l'entraînement, des filtres, de la communication et des technologies utilisées seront abordés dans ce chapitre.

4.1 Entraînement synchrone par ronde

Puisque la solution est demandante au niveau de la bande passante, car chaque nœud doit envoyer et recevoir plusieurs modèles, elle est plus orientée vers des cas d'utilisation d'apprentissage entre silos. Ainsi, on peut tenir pour acquis que les nœuds seront maintenus par des organisations sur des machines puissantes sans problèmes de connexion. Pour ces raisons, l'entraînement est synchrone et se déroule par ronde. En effet, les entraîneurs attendent d'avoir reçu la combinaison de mise à jour de tous les entraîneurs avant de passer à la prochaine ronde.

4.2 Filtres de sélection de modèle

Au début de chaque ronde, les entraîneurs doivent choisir les modèles qu'ils vont entraîner. Pour ce faire, les entraîneurs assignent un score basé sur les règles de sélections. Lorsque tous les modèles sont évalués, l'entraîneur sélectionnera les $\sqrt{n}$ meilleurs modèles. Ce nombre a été choisi pour éviter que le temps par ronde augmente linéairement avec le nombre de modèles disponibles. Cependant, cela ne limite cependant pas le nombre maximal de groupes dans le réseau et laisse les nœuds participer à l'entraînement de plusieurs branches pour faire du transfert d'apprentissage. Cette limite pourrait aussi être personnalisée individuellement par les noeuds.

Dans cette implémentation, deux règles ont été établies. La première utilise les performances empiriques des modèles de la ronde précédente. En effet, l'entraîneur établit une métrique soit la précision (Équation 4.1) pour un problème de classification ou la perte (Équation 4.2) pour

l'entraînement d'un autoencodeur et test le modèle avec ses données locales réservées pour la phase de test.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$MeanSquaredError = \frac{1}{N}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.2}$$

La deuxième règle est une règle de popularité pour encourager les entraîneurs à réutiliser une branche. Cette règle multiplie le score obtenu par $\sqrt{p}$ où p est le nombre de mises à jour reçues. La racine carrée a été choisie pour éviter d'augmenter linéairement le nombre de modèle entraînés par rondes.

Le pseudo-code illustrant l'implémentation des deux règles peut être lu dans l'encadré Algorithme 4.1.

Algorithme 4.1 Selection des modèles à entraîner

Input : Modèles candidats models
Output : Modèles sélectionnés

```
1 R ← testModels(models);
2 for i in range(models.length) do
3 |   popScale ← sqrt(popularity(models[i]));
4 |   modelScores[i] ← modelResults[i] * popScale;
5 end for
6 rankedModels ← models.orderBy(modelScores);
7 nModels ← sqrt(models.length);
8 return rankedModels[0 : nModels];
```

4.3 Filtres de sélection des mises à jour

Au niveau des filtres de sélection, les règles optimisent le choix pour limiter la divergence des poids afin d'avoir des modèles performants rapidement. Pour cette raison, les règles calculent la divergence des poids à l'aide de l'équation 4.3 tirée de (Chandran *et al.*, 2021).

$$WeightDivergence = \frac{||W^{PeerUpdate} - W^{LocalUpdate}||}{W^{LocalUpdate}} \tag{4.3}$$

Le filtre calcule la divergence entre toutes les mises à jour et la mise à jour locale. La mise à jour locale est toujours incluse dans l'agrégation. La déviation standard des mises à jour est calculée puis toutes les mises à jour qui sont en dehors du seuil de tolérance sont rejetées. Le seuil de tolérance est un nombre de déviations standard qui peut être configuré par chaque nœud. Ainsi, un seuil de tolérance de trois inclura toutes les mises à jour dont la divergence est à moins de trois déviations standards de la mise à jour locale.

L'algorithme de sélection (Algorithme 4.2) permet de s'ajuster au scénario en étant plutôt dynamique. En effet, puisqu'il est basé sur la déviation standard, le filtre exclut seulement les mises à jour qui divergent beaucoup des autres, mais ne limite pas la mise à jour directement.

4.4 Stockage des modèles

Les nœuds enregistrent leur modèle dans un espace de stockage accessible par les autres entraîneurs via internet.

4.5 Communication

Les nœuds communiquent entre eux à l'aide de message JSON qui sont envoyés en pair à pair à tous les nœuds du réseau. Ainsi, plus il y a de nœuds dans le réseau, plus il y aura de messages à envoyer. Pour simplifier la gestion des communications, les nœuds sont tous connectés à un

Algorithme 4.2 Selection des mises à jour pour l'agrégation

Input : MAJ du noeud myUpdate, MAJ candidates updates, seuil de tolérance tolerance
Output : MAJ sélectionné

```
1  for i in range(updates.length) do
2  |   divergences[i] ← wd(myUpdate, updates[i]);
3  end for
4  med ← median(divergences);
5  maxDiv ← med + std(divergences) ∗ tolerance;
6  selIndex ← 0;
7  selected[selIndex] ← myUpdate;
8  for i in range(updates.length) do
9  |   if divergences[i] < maxDiv then
10 |   |   selIndex ← selIndex + 1;
11 |   |   selected[selIndex] ← updates[i];
12 |   end if
13 end for
14 return selected
```

nœud RabbitMQ et publient leur message sur le même canal. Cela imite une connexion pair à pair, mais simplifie l'implémentation.

Chaque nœud génère un identifiant aléatoirement et envoie un message de bienvenue sur le réseau pour informer les autres nœuds de sa présence. Son identifiant est inclus dans chaque message envoyé sur le réseau.

Les nœuds n'envoient pas directement leurs mises à jour. En effet, ils envoient plutôt un URI vers l'espace de stockage contenant la mise à jour. Ainsi, les autres pairs peuvent le télécharger au besoin, lorsqu'ils seront prêts plutôt que de potentiellement saturer la connexion avec de gros modèles.

Les nœuds ne communiquent pas non plus le modèle agrégé. En effet, puisque tous les nœuds ont accès à toutes les mises à jour, les autres nœuds ont seulement besoin de communiquer leur combinaison de mises à jour pour qu'ils puissent recalculer le modèle final. Ainsi, cela évite

d'avoir à télécharger un modèle de plus. De plus, cela évite qu'un nœud malhonnête ne fasse pas la bonne agrégation et cause un conflit avec d'autres entraîneurs honnêtes.

4.6 Génération deterministique d'identifiants des modèles

Puisque l'agrégation se déroule en parallèle sur tous les nœuds du réseau, ils doivent établir un consensus sur l'identification des groupes pour éviter d'ajouter des branches qui seraient des doublons. Par exemple, si deux nœuds publient leur même combinaison en simultanées, ils doivent être en mesure d'établir qu'il s'agit de la même combinaison. Pour ce faire, les nœuds déterminent un identifiant unique en faisant la concaténation de la liste ordonnée de mises à jour avec le modèle parent, puis utilisent la fonction de hachage SHA512 pour obtenir une chaîne de caractères standardisée et unique. Ainsi, les combinaisons auront le même identifiant partout à travers le réseau, car les opérations requises pour obtenir l'identifiant sont toutes déterministes. L'algorithme 4.3 montre en détail le processus de génération. Aucun mécanisme pour gérer les collisions entre identifiants n'a été implémenté.

Algorithme 4.3 Identifiants deterministiques des modèles

Input : ID du modèle parent parentModelId, ID des entraineurs des mises à jour sélectionnés learnerIds
Output : ID de la combinaison

1 $learnerIds \leftarrow sort(learnerIds).join()$;
2 $id \leftarrow parentClusterId + learnerIds$;
3 **return** $SHA512(id)$;

4.7 Jeu de données

Les jeux de données des entraîneurs sont divisés en trois parties : entraînement, validation et test. La partie d'entraînement est utilisée pour faire l'entraînement à chaque ronde. La partie validation permet de tester le modèle entraîné sur des données qui n'ont pas été vues pendant l'entraînement. Finalement, le jeu de données de test est utilisé pour l'évaluation empirique des branches lors de la phase de sélection. Ce jeu de données n'est jamais utilisé pendant

l'entraînement pour éviter qu'un entraîneur puisse manipuler le choix d'un autre entraîneur avec un modèle qui a fait du surapprentissage.

Il est important pour faciliter les tests d'avoir une méthode simple et efficace de varier la distribution des données. Pour ce faire, la distribution est faite de façon dynamique à l'aide de masque de sélection d'échantillons. En effet, les échantillons seront sélectionnés aléatoirement selon les probabilités contenues dans le masque final. Le masque final est un vecteur de la taille du jeu de données contenant la probabilité que l'échantillon au même index soit sélectionné.

Ainsi, les entraîneurs téléchargent l'entièreté du jeu de données puis appliquent les masques séquentiellement pour modifier la distribution. Les masques peuvent être combinés pour obtenir des distributions complexes. Cette approche est basée sur les fonctionnalités disponibles avec PyTorch à l'aide du RandomSampler.

Exemples de masques :

- **Balancé** : Balance un jeu de données par rapport aux classes.
- **Gaussienne** : Applique une distribution gaussienne aux classes en suivant une moyenne et une déviation standard.
- **Partition** : Limite les données à un sous-ensemble des données. Par exemple, si l'entraînement se déroule avec 10 entraîneurs, le masque pourrait être appliqué pour limiter chaque entraîneur à une seule partition.
- **Classe** : Limite le jeu de donnée aux classes spécifiées.

4.8 Technologies choisies

Le choix du cadriciel d'apprentissage machine pour le développement de la solution s'est arrêté sur PyTorch. En effet, les deux choix principaux étant PyTorch et TensorFlow, PyTorch a été choisi pour son excellente documentation et sa vaste communauté. De plus, beaucoup de projets dédiés à l'apprentissage fédéré utilisent PyTorch. Ainsi, il est plus simple d'apprendre d'exemples en étant familier avec le cadriciel.

PyTorch Lightning a été utilisé pour la création de modèles et ses fonctionnalités avancées d'entraînement. Il s'agit d'une librairie facilitant le développement d'IA avec PyTorch.

Afin de faire le suivi des expériences, MLFlow a été intégré au logiciel à l'aide de la librairie disponible sur PIP et d'une instance locale. Ce logiciel gratuit et open source permet d'enregistrer et extraire les données des entraînements pour faire analyser les entraînements. Il est très réputé auprès de la communauté MLOps.

Finalement, afin de faciliter le déploiement et l'orchestration de plusieurs entraîneurs en simultané, les nœuds ont été mis dans des conteneurs Docker. Les conteneurs permettent de déployer des logiciels dans un environnement isolé et virtualisé. De plus, à l'aide de Docker Compose, il est possible d'orchestrer de nombreux conteneurs en même temps.

4.9 Conclusion

L'implémentation de la solution a été faite dans l'optique d'accélérer la convergence des modèles en réutilisant les meilleurs modèles et en créant des agrégations les moins divergentes possibles. L'implémentation respecte le cadre flexible de l'architecture et est ouverte à la modification. Les modules sont organisés de façon à être indépendants et s'assurent qu'il n'y ait pas de conflits de dépendances. Finalement, les technologies choisies complémentent la solution et permettent de livrer une solution fiable, mesurable et déployable dans tous types d'environnement.

CHAPITRE 5

ÉVALUATION

Ce chapitre cherche à évaluer la solution dans plusieurs contextes pour déterminer quels sont les avantages et désavantages de l'approche. Ainsi, plusieurs expériences sont conduites pour obtenir des résultats provenant de différentes configurations expérimentales.

5.1 Configuration expérimentale

Tous les nœuds du réseau sont isolés dans des conteneurs orchestrés avec Docker Compose. Ils ont été déployés sur le même serveur Ubuntu 22.04. Le processeur du serveur est un Intel Xeon Gold 5118 possédant 48 cœurs et 126GB de RAM.

Jeu de données : Les tests ont été faits sur deux jeux de données pour la vision par ordinateur, soit MNIST et FashionMNIST. Le premier consiste en des images de chiffre écrit à la main. Le second est des images de pièces de vêtements en noir et blanc. Bien qu'ils s'agissent de jeu de données balancé, ils seront distribués de façon non uniforme lors des différentes expériences. Il s'agit d'une méthodologie qui a été utilisée dans plusieurs papiers tels que (Li *et al.*, 2020) et (Wang, Liu, Liang & Joshi). Il s'agit de jeux de données simples qui permettent de valider des idées ainsi que de les comparer facilement à d'autres approches.

Modèles : Deux modèles sont utilisés pour la tâche de classification : l'architecture LeNet provenant de (LeCun *et al.*, 1989) et une version plus légère du modèle LeNet dans lequel les canaux des couches convolutionnelles ont été réduit de six à trois et 16 à neuf. Afin de démontrer que le système peut aussi fonctionner pour des problèmes non supervisés, des autoencodeurs seront entraînés. Il s'agit d'un type de réseau de neurones qui essaie de reproduire ses entrées en sorties en encodant les caractéristiques des données en entrées dans un espace latent de plus petite dimension. L'architecture des autoencodeurs réutilise les deux premières couches du modèle LeNet comme encodeur et miroir les couches pour le décodeur. Les résultats d'entraînements sont satisfaisants après 10 époques d'entraînement local comme cela peut être vu dans le tableau **??**.

Tableau 5.1 Performance des réseaux de neuronnes entraînés localement avec tous les échantillons

	MNIST		FMNIST	
Model	Accuracy	Loss	Accuracy	Loss
LeNet	97%	0.089	83%	0.46
LeNet Light	96%	0.16	77%	0.57
LeNet CAE	-	0.006	-	0.015

La solution de référence : Les différents modules de la solution ont été réutilisés dans l'implémentation de référence qui utilise FedAvg comme algorithme d'agrégation. Il s'agit donc d'une version décentralisée de FedAvg dans laquelle tous les nœuds font l'entraînement et l'agrégation.

5.2 Expérience avec données uniformes

Les premiers résultats dans le tableau **??** démontrent l'utilité des filtres pour limiter la divergence de poids lors de l'agrégation. En effet, l'expérience conduite distribue le jeu de données de façon uniforme entre les entraîneurs. Les résultats démontrent que la limitation de la divergence de poids rend les agrégations plus prédictibles. En effet, l'écart entre le meilleur et le pire entraîneur sont moins grand avec MCFL que FedAVG.

De plus, MCFL produit des modèles supérieur à FedAVG dès la première ronde, cependant l'écart se réduit et devient négligeable après la cinquième ronde. Cela peut être observé dans la figure 5.1.

Tableau 5.2 Comparaison des performances d'un réseau FedAvg et MCFL

Algorithm	Minimum	Average	Maximum
FedAvg	96.2%	98.2%	99.5%
MCFL	98.1%	99.2%	99.8%

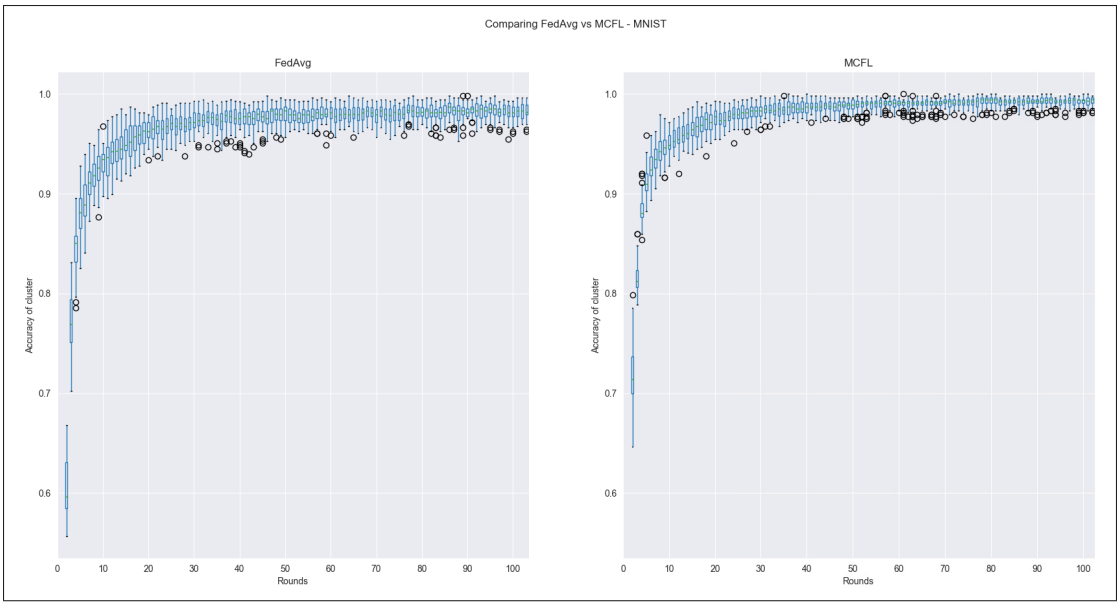

Figure 5.1 Comparaison de l'évolution de la précision des entraîneurs d'un réseau utilisant FedAvg et MCFL

5.3 Expériences avec données non uniformes

5.3.1 Classification avec sous-ensemble uniformes

Cette expérience cherche à démontrer la capacité du réseau de converger rapidement dans un scénario difficile. Pour ce faire, les données sont divisées selon les classes en trois groupes et chacun des entraîneurs est assigné à un des trois groupes de données. Puis les données sont réparties également entre les entraîneurs du même groupe. Ainsi, cela forme trois sous-groupes avec des données uniformes, mais le jeu de données est réparti de façon non uniforme à travers le réseau. Un exemple de distribution de données par groupe est visible dans le graphique 5.2.

La figure 5.3 illustre d'excellents résultats très tôt dans l'entraînement puisque chaque nœud commence dans sa propre branche. De plus, les nœuds convergent rapidement vers leur groupe, ce qui réduit le nombre de branches et augmente la capacité du modèle à généraliser. Il est possible d'observer ce comportement dans la figure 5.4 qui illustre le nombre de branche lors de

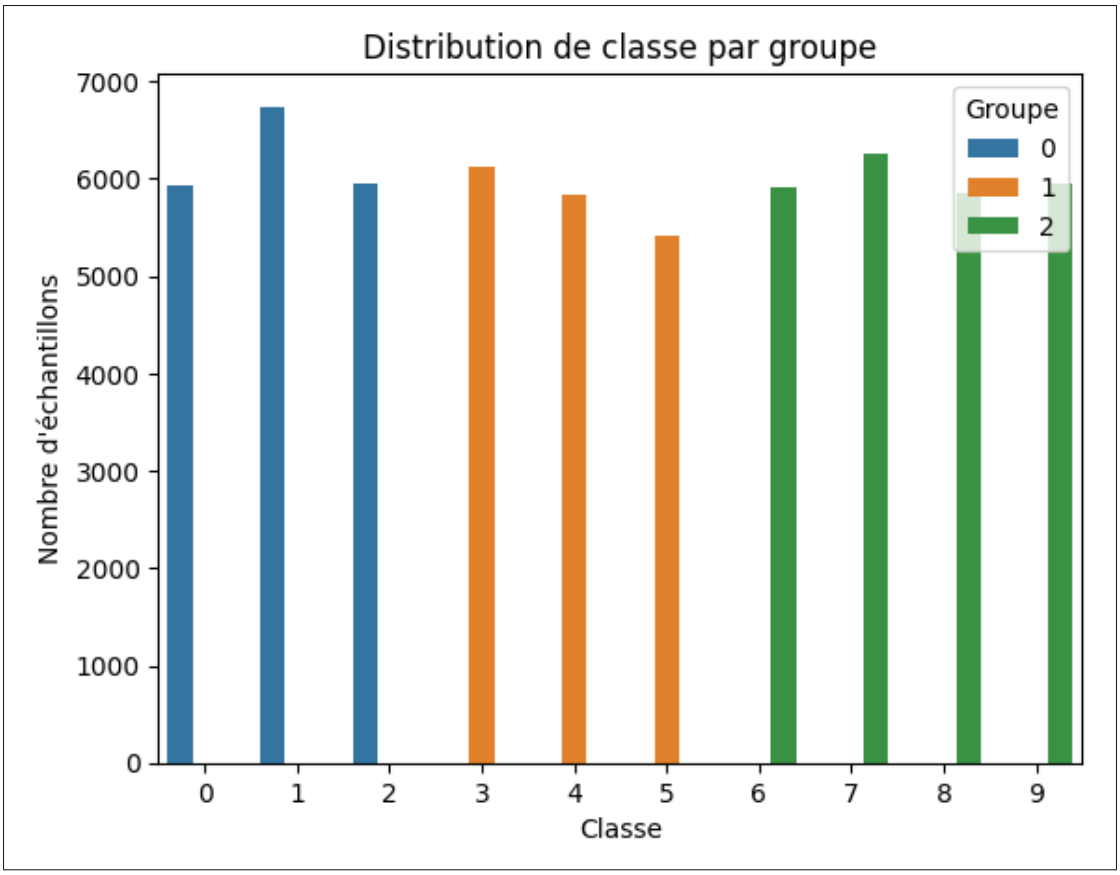

Figure 5.2 Distributions des classes dans les trois groupes de données

l'entraînement. Quant au réseau utilisant FedAvg, il a beaucoup de difficulté à converger puisque même après 100 rondes le pire entraîneur performe à seulement 82%.

Cette expérience est particulièrement intéressante puisqu'elle est très similaire à un scénario avec des acteurs tentant d'empoisonner un modèle global de manière subtile. En effet, tous les acteurs sont honnêtes, mais leur distribution de données est trop différente, car ils n'ont pas d'intersection. Donc les entraîneurs ne peuvent pas savoir s'il s'agit d'entraîneurs honnêtes ou non grâce à leur données. Ainsi, il est légitime pour un groupe de considérer les autres groupes comme des acteurs malveillants et en s'isolant dans différentes branches, les entraîneurs réussissent à protéger leur modèle pour converger rapidement. Bien que cette expérience démontre que le système peut se défendre à une attaque de base, cela tombe hors du champs de la recherche.

Ainsi, c'est seulement à titre informatif que la solution pourrait être appliquée pour la résistance aux acteurs malveillants.

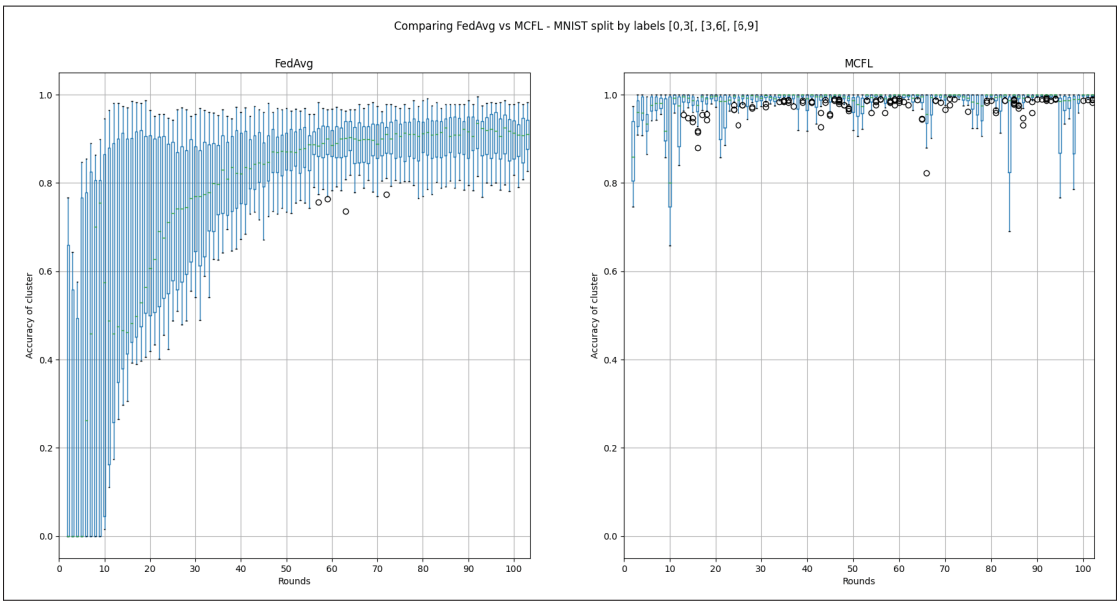

Figure 5.3 Comparaison de l'évolution de la précision des entraîneurs dans un réseau avec une distribution de données en sous-groupe

Il est aussi pertinent de comparer les résultats en fonction du temps écoulé, car les rondes de MCFL sont beaucoup plus longues puisque plusieurs modèles sont entraînés. En effet, puisque nos règles de sélections de modèles stipulent que les nœuds doivent entraîner $\sqrt{n}$ modèles, les rondes seront $\sqrt{n}$ fois plus longue qu'une ronde normale de FedAvg. Par exemple, dans une ronde où 15 groupes sont disponibles, l'entraîneur devra en entraîner trois ou quatre modèles. De plus, à cela doit aussi être ajouté le temps requis pour sélectionner le modèle. En effet, puisque les filtres de modèles se basent sur des données empiriques de performances, l'entraîneur doit évaluer tous les modèles, ce qui prend du temps et augmente linéairement avec le nombre de modèles disponibles. Cependant, il s'agit d'une opération plus rapide que l'entraînement sans oublier qu'elle peut être raccourcie en réduisant le jeu de données de test. Toutefois, les gains de performances non négligeables permettent tout de même à MCFL d'obtenir des résultats supérieurs lorsqu'on considère le temps passé depuis le début de l'entraînement comme il est possible de le voir dans la figure 5.5.

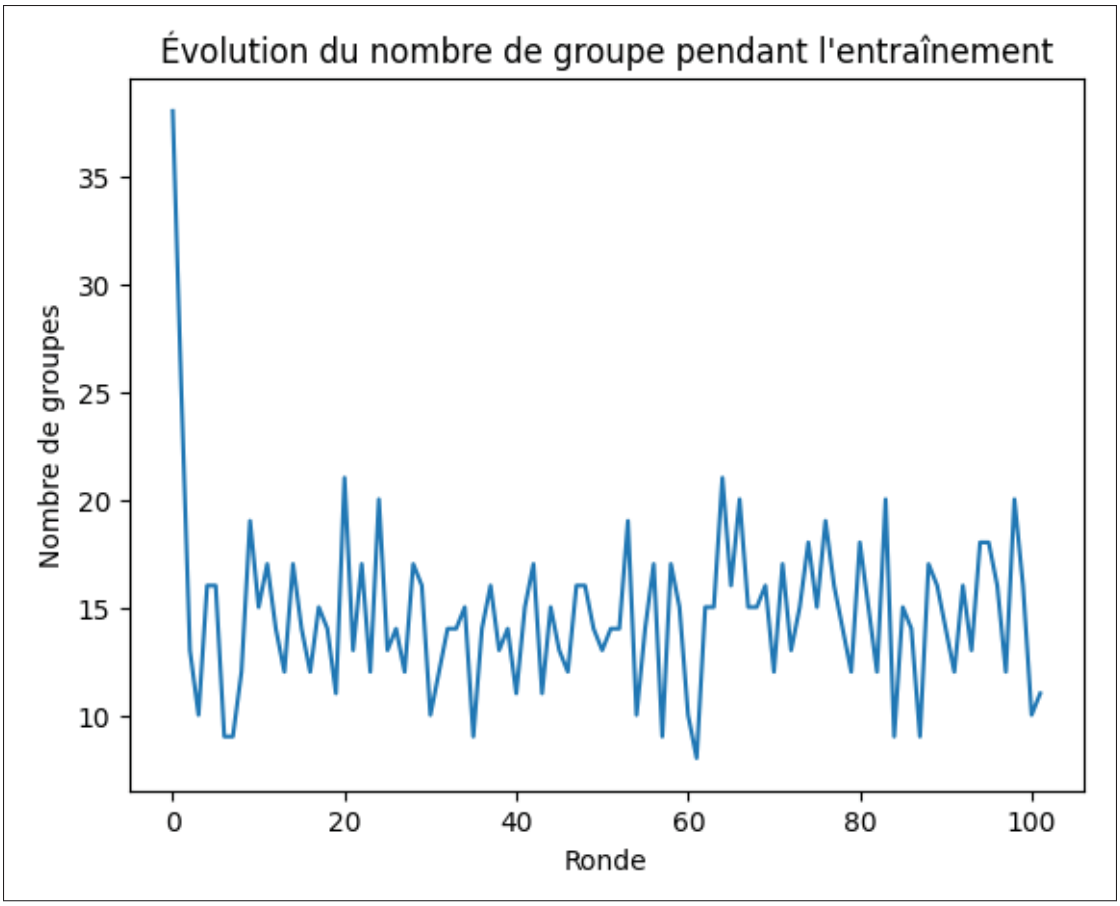

Figure 5.4 Évolution du nombre de groupes dans le réseau MCFL

Une pratique courante lors de l'entraînement fédéré est celle de peaufiner le modèle global avec un apprentissage local. Ainsi, les performances après la période de peaufinage ont aussi été évaluées pour créer le graphique 5.6. Dans ce dernier, il est possible de voir que FedAvg performe beaucoup mieux que précédemment. Cependant, MCFL domine toujours les performances surtout en début d'entraînement.

Les gains de performances du peaufinage pour MCFL sont assez négligeables et permettent surtout d'améliorer les pires cas. Ainsi, le peaufinage n'est probablement pas nécessaire avec MCFL puisque l'approche par branche crée des modèles déjà spécialisés. Finalement, il est possible d'expliquer la grosse différence du FedAvg entre le modèle global et le modèle peaufiné par la divergence des poids lors de l'agrégation qui a pour résultats de faire un nouveau modèle

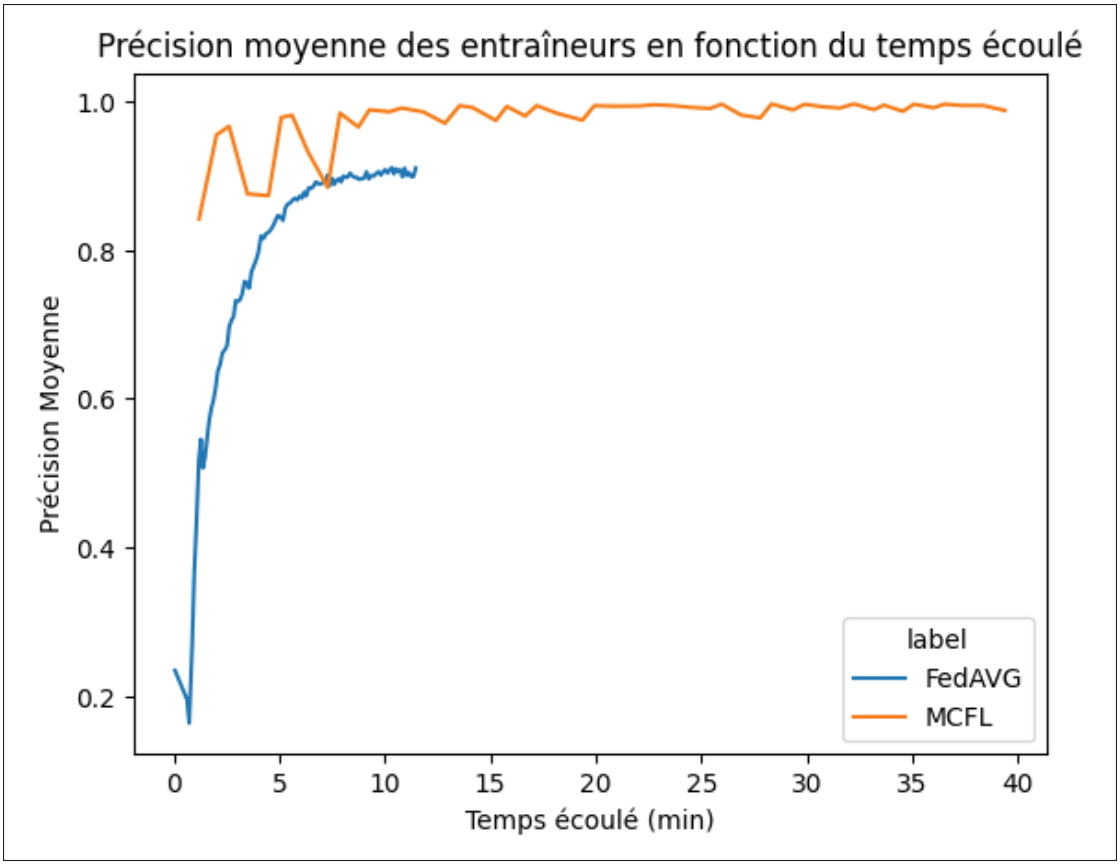

Figure 5.5 Comparaison de l'évolution de la précision en fonction du temps écoulé depuis le début de l'entraînement dans un réseau avec une distribution de données en sous-groupe

mal optimisé, mais qui peut être corrigé relativement facilement, puisque tous les entraîneurs ont des tâches connexes donc requièrent les mêmes connaissances de base.

5.3.2 Classification avec distribution normale par entraîneur

Dans ces expériences, les données ont été distribuées selon une distribution gaussienne. Les entraîneurs sont donc affectés à une classe moyenne et auront un jeu de données non balancé selon une distribution normale. Le graphique 5.7 est un exemple de distribution pour un entraîneur dont la classe moyenne est quatre avec une déviation standard de un.

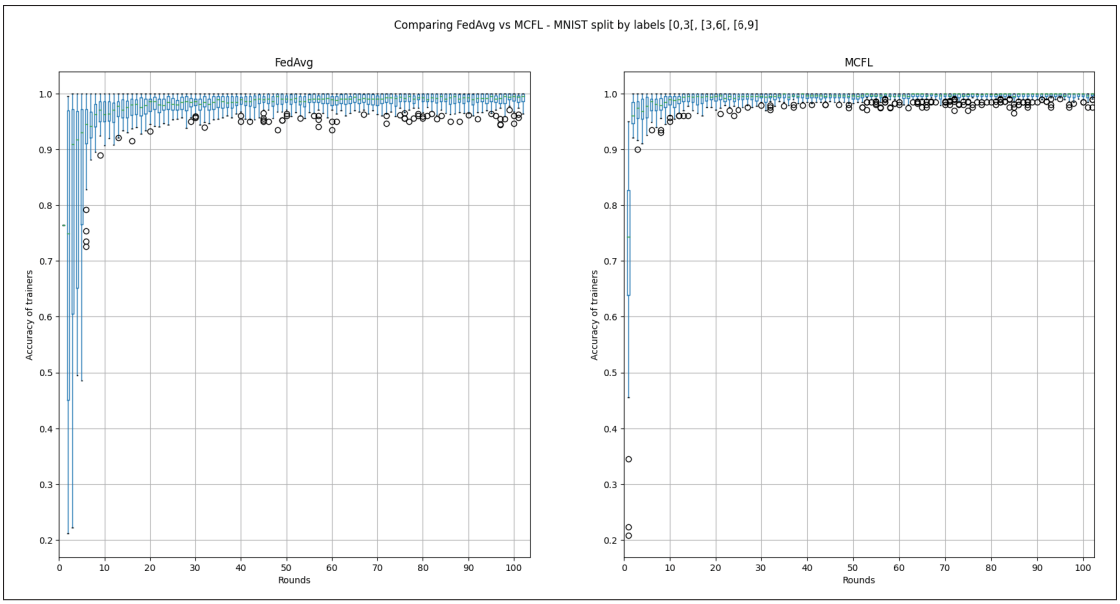

Figure 5.6 Comparaison de l'évolution de la précision après le peaufinage des entraîneurs dans un réseau avec une distribution de données en sous-groupe

Cette distribution ne crée pas de groupes bien distincts, car les entraîneurs ont tous des classes en communs. Trois expériences ont été menées pour tester différentes configurations de tolérance. Le graphique 5.8 illustre la performance moyenne des entraîneurs pour chacun des réseaux. Ce scénario est plus facile pour FedAvg et plus difficile pour MCFL, mais ce dernier obtient encore des performances supérieures. En effet, MCFL performe mieux en début d'entraînement et devient marginalement meilleur plus l'entraînement progresse. Entre les différentes configurations de MCFL, la combinaison de différentes tolérances semble la plus performante. Cela peut être expliqué par le plus grand nombre de branches, car certains nœuds utilisent une déviation standard de un. Il est donc peu probable que leur combinaison soit la même qu'un entraîneur avec un seuil de deux ou trois déviations standard.

Le graphique à la figure 5.9 illustrant l'évolution du nombre de groupes confirme qu'il y avait plus de groupes lors de l'expérience avec la configuration de seuil mixte. Il est aussi intéressant de voir que le réseau converge vers une seule branche avec la configuration de trois déviations standards.

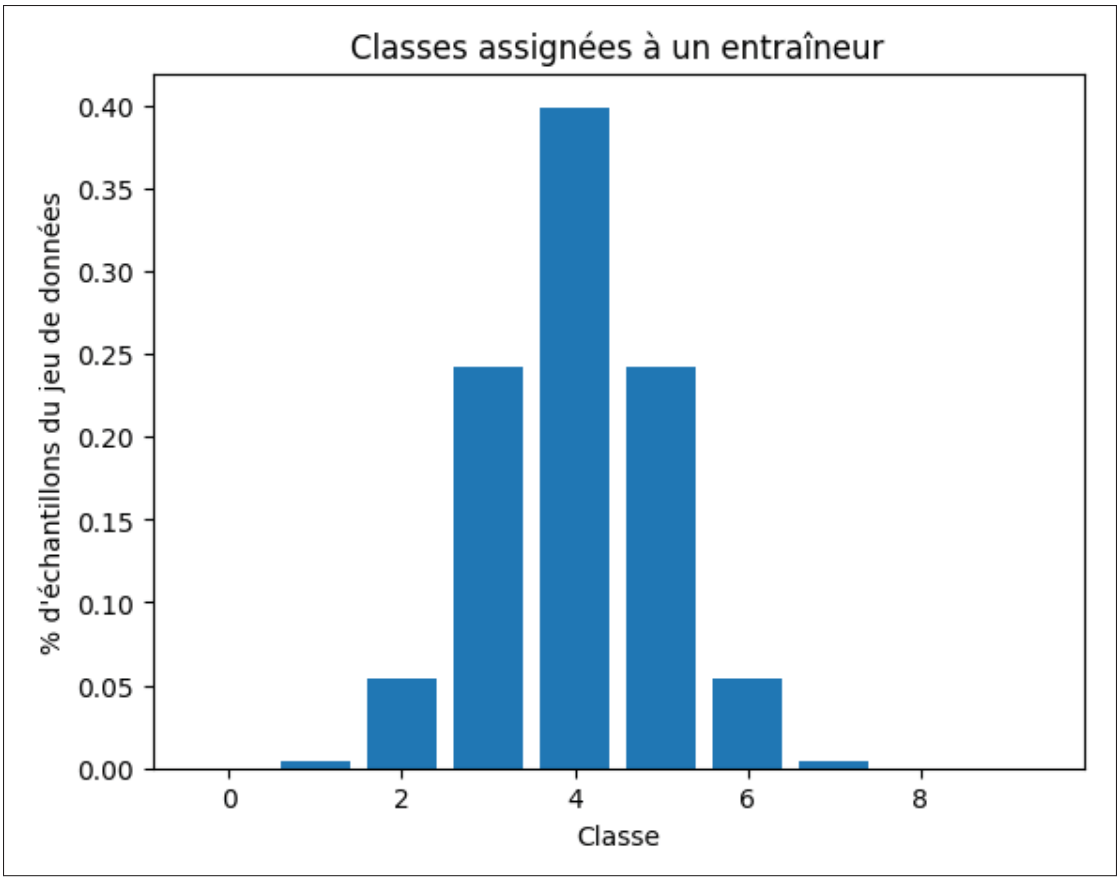

Figure 5.7 Exemple de distribution de données d'un entraîneur

Puisque nous considérons que les entraîneurs sont indépendants et autonomes, le modèle mixte est un scénario plus réaliste, car on ne souhaite pas imposer un seuil aux entraîneurs. De plus, il est possible que certains entraîneurs décident d'utiliser différentes métriques, ce qui pourrait créer plus de branches comme dans le scénario mixte.

Le nombre de branches a un impact sur le temps d'entraînement puisque les règles de sélection choisissent plusieurs modèles à entraîner. Ainsi, plus le nombre de branches est grand, plus il y a de modèles à entraîner chaque ronde. Le graphique 5.10 illustre le temps requis pour chaque ronde de chaque configuration. Seulement la configuration avec un seuil de tolérance de 3 est compétitive avec la solution de référence, car elle est environ 3 fois plus lente, sa convergence est donc significativement plus rapide.

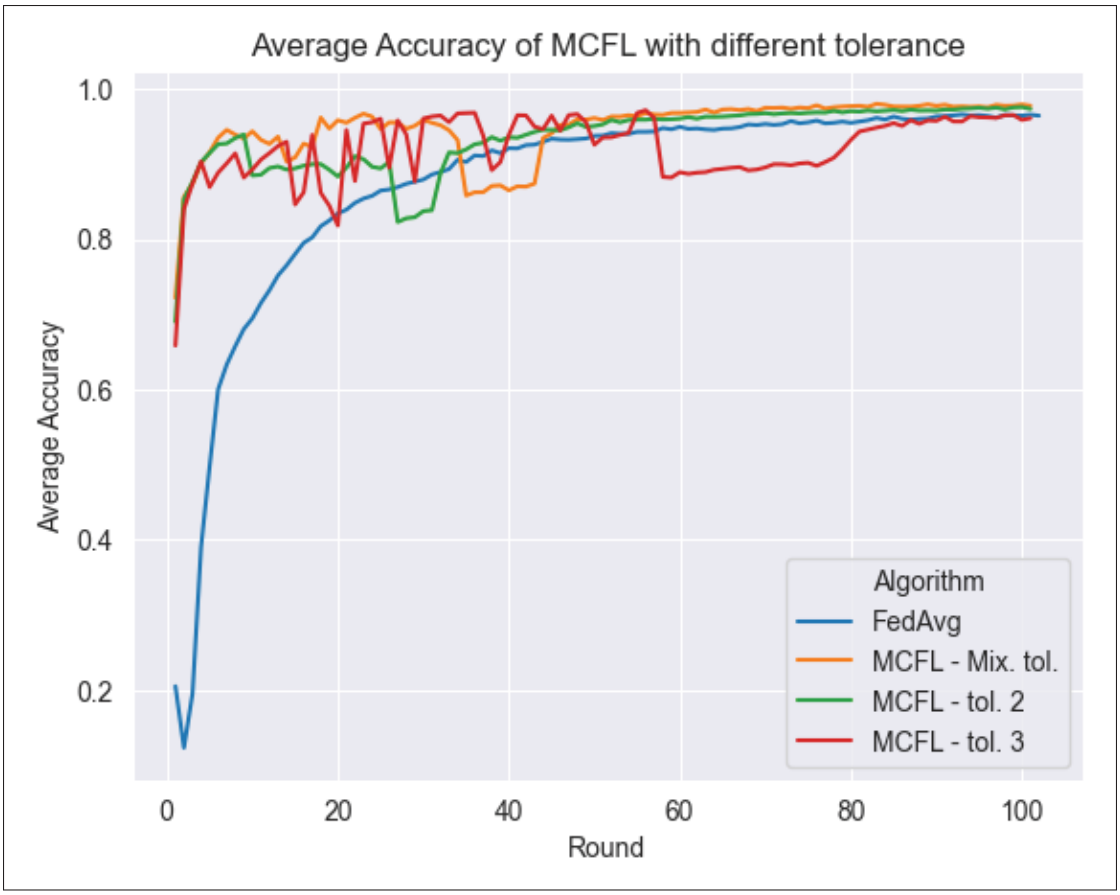

Figure 5.8 Comparaison de l'évolution de la précision des entraîneurs dans un réseau avec une distribution de données normales

5.3.3 Classification avec jeux de données distincts

Cette configuration cherche à tester la capacité du réseau à créer des branches spécialisées dans les cas où la fusion des branches réduirait les performances. Pour se faire, le modèle léger a été choisi pour que le modèle soit plus limité dans sa capacité d'apprentissage. De plus, deux jeux de données sont utilisés en simultané dans le réseau. En effet, la moitié des entraîneurs se partagerons MNIST et l'autre moitié FMNIST. Il y a ainsi plus de caractéristiques à apprendre dans un modèle limité.

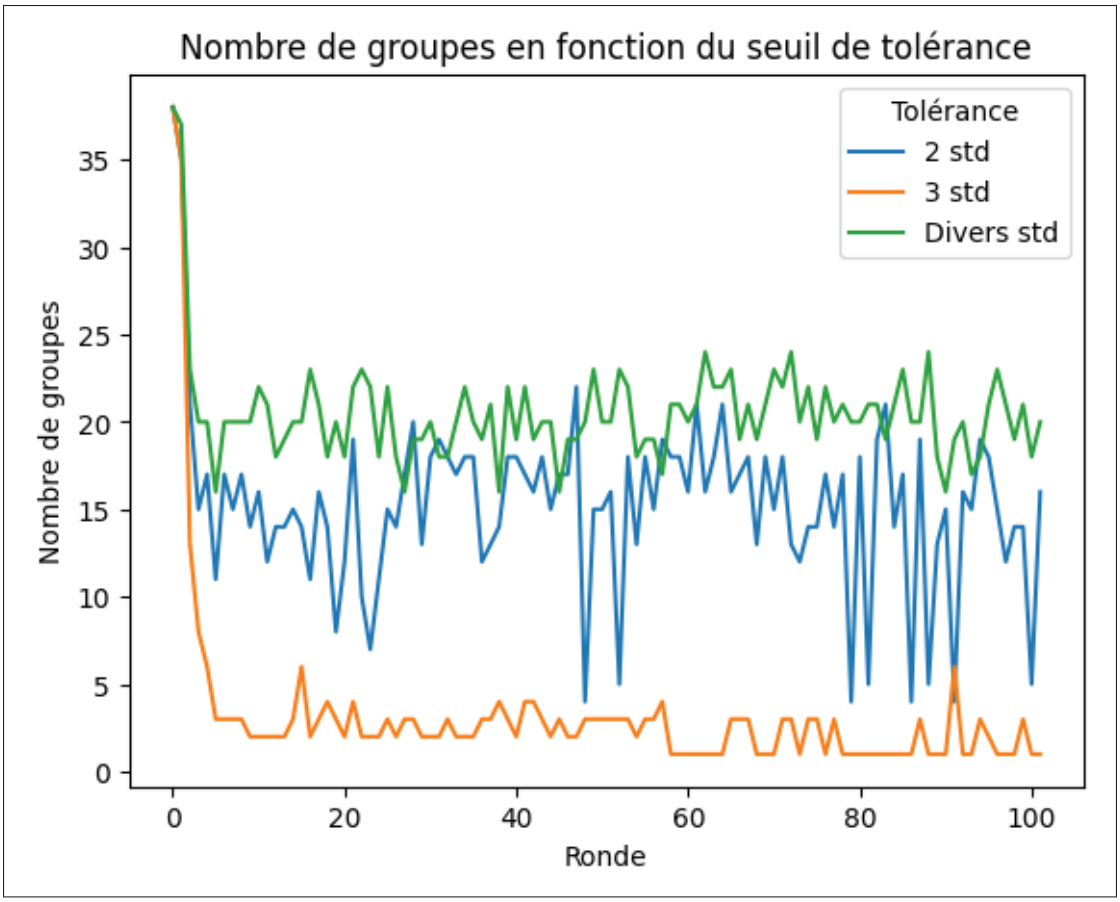

Figure 5.9 Évolution du nombre de groupes pour différents niveau de tolérance

Les résultats illustrés dans la figure 5.11 et 5.12 sont intéressants. Les résultats sont séparés pour chaque jeu de données puisque les deux tâches ont une précision différente. Cependant, il est possible d'observer que les deux groupes ont une meilleure performance avec MCFL.

Il est particulièrement intéressant dans la figure 5.11 de remarquer qu'il y a beaucoup d'entraîneurs ayant le jeu de données MNIST qui choisissent un modèle entraîné avec principalement des entraîneurs FMNIST à cause de la règle de popularité qui les influences à le faire. Ce n'est pourtant pas réciproque au niveau de FMNIST. En effet, les entraîneurs FMNIST sélectionnent seulement des modèles entraînés par des entraîneurs avec FMNIST. Ceci implique donc que le transfert d'apprentissage se fait seulement de FMNIST à MNIST. Toutefois, les entraîneurs gardent toujours une branche spécialisée pour MNIST qui atteint une excellente performance.

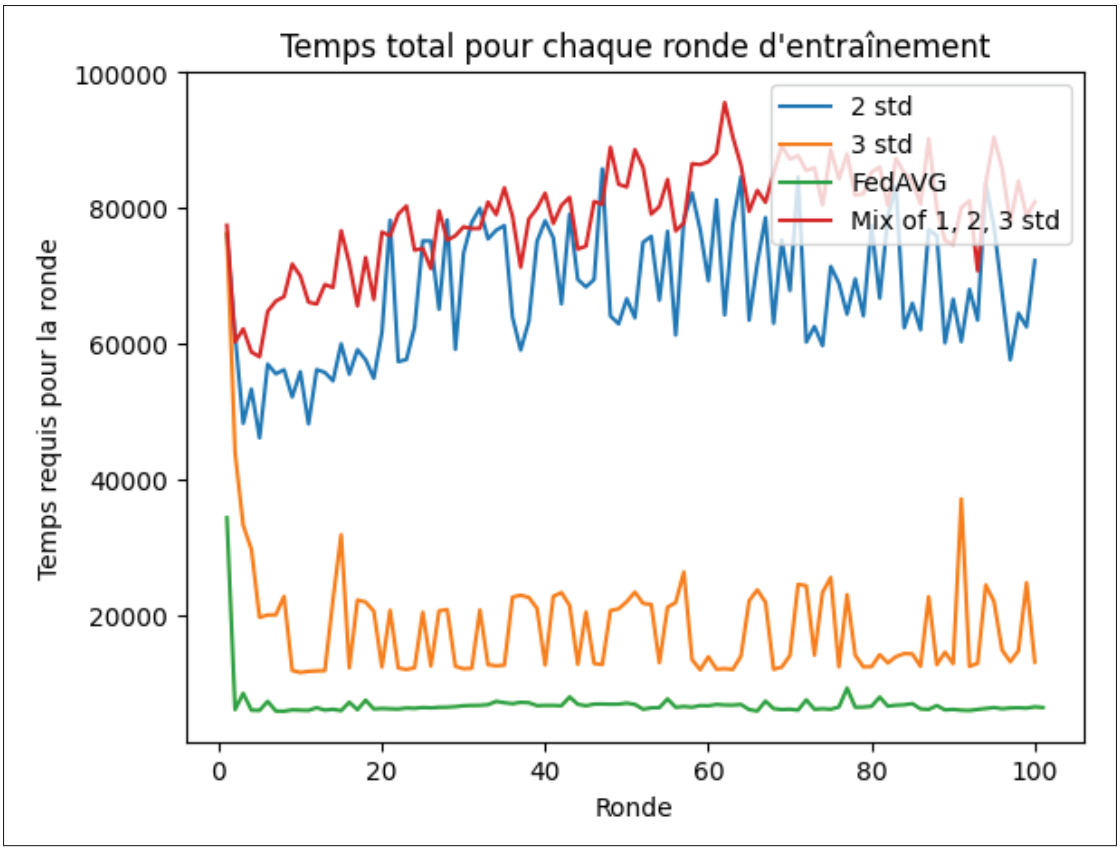

Figure 5.10 Temps requis par ronde pour différents niveaux de tolérance

Finalement, le nombre de branches n'est pas problématique, puisque le réseau se stabilise entre 10 et 20 branches. Les entraîneurs doivent donc entraîner entre trois et quatres modèles environ.

5.3.4 Autoencodeur avec jeu de données distincts

Afin de démontrer la capacité de l'architecture à s'adapter à divers modèles, la prochaine expérience entraîne un autoencoder (CAE Light) sur un réseau assez spécial. La figure 5.13 compare les performances des entraîneurs des groupes spécialisés et d'un modèle global. Les modèles spécialisés performent mieux que le modèle global par 24% pour le modèle MNIST et 17.5% pour le modèle FMNIST.

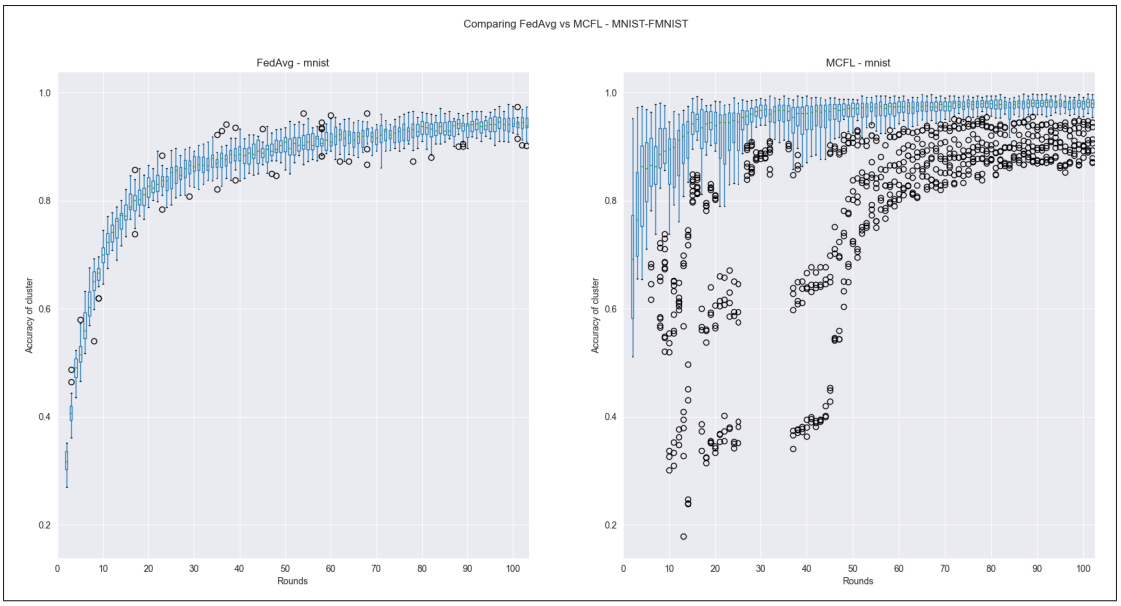

Figure 5.11 Évolution de la précision pour les entraîneurs avec MNIST comme jeu de données dans un réseau non uniformes

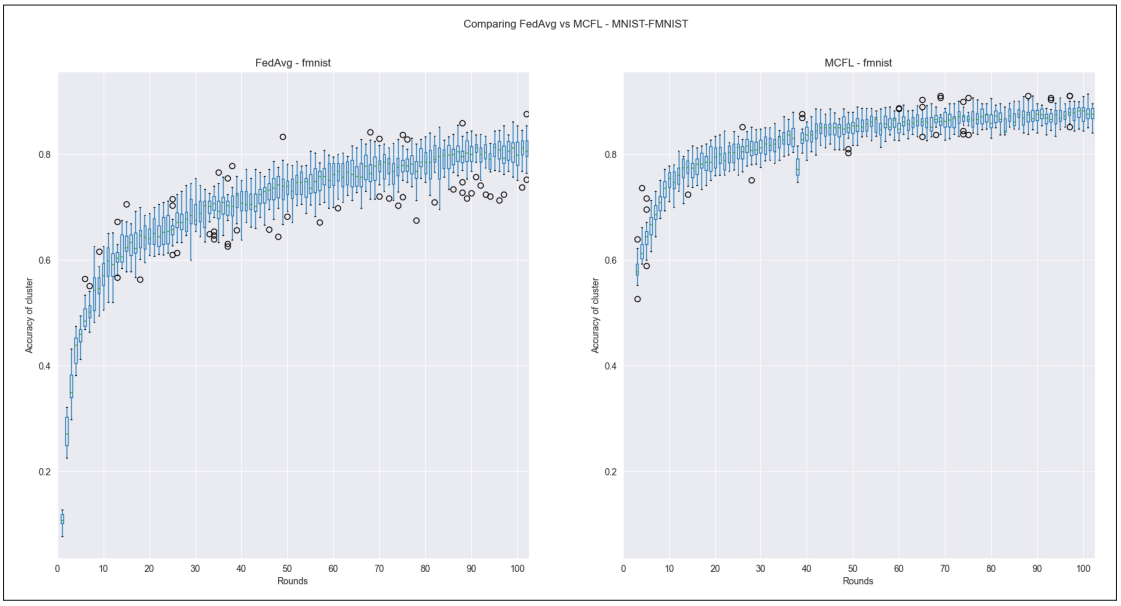

Figure 5.12 Évolution de la précision pour les entraîneurs avec FMNIST comme jeu de données dans un réseau non uniformes

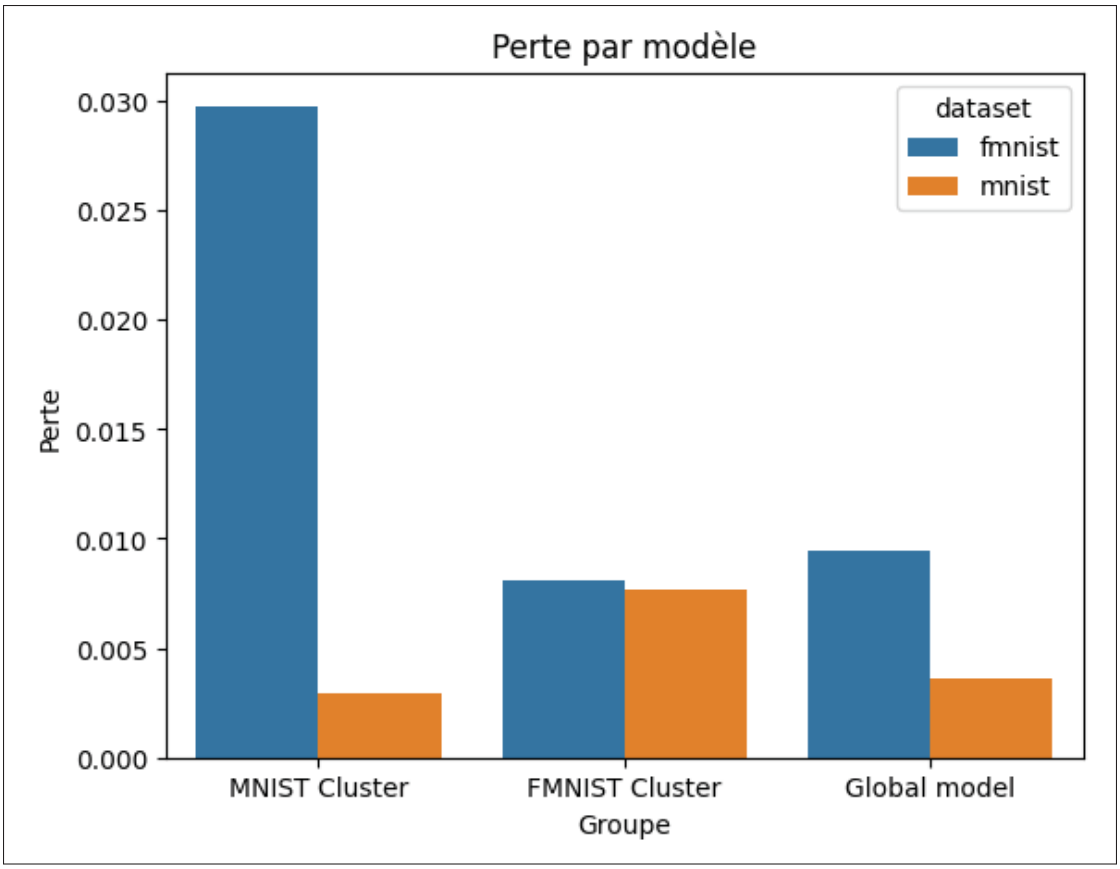

Figure 5.13 Comparaison des métriques des autoencodeurs spécialisés et global d'un réseau avec des entraîneurs MNIST et FMNIST

5.4 Composition de modèles

Comme mentionné précédemment, lorsque le réseau ne converge pas vers une seule branche, les différentes branches auront une spécialité. Cependant, en utilisant plusieurs modèles spécialisés, il est possible d'obtenir un modèle généralisé à l'aide d'un petit jeu de données balancé.

5.4.1 Modèle spécialisé

Pour démontrer cette capacité, trois modèles sont extraits du graphe provenant de l'expérience des classificateurs avec trois sous-groupes uniformes. Les modèles ont été sélectionnés en analysant le graphe pour déterminer les branches principales qui ne partagent pas ou peu d'entraîneurs entre eux afin d'obtenir des branches avec différentes spécialités.

Par la suite, plusieurs modèles ont été entraînés sur un jeu de données balancé pour tester différents algorithmes. L'objectif est d'obtenir un modèle généralisé en utilisant le moins de données possible. Tous les algorithmes ont été entraînés avec et sans PCA et avec différentes quantités de données.

L'algorithme de référence pour évaluer si l'entraînement est utile consiste simplement à utiliser la prédiction la plus confiante des trois classeurs. Cet algorithme obtient une précision de 55%, ce qui est meilleur qu'une décision aléatoire, mais n'est pas acceptable, car elle est nettement inférieure aux résultats obtenus avec FedAvg.

La figure 5.14 démontre les résultats obtenus. Les résultats sont plutôt bons, surtout considérant la quantité de données requises pour entraîner le modèle. En effet, lors de l'apprentissage local, les entraîneurs vont tenter de rebalancer leur jeu de données en retirant des échantillons des classes majoritaires et en ajoutant des doublons de classes minoritaires. Ainsi, moins ils auront besoins d'échantillons de classes minoritaires, moins le modèle aura tendance à suraprendre les classes minoritaires.

Les performances deviennent comparables à la moyenne des entraîneurs en FederatedLearning avec FedAvg. MCFL reste cependant plus polyvalente avec l'approche de composition de nouveaux modèles à partir de modèles spécialisés.

5.4.2 Correction de biais

Il est aussi intéressant de s'intéresser à la correction de biais qui peut être accomplie de la même façon. En effet, en adoptant la même stratégie, à l'aide d'un jeu de donnée étiqueté différemment, le nouveau classificateur peut apprendre à changer les étiquettes. Pour cette expérience, les classificateurs MNIST et FMNIST sont extrait du réseau précédent contenant des entraîneurs avec MNIST et FMNIST.

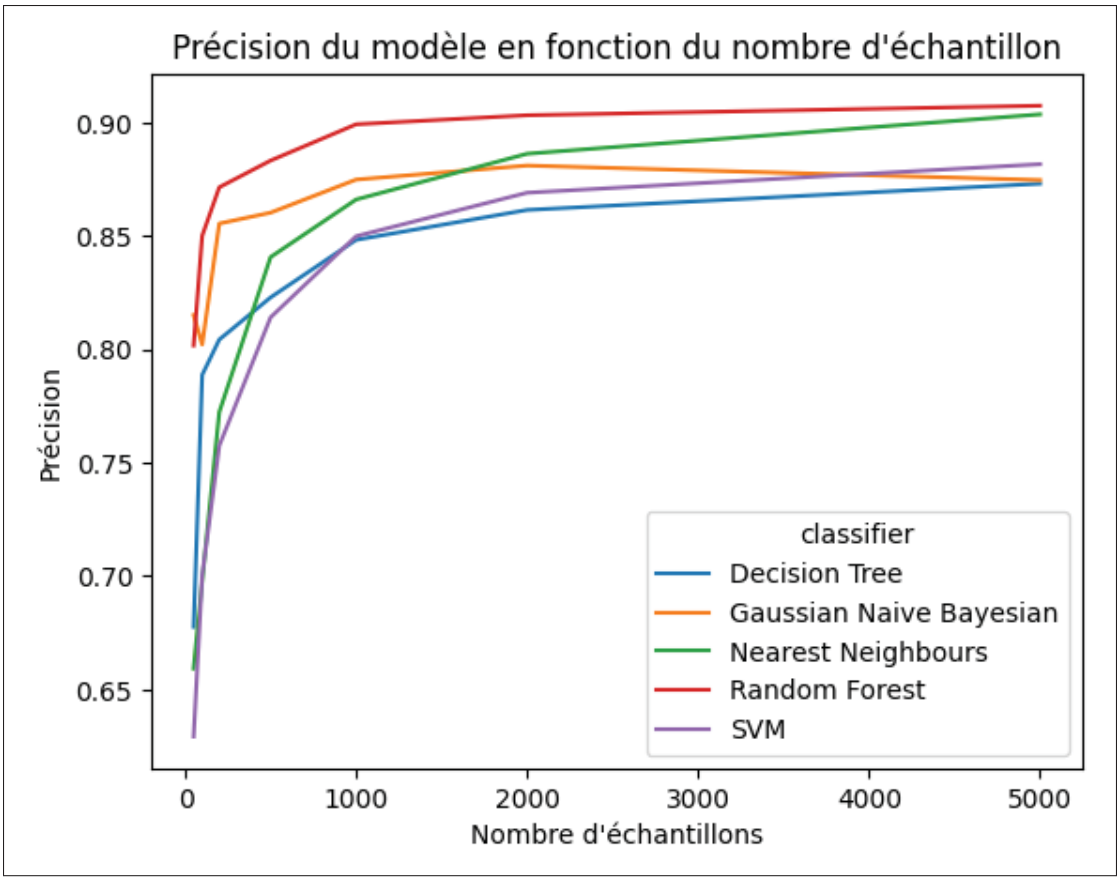

Figure 5.14 Classification de MNIST en utilisant trois classificateurs spécialisés

Puis, un nouveau jeu de données est créé à partir de MNIST et FMNIST, cependant les classes de FMNIST sont modifiées pour être étiquetées de 10 à 19 plutôt que zéro à neuf. Le jeu de données final a donc 20 classes et est composé de chiffres et de pièces de vêtements.

La même méthodologie est utilisée pour l'entraînement du classificateur final. La figure 5.15 illustre les résultats obtenus. Le classificateur apprend à classifier les données très rapidement avec seulement 50 données. La forêt aléatoire est aussi le modèle le plus performant dans ce scénario.

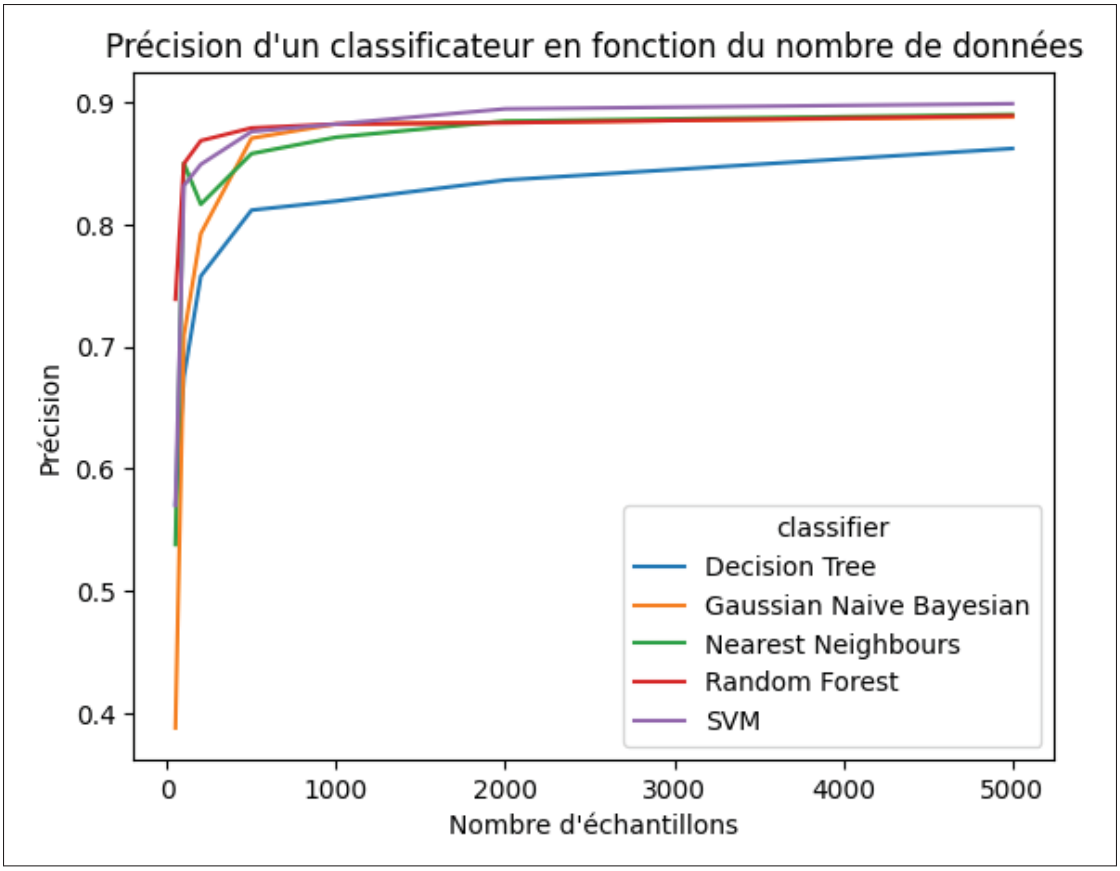

Figure 5.15 Classification de MNIST et FMNIST

5.5 Conclusion

Les expériences conduites ont permis de découvrir les forces et les faiblesses de la solution proposée. Sa plus grande faiblesse est son long temps d'exécutions. En effet, comme il a été démontré dans les différentes expériences, l'entraînement est plus long, car les entraîneurs filtrent les modèles disponibles en faisant de l'inférence en plus d'avoir plusieurs modèles à entraîner par ronde. Cependant, les performances des modèles atteignent leur maximum bien plus rapidement. Il est difficile de faire une comparaison entre les résultats obtenus et FedAvg, cependant, voici les grandes lignes :

- Avec MCFL le temps d'une ronde augmente avec le nombre de modèles.
- MCFL peut converger vers un modèle ou avoir plusieurs modèles spécialisés.
- MCFL obtient des modèles performants dès les premières rondes.

- En regardant le graphique de la précision en fonction du temps écoulé, on peut déterminer que MCFL est plus performant que FedAvg tant qu'il n'y a pas trop de branches.

CONCLUSION

L'importance de développer des techniques d'apprentissages respectueuses de la vie privée et de la souverainetés des entitées participantes a motivé cette recherche. En identifiants certains problèmes majeurs comme la distribution non uniforme des données à travers une revue de la litérature et une analyse d'impact des paramètres, ce mémoire a été en mesure de suggérer une architecture générique pour réduire l'impact de certains aspects négatifs de l'apprentissage décentralisé afin de rendre ces techniques plus attrayantes du point de vue des entraîneurs.

Les suggestions majeures de cette architecture cherchent à donner plus de contrôles aux entraîneurs contrairement aux approches traditionnelles dans laquelle des acteurs externes valident les mises à jour. En effet, certains scénarios rendent la prise de décision très difficile par un acteur externe qui ne peut voir les données sur lesquelles les mises à jour ont été entraîné. Alors que les entraîneurs peuvent prendre des décisions plus éclairées basées sur leur données.

Pour ce faire, la solution proposée est décentralisée et permet l'entraînement de plusieurs modèles en simultanés permettant une évolution du modèle non-linéaire. De plus, l'ajout de règles définies par l'entraîneur, ont permis de mitiger l'impact de certains de ces problèmes. Cette approche a été démontré comme étant flexible et fonctionnelle avec une implémentation visant la rapidité de convergence du modèle dans quelques scénarios.

Cependant, les évaluations complétés sont plutôt limités considérants l'explosion combinatoires de tous les paramètres qui peuvent avoir un impact dans ce type de système. En effet, il serait pertinent dans le futur d'évaluer plus de modèles, de jeux de données, de filtres de sélections et de distribution de données ainsi que divers scénarios avec des acteurs malveillants.

L'élaboration de ce système a cependant permis de développer une expertise et d'identifier de nouveaux points d'améliorations. En effet, il existe plusieurs autres améliorations qui pourront être fait de façon itérative comme l'entraînement asynchrone, l'intégration à une chaîne de bloc

et la distillation de connaissance de modèles spécialisés à un modèle généralisé au cours de travaux futurs.

Finalement, tout le code pour cette recherche est accessible à partir de Github (https://github.com/Mick00/dml).

RECOMMANDATIONS

L'expérience acquise en développant la solution a permis de trouver plusieurs points à améliorer qui pourait grandement bénéficier la solution. Premièrement, rendre l'entraînement asynchrone. Deuxièmenent, ajouter une chaîne de bloc. Troisièmement, choisir des règles plus rapides.

7.1 Entraînement asynchrone

L'implémentation actuelle est synchrone puisque les entraîneurs attendent que tous les autres entraîneurs aient publié leurs mises à jour avant de passer à l'agrégation. Il y a cependant certains désavantages à cette approche. En effet, il est possible que certains entraîneurs subissent des pannes qui les empêchent de participer à l'entraînement. Dans ce cas, le réseau sera bloqué jusqu'à ce que l'entraîneur revienne en ligne pour soumettre sa mise à jour.

Une alternative assez simple pour implémenter une version asynchrone serait de fonctionner par délai d'attente. C'est-à-dire que les entraîneurs proposeraient leurs combinaisons de mises à jour après avoir attendu un certain temps après la dernière mise à jour. Ainsi, l'entraînement ne sera jamais bloqué par un entraîneur hors-ligne, puisque les entraîneurs feront l'agrégation sans avoir besoin de toutes les mises à jour.

Une implémentation asynchrone rendrait la solution plus apte à gérer l'apprentissage dans un contexte entre appareils plutôt qu'en silo. En effet, des appareils comme des téléphones ne seront pas toujours en ligne puisqu'ils peuvent manquer de pile ou de connexion internet, ce qui freinerait l'apprentissage dans l'implémentation actuelle. Cependant, dans une version asynchrone, cette limitation n'est plus présente, car l'entraînement peut continuer même si ce ne sont pas tous entraîneurs qui ont soumis une mise à jour.

7.2 Utilisation d'une chaîne de bloc

Plutôt que d'utiliser un système comme MQTT ou une connexion pair à pair, il serait possible de déployer le réseau sur une chaîne de bloc. La chaîne de bloc est un système hautement redondant qui fournit une méthode de coordination avec une très haute disponibilité.

De plus, les fonctionnalités de contrats intelligents permettent d'exécuter du code de façon décentralisé sans avoir besoin de faire confiance aux entraîneurs. Ainsi, il serait possible de valider les mises à jour soumises et de générer les identifiants des modèles à partir des contrats intelligents. Ils peuvent aussi émettre des évènements pour notifier les entraîneurs et démarrer de nouveau processus hors chaîne.

L'annexe II démontre un exemple d'implémentation de contrats intelligents qui auraient ces fonctionnalités. Le code est écrit en Vyper. Ce langage a été choisi pour sa syntaxe similaire à Python.

L'utilisation de clés privées traditionnellement utilisée par les chaînes de bloc permettrait aussi de s'assurer de l'authenticité de l'émetteur de la mise à jour. Il serait donc impossible d'avoir des imitateurs qui envoient des mises à jour au nom d'un autre entraîneur.

Finalement, l'ajout de la chaîne de bloc permettrait de créer un système économique pour soutenir financièrement les entraîneurs et les inciter à fournir de bonnes mises à jour. Il serait aussi possible de mitiger certaines attaques comme des attaques de SPAM en les rendant plus couteuses à exécuter. Le plus gros défi à ce niveau est de trouver comment prévenir des attaques plus subtiles comme republier une mise à jour d'un autre entraîneur en changeant les poids légèrement.

7.3 Règles plus intelligentes

Alors que les résultats démontrent que les règles sont plutôt efficaces pour atteindre l'objectif d'accélérer la convergence des modèles, il serait intéressant de trouver des euristiques plus rapides. En effet, les règles de sélection fonctionnent bien, mais sont lentes à exécuter, ce qui ralentit grandement le processus.

Par exemple, pour éviter d'évaluer tous les modèles de la dernière ronde pour choisir les modèles à entraîner, le graphe pourrait être utilisé pour seulement tester les branches auxquelles l'entraîneur a déjà contribué.

Au niveau des règles de sélection des mises à jour, il serait intéressant d'utiliser la distance cosinusoïdale pour tenir en compte la direction de la mise à jour. En effet, la divergence des poids calcule la différence absolue entre chacun des poids sans considérer si le poids du paramètre augmente ou diminue.

7.4 Conclusion

Bien que les résultats soient concluants et prometteurs, beaucoup de choses restent à faire pour rendre le système réellement utile. Par exemple, l'ajout d'une chaîne de bloc serait particulièrement intéressant pour augmenter la traçabilité de chacun des nœuds ainsi qu'ajouter un système d'incitatifs financiers. De plus, une version asynchrone pourrait être plus adaptée pour un apprentissage entre appareils si les règles de sélection de modèles sont adaptées pour réduire le travail nécessaire pour sélectionner le bon modèle.

ANNEXE I

MULTICONFEDERATED LEARNING: INCLUSIVE NON-IID DATA HANDLING WITH DECENTRALIZED FEDERATED LEARNING

Michael Duchesne[1] , Kaiwen Zhang[1] , Chamseddine Talhi[1]

[1] Département de Génie Logiciel et TI, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Article soumis à la conférence « Middleware 2023 » en mai 2023.

1. Introduction

Data privacy problems are growing everywhere. Therefore, it is critical to continue to innovate with privacy-preserving machine learning techniques Liu *et al.* (2021); Xu, Baracaldo & Joshi. Federated learning (FL) is a step in the right way because it unlocks siloed data by enabling collaborative training between data owners while respecting the restrictions on their data. FL works by training the models on the devices of the data owners Yang, Liu, Chen & Tong (2019). Then, the devices communicate with an orchestrator, which aggregates all the updates from the devices to create a single global model.

Although FL allows for the training of powerful global models using crowd-sourced data from a large number of learners without compromising their privacy, it suffers from several important issues. First, the presence of an orchestrator introduces a single point of failure, since aggregation is a critical component to produce the desired global model. Moreover, there is no guarantee the federated learners have independent and identically distributed (IID) datasets which can reduce the effectiveness of training. This problem commonly occurs in a FL setting since there is no guarantee that each learner will collect local data representative of the entire set and the learners cannot share complementary data with each other due to privacy concerns. Performances can be reduced by up to 50% in some cases depending on the level of skewness between learners Zhao *et al.* (2018).

To overcome the problem of the central point of failure, Decentralised Learning (DL) systems like Swarm Learning Warnat-Herresthal *et al.* (2021), BFLC Li *et al.* (2021) and Brain TorrentRoy *et al.* (2019) have been proposed. These methods eschews the dependence on a central actor by connecting the learners using a peer-to-peer network or a blockchain.

This paper aims to improve the collaboration of siloed data owners by proposing a generalized decentralized approach of FL. Our solution considers the non-IID challenges in supervised and unsupervised settings, while retaining the privacy related benefits.

We propose MultiConfederated Learning (MCFL), a decentralized FL approach. Contrary to traditional FL, various models exists simultaneously instead of having a single global model. Furthermore, learners can select the most relevant models based on the performance of that model using their local dataset. Each model thus acquires a specialization by being trained only from a subset of all learners with data sharing similar features. Moreover, they can train multiple models and propose new updates by aggregating the updates of their peers. A forking mechanism is created as learners can choose different combinations of updates. This mechanism gives learners the necessary flexibility and independence to create a model that fits their data, while making their knowledge accessible to other peers. Moreover, we propose control mechanisms to converge to a single model when possible. The freedom given to every learner on the network to select models and updates to aggregate gives them the choice to select what is best for them while collaborating with other learners to achieve better results in less time. For this reason, we decided to use the term confederation for our solution to reflect the analogy of power sharing among a group of sovereign entities collaborating to achieve a common objective.

The main contributions of this paper are the following :

1. We propose a novel approach to decentralized Federated Learning where multiple models are trained in parallel. Each learner uses a selection algorithm to decide the set of models to contribute. Moreover, the network uses transfer learning between groups to converge to a single model whenever possible. Models can be forked to satisfy every learner.

2. We present a generalizable solution that takes advantage of the knowledge of multiple models using a locally trained machine learning algorithm.
3. We compare the performance of our proposed approach with that of using the baseline FedAvg. Furthermore, we empirically demonstrate the benefits of the convergence feature of MultiConfederated Learning.

The structure of the paper is as follows. Section 2 covers previous related papers. Then, Section 3 covers background concepts such as FL, DL, non-IID data, and multi-task learning. Section 4 presents the architecture and details of our proposed solution MultiConfederated Learning. Section 5 shows the implementation details and the results of our evaluation in IID and non-IID scenarios. The paper concludes with Section 6.

2. Related works

This section explores the various works on Federated Learning and non-IID data. While previous papers dealt well with parts of the challenges highlighted in the introduction, this section will outline the advantages of our proposed approach over other solutions.

2.1 Non-IID data

In Zhao *et al.* (2018), the author manages to mitigate the impact of non-IID when using FedAVG by sharing a public dataset with all learners to include in their private training dataset. Contrary to this approach, our solution MultiConfederated Learning does not require any public dataset to be shared, although it can still benefit from having a small balanced public dataset to create a more generalizable solution.

A Divide and Conquer FL algorithm was implemented in Chandran *et al.* (2021) with good results because their strategy was based on limiting weight divergence. We adopt the same strategy of trying to limit weight divergence, but quite differently by grouping learners based on their preferred model.

In Shoham *et al.* (2019), the authors demonstrate that learning from a non-IID dataset is similar to learning multiple tasks. However, multi-task learning tends to suffer from forgetting previous tasks to optimise for the current task. To overcome forgetting in FL, the authors used EWC Kirkpatrick *et al.* (2017), which modifies the loss function to mitigate the issue. This approach assumes that the global model is over-parameterized to learn all the tasks successfully. On the other hand, learners cannot be sure if the model has enough parameters as they can only see their own part of the dataset. For this reason, the proposed approach will fork the best model for a new task if it deteriorates the performance of the model for another task.

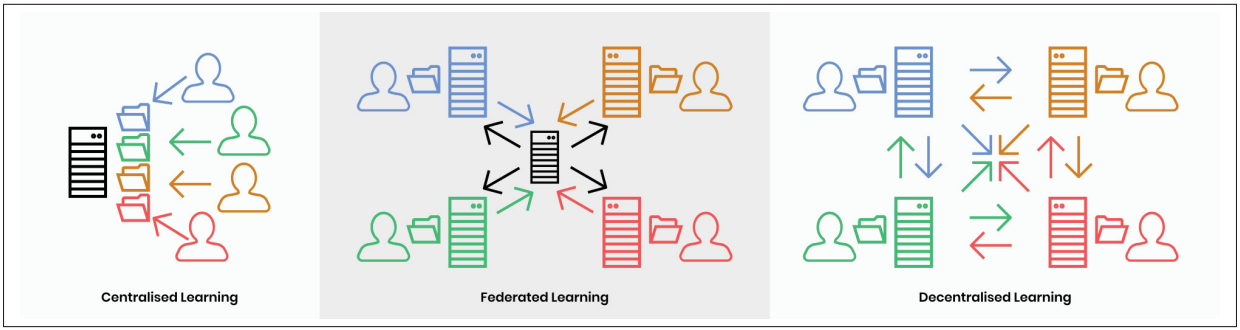

Figure-A I-1 Overview of the three types of machine learning network deployments.

2.2 Federated Learning

FL algorithms have been proposed, such as FedProx Li *et al.* (2020) and Scaffold Karimireddy *et al.* (2021). The former attempt mitigates performance reductions from non-IID data by tolerating partial work and limiting the update range. The latter is adding a bias during the training to align all learners to the same optimal parameters. It produces excellent results but may not have an optimal joint solution. These works are complementary to our proposed solution and could replace FedAvg.

Update selection and clustering have been tried and proven in the Federated Learning settings. HypClusters Mansour *et al.* (2020) shows clustering can be beneficial for FL. In Ghosh *et al.* (2021), the orchestrator clusters clients according to their model using k-means. Hierarchical Clustering in Briggs *et al.* (2020) also clusters updates based on the Euclidian or Cosine distance

of the models. A direct acyclic graph (DAG) based approach was proposed in Beilharz *et al.* (2021), which always merges two updates. Finally, Kopparapu & Lin (2020)'s approach also builds a DAG with fork and merge abilities using the EWC techniques.

Similarly to these works, our proposed approach also employs clusters, but it does not limit a learner to a single cluster. Moreover, our approach lets the learner decide which groups they belong to at the start of every round. For this reason, we also refer to clusters as groups in our paper. Nonetheless, most of the previous works fail to consider weight divergence between the updates and assume over-parameterization of the model for the task when merging. As a result of clustering, the models are not a generalized solution. In our approach, we offer a way to access the global knowledge of the network if the proposed approach does not converge to a single cluster.

Decentralised Learning is an adaptation of Federated Learning which does not have a central orchestrator (see figure I-1). Instead, it connects all learners through a peer-to-peer (P2P) network as in Roy *et al.* (2019) and delegate the aggregation of models to nodes within the networks. Blockhain has also been used in Li *et al.* (2021) where a comitee evaluates new iterations of the global model before appending them to the chain. Smart contracts were used in Swarm Learning Warnat-Herresthal *et al.* (2021) to aggregate models. Our approach improves coordination by grouping learners to reduce the adverse effect of non-iidness. We also significantly change the training process as we train multiple variations of global models. It can be seen as if every node is running its own Federated Learning instance, so they keep total control of the training and aggregation process.

3. Background

In this section, we explore the essential techniques employed to implement the proposed solution successfully. We start by elaborating on Federated Learning and Decentralized Learning. Then, we give the reader more information on the challenges of non-IID data and explain how it is related to multi-task learning and transfer learning.

3.1 Federated Learning

As suggested by Yang *et al.* (2018), Federated Learning is an emerging technique that train models on the devices and aggregate models on a centralized server. Thus, the central service does not have direct access to any data. It will randomly sample a subset of available devices and starts a training round with them. Then, selected devices have a defined amount of time to train and send their updated models to the server. Once the round is over, the aggregate of the models becomes the new global model for the network. The server then sends the global model to the devices for the next round. The most straightforward aggregation is done using FedAvg. Assume that there are K learners in round n training the model W_n. The aggregator will determine a weight (w^k) for every model according to the number of samples on which it was trained and sum them all up. As a result, the sum becomes an average of all models (see Equation A I-1).

$$W_{n+1}^{g} = \sum_{i=1}^{K} \frac{W_n^k}{w^k} \tag{A I-1}$$

With respect to communication costs, FL can be demanding depending on the size of the model. For example, in a network of K clients with sampling rate r, the aggregator will receive at most $K * r$ models. For this reason, the number of models transferred by the aggregator increases linearly with the number of clients. On the other hand, clients only need to send and receive at most one model per round.

3.2 Decentralized Machine Learning

Decentralized Machine Learning (DML) removes the centralized server for aggregation and relies on a Peer-to-Peer network or a distributed ledger to handle communication between learners. The removal of the aggregators push the responsibility to the learners to aggregate the updates. Moreover, DML implementations like Warnat-Herresthal *et al.* (2021) considers learners as sovereign actors as opposed to most FL implementation. Indeed, FL is often used to leverage data from clients by the aggregator's operator.

As DL operates in a decentralised network, it tends to communicate more models compared to FL, because the nodes have to send their models to every other node. Assuming the worst case scenario where every node participates to the training in a network of K nodes, each of them needs to send and receive K models every round. Moreover, the lack of a centralized server implies that K^2 models are sent every round through the entire network since every node has to receive all models from their peers to perform the aggregation.

FL and DL can be executed in two variants : cross-devices and cross-silo. The former is the most well-known approach where thousands of devices are expected to participate in training. The latter usually assumes that the clients are organizations, reducing the number of expected participants Huang *et al.* (2022). Cross-silo is a more realistic scenario for DL since the communication cost increases exponentially with the number of participants.

3.3 Non-IID Data

In both FL and DL scenarios, all learners rarely have the same distribution of samples. For example, they can be distributed across different regions or have different habits that generate very different data. Learners must consider the difference in distributions because it significantly impacts model convergence. As shown by Zhao *et al.* (2018), weights from models trained on different distribution will tend to diverge more. While the models will perform well individually, their aggregation will be a worse solution. Moreover, the divergence between models can lead to slower convergence for the global model.

Data can be skewed in various ways ; for example, the dataset can be unbalanced, so certain learners have more data samples from a label than others or even completely absent classesZhu *et al.* (2021). Another example is label preference skew, which refers to improperly labeled data or learners disagreeing on a label for a very similar sample.

Moreover, non-IID data is hard to distinguish from a covert poisoning attack Jeong *et al.*. Indeed, as the attacker can simply switch labels for a class, it is hard to differentiate from a learner with a preference skew. Learners might also not know the entire domain of the data. This uncertainty

means that approaches ignoring updates like Guo *et al.* (2021) might exclude legitimate update from the network. Our proposed solution is more inclusive, since it lets the learner fork and train a separate model if some peers exclude them.

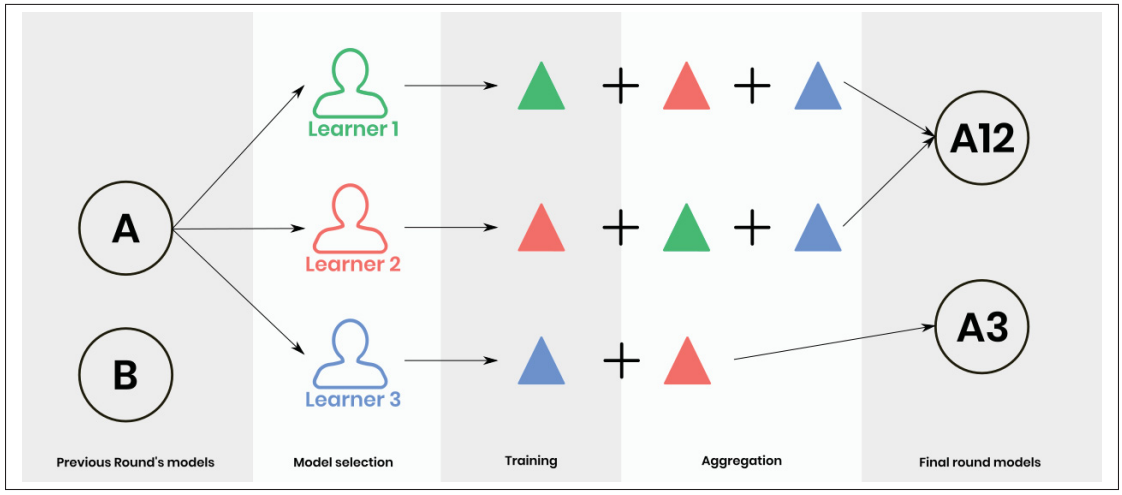

Figure-A I-2 Evolution of models during a single MultiConfederated Learning round.

3.4 Multitask Learning

In Shoham *et al.* (2019), FL of a non-IID dataset is considered as multitask learning. Indeed, take as an example a classifier trained by two learners where the first has pictures of cats and dogs while the second has pictures of birds and snakes. They are both trying to classify animals, but their data distribution does not let them properly train to identify all animals. As a result, the learners are learning a sub-task of the global task. Both tasks are learned simultaneously and have commonalities which can be transferred from one sub-task to the other.

When a NN attempts to learn tasks sequentially, it tends to forget previous tasks as it optimizes for the task currently learning Kirkpatrick *et al.* (2017). Similarly, forgetting can also occur in FL when learners fine-tune the global model Wang *et al.* (2019). As a result, its performance increases for data in the local distribution, but decreases for the rest. Although improved performance on the learner dataset is desirable, one key advantage of FL and DL is creating a generalizable solution for all learners.

Additionally, because of the commonalities between tasks, neural networks can transfer knowledge from one task to another using Transfer Learning Torrey & Shavlik (2010). This ability has the effect of reducing training time and improving generalization which can manifest in a better performance for new tasks.

4. Proposed solutions

We propose MultiConfederated Learning, a decentralized FL approach using FedAvg to reduce the impact of non-IID data distributed amongst sovereign entities. It is personalizable rule based and multifaceted approach using groups, transfer learning and weight divergence. In section 4.1 we present our assumptions and main architectural decisions. Then we expand on the training process in 4.2. Section 4.3, 4.4 and 4.5 elaborates on the forking mechanism during aggregation selection. Finally, we analyse communication costs in 4.6 and how to use multiple models to obtain a more generelizable solution in 4.7.

4.1 System Model

The proposed system model is a decentralized peer-to-peer approach based on Federated Learning. Each node acts as both the FL aggregator and the client, allowing them to train models and aggregate updates in synchronous round-based iterations. As a result, the system inherits the privacy properties of Federated Learning.

Since our approach is decentralized, learners cannot rely on a central server to assign groups and aggregate models. Therefore, learners use a DAG to track groups and models. We also introduce selection rules for nodes to choose the most appropriate groups for themselves.

Our solution does not consider malicious actors since non-IID data cannot be distinguished from covert poisoning attacks, and we don't want to exclude learners falsely from the network. Thus, we provide every learner with the ability to collaborate while retaining control over the final model published so they can exclude bad updates and excluded learners can publish their own version.

Figure I-2 presents a visual representation of the grouping model. In this figure, all learners decide to update model A, which causes model B to die in this round. The model is then forked into two different models as learner 3 decides to exclude the update from learner 1.

The overview of a round can be divided into 3 steps :

1. Learners select the most relevant models based on their own metrics. The selection is explained in Subsection 4.2.
2. Learners train the selected models and share their updated models. The training is also explained in Subsection 4.2.
3. Learners aggregate updates and publish their selection of accepted updates. Subsection 4.3 explains the aggregation strategy in more detail.

As discussed in 3.2 and evaluated in section 4.6, due to communication increasing with the number of participants, we consider our approach aimed for cross-silo usage. Consequently, we expect nodes to be hosted on good hardware that is always online with a reliable and fast connection. Moreover, nodes communicate through TCP, which guarantees the delivery of every message.

While our current solution is based on peer-to-peer communication, it could be implemented using a blockchain. Nevertheless, our main objective is to demonstrate the effectiveness of the grouping strategy, which does not require a blockchain.

Furthermore, our proposed approach prioritizes the autonomy of learners and provides them with considerable flexibility during the aggregation phase. Specifically, our approach allows them to selectively choose updates based on their weight divergence. We explain how this can creates fork and how it can be handled in Subsection 4.5. Although there is the possibility of adding new rules to provide additional protection against malicious actors, this is outside the scope of this paper and will be done in future work.

Finally, in Subsection 4.7, we propose using various models in a second machine-learning algorithm to create a more generalizable solution.

4.2 Network Initialisation and Learning Process

To begin with, the learners need to divide their dataset into three distinct subsets : training, validation, and test. The training and validation subsets are utilized during the training phase, whereas the test subset is used during the selection phase to ensure that the network has not been exposed to the samples to avoid any selection bias. The splits should be performed randomly to maintain the same distribution, and in proportions that are considered acceptable by the learners. To join the network, learners use an abstract discovery mechanism and introduce themselves to their peers using a randomly generated identifier. Although a more robust system could replace this mechanism, the approach assumes that no actors will attempt to impersonate other learners. After all the learners are connected, they begin with the genesis of the network. During this initial phase, each learner trains a model on their training dataset and shares it with their peers as a new model.

At the beginning of each round, after sharing their initial models, the learners enter the selection phase. For this phase, learners have to establish their relevant metrics for their use case. For example, it can be accuracy, loss, f1, etc. During this phase, each learner evaluates every model on their test dataset and records the corresponding performance metrics. Based on these metrics, each learner selects the best-performing models to train during the round. The selection of models is left to the discretion of each learner, who can use any relevant metrics to make their decisions. As the learning process is stochastic, some models will perform better than others on data that they have never seen before. Consequently, some learners may decide to switch to a model they never participated in its training, consolidating learners to the best models, and improving the generalisation capability of the model.

Learners can train more than one model to transfer knowledge about their data to other models through transfer learning. As previously discussed, learning from a non-IID distribution can be considered as a multitask problem, and transfer learning can be applied to help improve the model's performance from one task to another. The solution's implementation for transfer learning involves training the models of other groups as the learner would do for its own group.

While more sophisticated transfer learning techniques could be used, we believe it is enough to show the relevance of this feature in the system. After training, the learner shares the updates of every trained model. Finally, during the aggregation phase, learners select the updates they want to aggregate based on their aggregation rules. Subsequently, the learners communicate their choice with the rest of the network. Once the learners have received the choices from their peers, the round is finished, and they start another round in the selection phase.

4.3 Updates Aggregation

The proposed approach sacrifices the usual generalization of a global model by training multiple models to address the non-IID issue. The goal is to group learners around models instead of data. As a result, the non-iidness of data within the group should be reduced. By reducing non-iidness, we hope to reduce the divergence of updates and increases the convergence speed. In other words, learners should be grouping to train models where their updates will be the least divergent. Furthermore, as groups have similar data, they should be able to find a common optimal solution. Finally, every group ends up with a model tailored to the group members.

In contrast with previous clustered networks, the proposed approach does not limit learners to a single group Ghosh *et al.* (2021); Kopparapu & Lin (2020); Mansour *et al.* (2020). Learners have access to the updates and models of every group in the network. Moreover, they should contribute to multiple models simultaneously in an attempt to transfer knowledge between the groups and gradually expose other models to new data. Figure I-3 illustrates the network architecture, showing how learners group for aggregation and transfer of knowledge between groups. In this hypothetical network, nodes are split into three different groups, with some nodes belonging to two groups simultaneously. This allows those nodes to engage in transfer learning, which can facilitate convergence toward a single group.

The network constructs a DAG of the evolution of the models by always training a parent model, which would translate to a node in the graph. Learners always submit their updates to a parent model. Furthermore, learners can select a combination of updates to be aggregated for the next

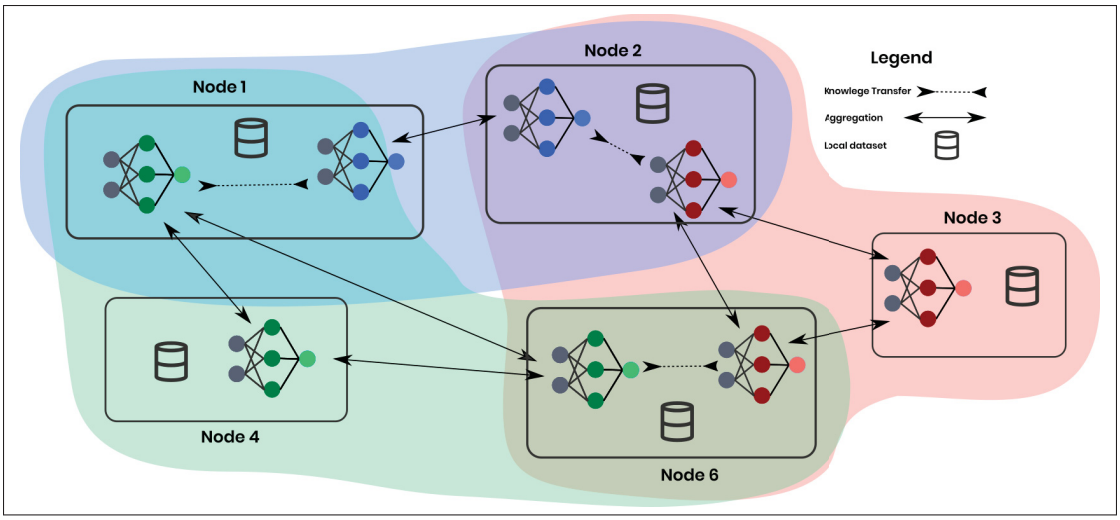

Figure-A I-3 Overview of a MultiConfederated Learning network with 5 nodes split into 3 groups.

round. It is important to note that each learner selects updates and publishes their selection of updates. Indeed, they are free to choose any updates according to their specifications and who they consider a good actor.

The updates are aggregated by each learner using FedAvg. The proposed approach uses the FedAVG algorithm for its simplicity and its deterministic process. Indeed, because learners have access to every model and update, they can individually do all aggregations for every model version proposed by their peers and end up with the same models. This ability spares the communication of any additional models. Moreover, aggregation is not a compute-intensive task compared to training a model.

Flexibility at the aggregation level creates different model evolutions, as different learners will choose different combinations of updates. The combinations create a forking mechanism where a parent model can have many children. As every learner can fork a parent model, it guarantees they will have at least one satisfying model for the round.

Moreover, learners should only publish updates to the best-performing model on their data, resulting in a natural selection mechanism. Indeed, a model that receives no updates will be

forgotten and considered a dead evolution. Nodes do not account for dead evolution which prevent the network from having an excessive number of models to choose from during the selection phase.

Although our implementation does not currently uses this, the DAG can also help reduce the search scope for good models by only selecting live models starting from a specific parent.

4.4 Deterministic Model Identifier

As mentioned previously, learners do not communicate aggregated models. Nonetheless, learners need to establish a consensus on an identifier for every aggregated model to publish updates to it in the next round.

To identify models, learners compute an identifier in a deterministic way, as shown in Algorithm I-1. Learners start by ordering the learner identifier of accepted updates, then concatenate all the identifiers in a string with the identifier of the parent model. To standardize the length of the identifier, the learners hash the string using SHA512 Handschuh (2005). Any modern hashing algorithm would be sufficient ; our objective is to have unique names with a maximum length in a deterministic way. With this process, learners obtain the same model identifier across the network without needing additional communication.

Algorithme-A I-1 Deterministic Model Identifier

Input : ID du modèle parent parentModelId, ID des entraineurs des mises à jour sélectionnés learnerIds
Output : ID de la combinaison

$learnerIds \leftarrow sort(learnerIds).join()$;
$id \leftarrow parentClusterId + learnerIds$;
return $SHA512(id)$;

4.5 Forks Management

As previously stated, the proposed approach gives a lot of freedom to the learners to select which models they train and which updates they decide to aggregate. The combinations created lead to more forks, increasing training time, and reducing training efficiency as learners spend more time testing models.

The network must strike a balance between the performance of models, the number of models, and model generalization. The proposed approach reaches this balance with a model selection rule that encourages the selection of more popular models. More specifically, the cluster selection algorithm uses the test metric and the amount of updates it received, defined as popularity, in the previous rounds to compute a score. Assuming there are n models, they will select the $\sqrt{n}$ best models to train during the training phase. We propose using a square root so that the number of trained models does not scale linearly with the number of models since selecting more models increases round time. On the other hand, it improves transfer learning between models. An overview is shown in Algorithm I-2.

Algorithme-A I-2 Selection des modèles à entraîner

Input : Modèles candidats models
Output : Modèles sélectionnés

$\mathcal{R} \leftarrow testModels(models)$;
for *i in range(models.length)* **do**
    $popScale \leftarrow sqrt(popularity(models[i]))$;
    $modelScores[i] \leftarrow modelResults[i] * popScale$;
end for
$rankedModels \leftarrow models.orderBy(modelScores)$;
$nModels \leftarrow sqrt(models.length)$;
return $rankedModels[0 : nModels]$;

After completing their local training and sharing their updates, learners must select which updates to incorporate into their aggregate using aggregation rules to exclude potentially bad updates. Weight divergence, as discussed earlier, is a primary reason for the reduced performance

of aggregated models, particularly in scenarios where data is non-uniformly distributed. To address this issue, we have devised an aggregation rule based on weight divergence, which is calculated using Equation A I-2 from Zhao *et al.* (2018). While it is possible to develop additional rules to meet specific learner needs (e.g., a rule for Byzantine resistance based on the trustworthiness of other learners), our focus is on accelerating model convergence in non-IID settings by mitigating weight divergence.

$$WeightDivergence = \frac{||W^{PeerUpdate} - W^{LocalUpdate}||}{W^{LocalUpdate}} \tag{A I-2}$$

As shown in Algorithm **??**, the rule requires a configurable value called the tolerance. This value should be a number of standard deviation and will impact how many updates are selected. It calculates the weight divergence between the learner's update and each received update, and then computes the median and standard deviation of the weight divergences. Using these statistics and the tolerance value, it selects the updates that are "similar enough" to the learner's update (i.e., have a weight divergence within the tolerated range). The goal is to avoid aggregating updates that are likely to reduce the accuracy of the aggregate model because their weights are diverging too much. A higher tolerance will include more updates and reduce forks, but will lead to aggregating more divergent updates, which can impact the model. On the other hand, a lower tolerance level excludes more updates from the aggregate, leading to an increased number of possible combinations which results in more forking.

4.6 Communication Costs

In MCFL, the nodes communicate models with each other through a peer-to-peer network. As a result of training multiple models, the learners have to transfer more models compared to the traditional DL network for each round. For example, in a network of K nodes that train M models, a single node will send $M * K$ models. Although it increases communication cost linearly with the number of models being trained, we do not consider it to be more problematic than for DL, because the time for a round in MultiConfederated Learning also increases linearly

Algorithme-A I-3 Update Selection

Input : MAJ du noeud myUpdate, MAJ candidates updates, seuil de tolérance tolerance
Output : MAJ sélectionné

```
1  for i in range(updates.length) do
2  |  divergences[i] ← wd(myUpdate, updates[i]);
3  end for
4  med ← median(divergences);
5  maxDiv ← med + std(divergences) ∗ tolerance;
6  selIndex ← 0;
7  selected[selIndex] ← myUpdate;
8  for i in range(updates.length) do
9  |  if divergences[i] < maxDiv then
10 |  |  selIndex ← selIndex + 1;
11 |  |  selected[selIndex] ← updates[i];
12 |  end if
13 end for
14 return selected
```

with the number of models trained. As both round time and communication increases by the same factor, the transfer of models can simply be done on a longer timeframe at the same rate.

In addition, learners have the overhead of communicating their aggregate models when compared to FedAVG. However, they only need to share their selection and no models, as discussed in Section 4.4. For this reason, we consider the communication overhead to be negligible when compared to traditional DL.

4.7 Generalizable Solution Utilizing Multiple Models

The network contains multiple models with different knowledge and biases as they were trained on non-IID data. While each has partial knowledge of the task, the network should have enough knowledge to have a generalizable solution.

Moreover, models tend to have noisy outputs and little confidence in their predictions for samples outside the distribution on which they were trained. However, a second local machine learning

can learn to filter that noise and use the outputs of multiple models to make more reliable predictions using an architecture as seen in Figure I-4. Indeed, it shows a local classifier can use the predictions from multiple models to make a more reliable prediction.

The local model can be a simpler solution than a neural network such as decision trees, SVM, or Naïve Bayesian models. Furthermore, the new model requires fewer data because it does not need to learn any features, only to filter noisy outputs.

To train the outputs classifier, learners extract models from the network and use them to build a new dataset of labelled model outputs. As seen in Algorithm I-4, they transform all their samples using a forward pass on each sample for each extracted model. The outputs for every sample are concatenated and will become the new input data for the classifier while keeping the same label. Subsequently, learners can train the classifier locally using the new dataset. In this regard, using Principal Component Analysis Bro & Smilde (2014) can be beneficial to reduce the dimension and remove redundant outputs before training the local model.

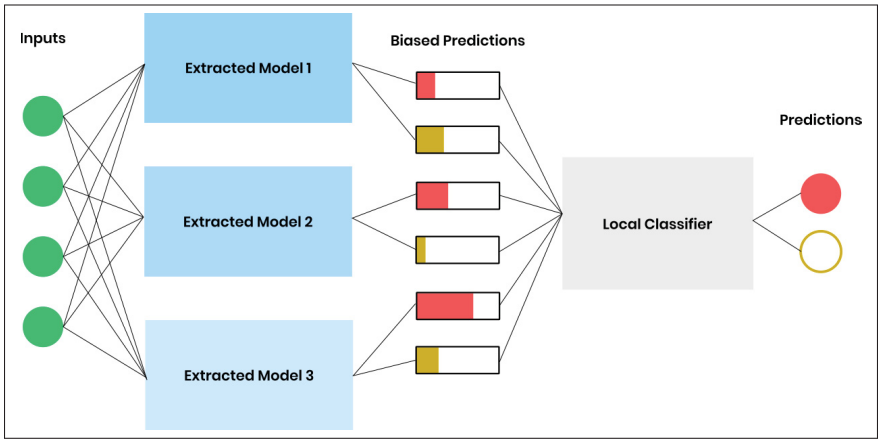

Figure-A I-4 Generalizable local multi-model solution for each learner.

5. Results

In this section, the performance of the proposed approach is assessed. First, the experimental setup is described. Second, the proposed approach is compared to the baseline solution. Finally, the results show that multiple models can give better performance and faster convergence.

Algorithme-A I-4 Local Classifier Training

Input : inputs x, labels y, selected models
Output : Trained classifier

```
1  for i in range(x.length) do
2  |   out ;
3  |   for j in range(models.length) do
4  |   |   out[j] ← models[j].forward(x[i]) ;
5  |   end for
6  |   x'[i] ← flatten(out) ;
7  end for
8  x' ← pca(x');
9  classifier ← newClassifier() ;
10 classifier.train(x', y);
11 return classifier;
```

Moreover, we demonstrate how using a second ML algorithm with a few balanced labels can enable learners to generalize the knowledge from multiple models.

5.1 Experimental Setup

We implemented the system in Python 3.9 using Pytorch Lightning 1.8, a framework built on Pytorch. The learners have access to an MLFlow instance to store information about their training. The MLFlow instance relied on a PostgreSQL database to store its data. All nodes were deployed in Docker containers on a bare metal server running Ubuntu 22.04. The server uses dual Intel Xeon Gold 5118 CPU totalling 48 cores with 126GB of RAM. All the nodes in the network are doing training and aggregation. Nodes send messages peer-to-peer through an abstracted communication layer (MQTT). Moreover, learners do not communicate their models directly. Instead, they save their models to a location accessible to other peers and send the URL to the model.

Dataset : The network was tested using two computer vision problems with MNIST and FashionMNIST datasets. MNIST consists of handwritten letters while FMNIST contains pictures of clothings. Although these datasets are usually balanced, we distributed samples in different

non-IID ways. The distributions are explained before the experiments. This methodology for generating non-IID data has been previously used in many papers such as Li *et al.* (2020) and Wang *et al.*.They are convenient datasets for proving concepts using the same input dimensions with well-known comparable benchmarks. At the start of every experience, the dataset is split between all the learners. Data samples are not reused between learners. All the tests are run using 38 learners for 102 rounds with two epochs per round.

Models : Two models are used for the classification tasks. First, the LeNet architecture from LeCun *et al.* (1989). Second, a lighter version where the channels of the two convolutional layers were reduced from 6 to 3 and 16 to 9. While the LeNet model works for supervised problems, we also want to demonstrate the ability of the network to work with unsupervised learning problems. For this reason, we chose to train autoencoders. They are a type of neural network that tries to replicate its inputs to the outputs by encoding the features in smaller latent spaces. The architecture of our autoencoders reuses the first two convolutional layers of the LeNet model as the encoder and mirrors them for the decoder. The baseline local training achieved a satisfying level of accuracy with ten epochs.

Tableau-A I-1 Performance of local learners with all samples

	MNIST		FMNIST	
Model	Accuracy	Loss	Accuracy	Loss
LeNet	97%	0.089	83%	0.46
LeNet Light	96%	0.16	77%	0.57
LeNet CAE	-	0.006	-	0.015

Baseline Solution : We implemented the baseline solution by reusing the architecture of our proposed approach. As a result, the baseline solution is a Decentralised Learning network where all learners participate in every round and perform the FedAvg aggregation.

Metrics : The performance of the models are measured on the learners dataset after the aggregation phase without fine-tuning. For classification problems, we use accuracy as shown

in Equation A I-3. TP stands for true positive, TN stands for true negative, FP stands for false positive, and FN stands for false negative.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{A I-3}$$

The Mean Squared Error (Equation A I-4) is used for the autoencoder task. y_i stands for the target prediction while $\hat{y}_i$ stands for the model output.

$$MeanSquaredError = \frac{1}{N}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{A I-4}$$

5.2 IID Experiments

The first results in Table I-2 show the weight divergence selection mechanism is helpful to outperform FedAvg even on IID datasets. For this experience, we have simply distributed so that each learner has a balanced dataset. MultiConfederated Learning outperformed the FedAvg implementation, with the worst performer from MultiConfederated Learning almost outperforming the average learner in the baseline solution. As a result of the limited weight divergence between aggregated updates, the models' performances are more predictable, and the gap between the worst and best results is smaller than with FedAvg. Moreover, the selection algorithm only selects the best-aggregated cluster, which helps with a slightly faster convergence rate in this scenario. In fact, the average performance is 20% higher with the proposed approach for the first round but fades to 2-3% improvements by the fifth round.

Tableau-A I-2 Performance Networks on IID Dataset

Algorithm	Minimum	Average	Maximum
FedAvg	96.2%	98.2%	99.5%
MCFL	98.1%	99.2%	99.8%

5.3 Non-IID Experiments

The proposed approach was also tested on various non-IID cases. In our first experiment, we split the learners into three subsets and assign classes exclusively to one of the subsets. The subsets of learners split data equally amongst them. As a result, every subset becomes IID as they share data from the same class, but the entire dataset is distributed in a non-IID way through the network. Figure I-5 shows the accuracy of the aggregated model on the learner's dataset. It is a challenging scenario for the baseline solution to converge. Even after 100 rounds, the global model is not performing as well as in an IID scenario, with the worst learners having 82% accuracy.

On the other hand, the learners from the proposed approach end up grouping properly and contributing to models which perform well for their data. They achieve to accurately label data from their dataset with higher accuracy than the IID scenario, as classifying the subset is an easier task with fewer labels than the baseline task. However, their accuracy falls if tested on a balanced dataset as the models cannot label data outside of the group's distribution. This case fails to do transfer learning as the groups have very different distributions. However, their current groups are very effective.

We empirically tested the effects of various tolerance thresholds explained in 4.3 on a new non-IID scenario. This time, we distribute the labels to the learners using a different normal distribution for every learner. This case does not have clear groups ; every learner has overlapping data with many other learners.

Figure I-6 shows the accuracy of the network for different tolerance thresholds. Many spikes can be seen in the figure which are the results of the popularity modifier to encourage merging and transfer of knowledge between models. Those degradations only impact a single cluster and it quickly recovers. The experiment shows that natural selection is working as expected, learners are trying new evolutions by transferring knowledge, but sometimes it does not work correctly. In this case, we consider the advantage of cluster convergence outweighs the temporary degradations in performance as it can lead to more generalized models.

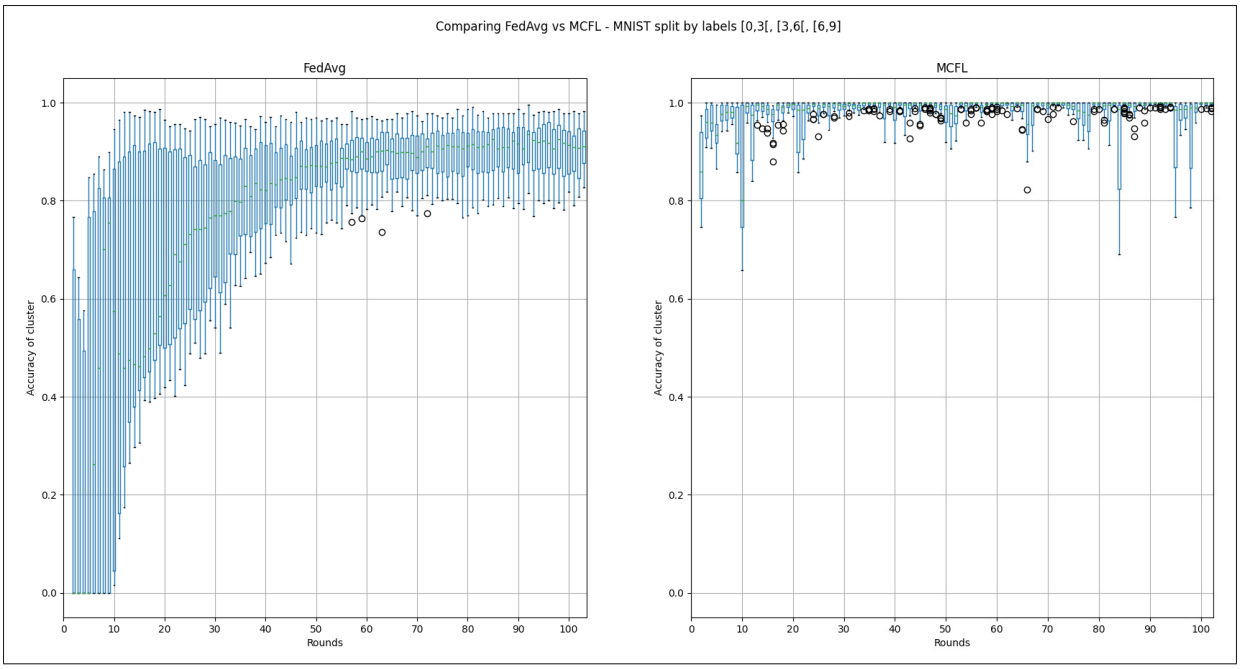

Figure-A I-5 Accuracy of learners using FedAvg vs. MultiConfederated Learning.

The mix of threshold has interesting performance with less volatility in both accuracy and cluster counts in Figure I-7.

Moreover, Figure I-7 shows a lower tolerance threshold leads to more forking. It is also possible to see that a standard deviation of 3 makes the network converge to a single cluster, while the other cases do not converge. This result makes sense as three standard deviations mean learners will include most updates.

The grouping approach also works with non-supervised problems as we demonstrate by training autoencoders. The experiment splits the learners into two groups ; the first group is assigned MNIST, the second FMNIST, and both datasets are divided between all their learners. By training the LeNet CAE model with an aggregation tolerance of 3 standard deviations, the results show that the network was successfully grouped to train two models with different performances for the two datasets (Figure I-8). The figure also shows the global model performance of the

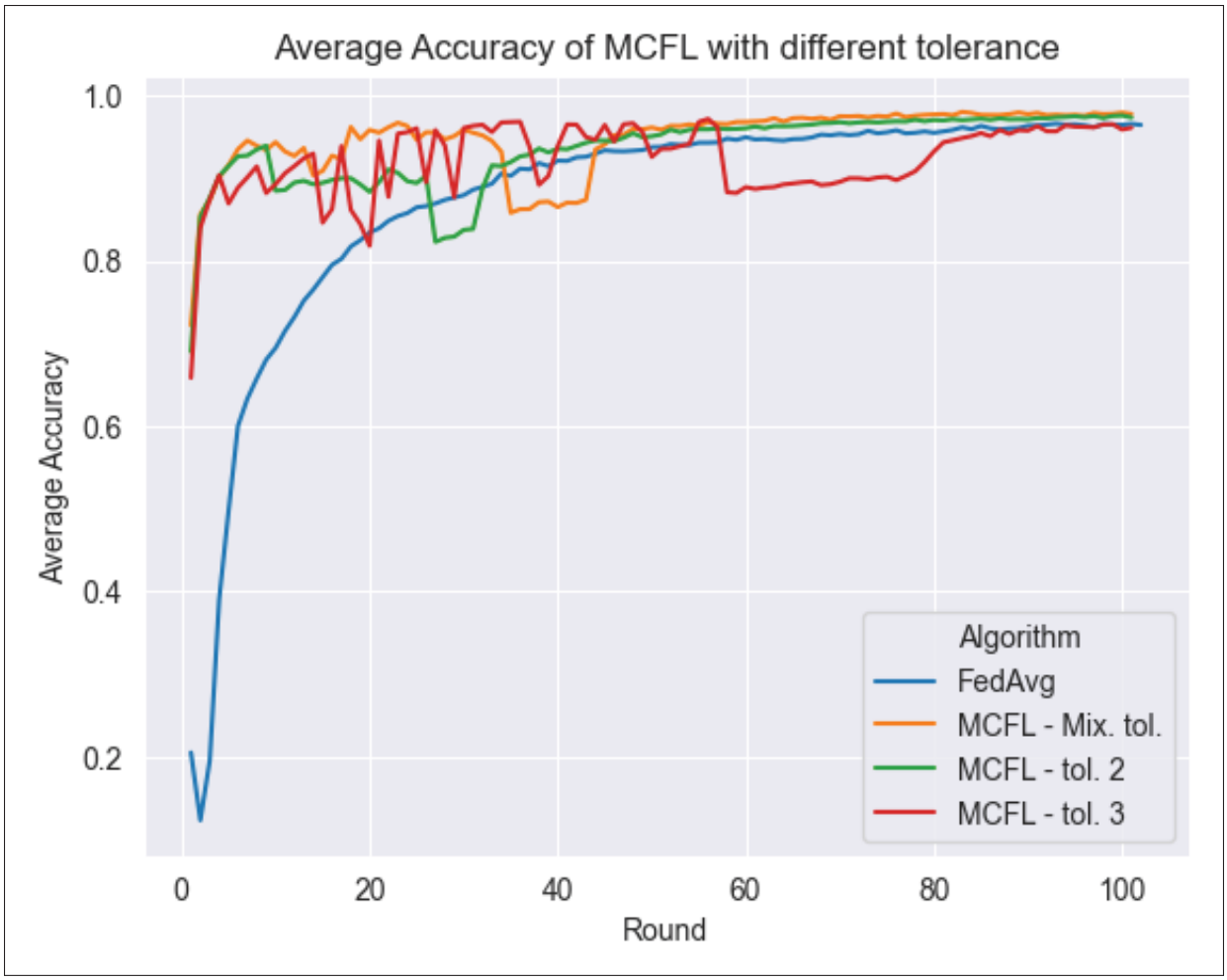

Figure-A I-6 Average accuracy of multiple update selection tolerances with data distributed with no clear groups

FedAvg network, and as expected, the models from our approach outperform FedAvg by 24.5% for MNIST and 17.5% for FMNIST.

5.4 Experiments with Multiple Models

Three classifiers are extracted from the network of the three subsets experiment. Each model has learned different features from different distributions. Using the highest output of the different models yielded an accuracy of 55%. Furthermore, we tried various classification algorithms with and without PCA reduction as discussed in Section 4.7. Figure I-9 shows the accuracy of these

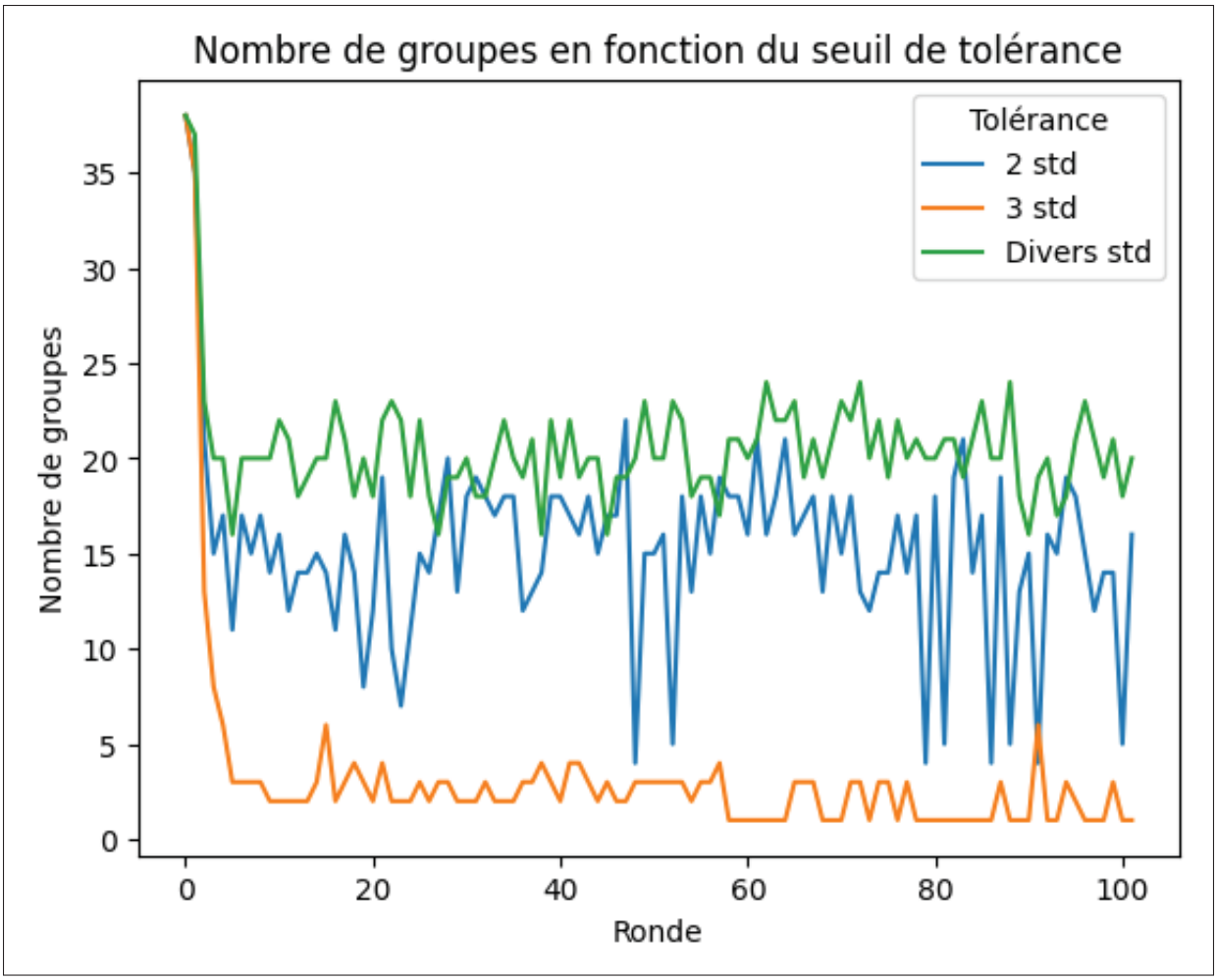

Figure-A I-7 Number of models generated using various update selection tolerances with data distributed with no clear subsets

algorithms depending on how many data samples were used for training. With only 1000 labels, the random forest reached the same accuracy as the average learner in the FedAvg network.

To demonstrate how the proposed approach can deal with preference skew, we configured an experiment to run an MNIST and FMNIST classifier simultaneously on the network by assigning the former to half of the learners and the second to the other half. Both halfs try to classify labels from 0 to 9, meaning the digit 0 and a t-shirt would get the label 0 assigned. The learners use the LeNet Light model and converge to 2 main models. FMNIST learners reach an accuracy of 85%, while MNIST learners reach an accuracy of 98%.

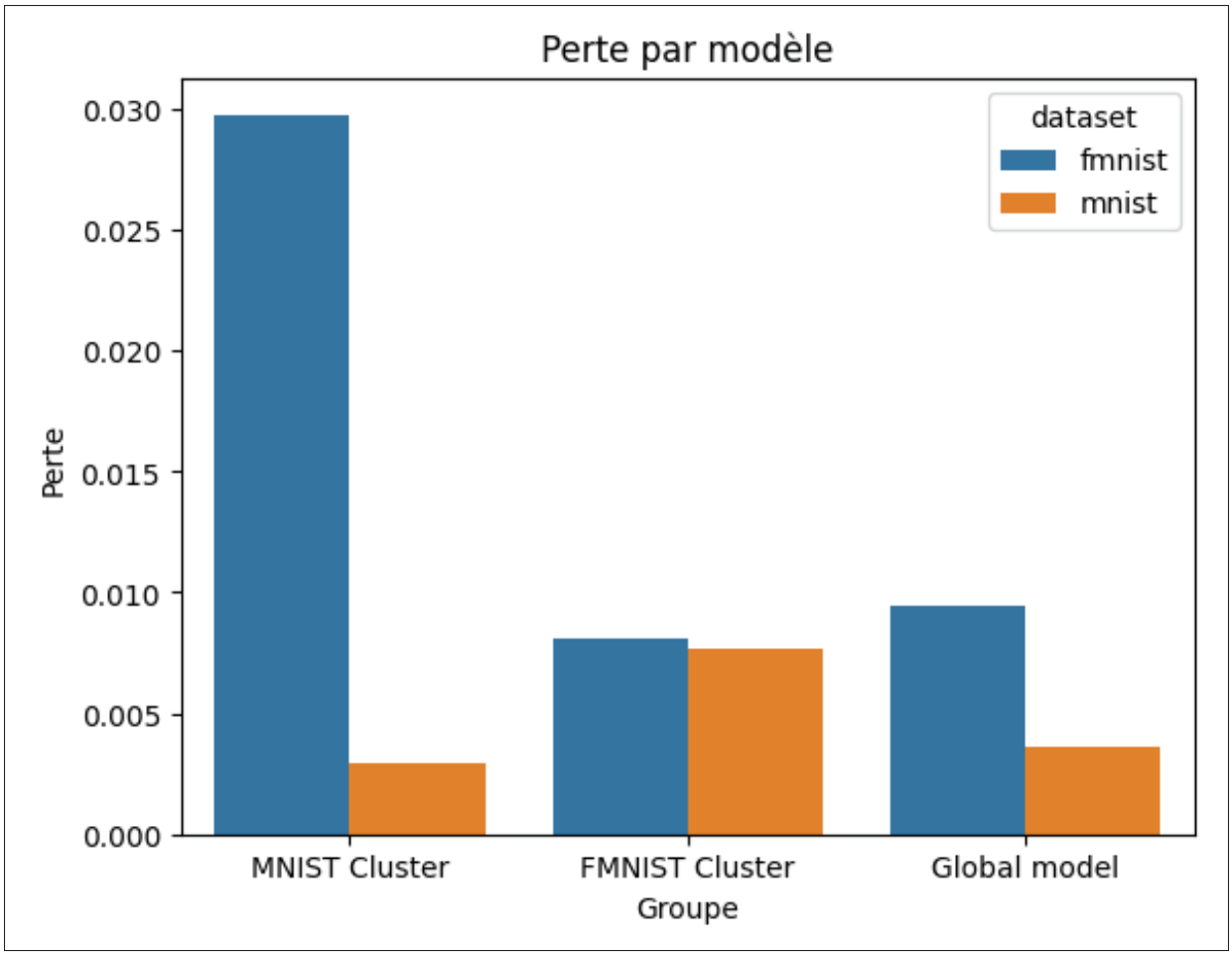

Figure-A I-8 Loss per model for an unsupervised task of learners using FedAvg to train a global model with the proposed grouping approach

By extracting both models, they can be used to relabel the data, so a digit is 0 and a t-shirt is 10. As shown in Figure I-10, some algorithms can learn to relabel outputs with as little as 50 samples or approximately two samples for each class. Moreover, the network is learning multiple tasks at the same time, but this also shows the ability of learners to correct labelled data or divide it into sub-classes with some local training and a few samples.

5.5 Results Discussion

The results show that it is possible to have models converging much faster by grouping learners. We also demonstrated the ability of the network to converge to a single model.

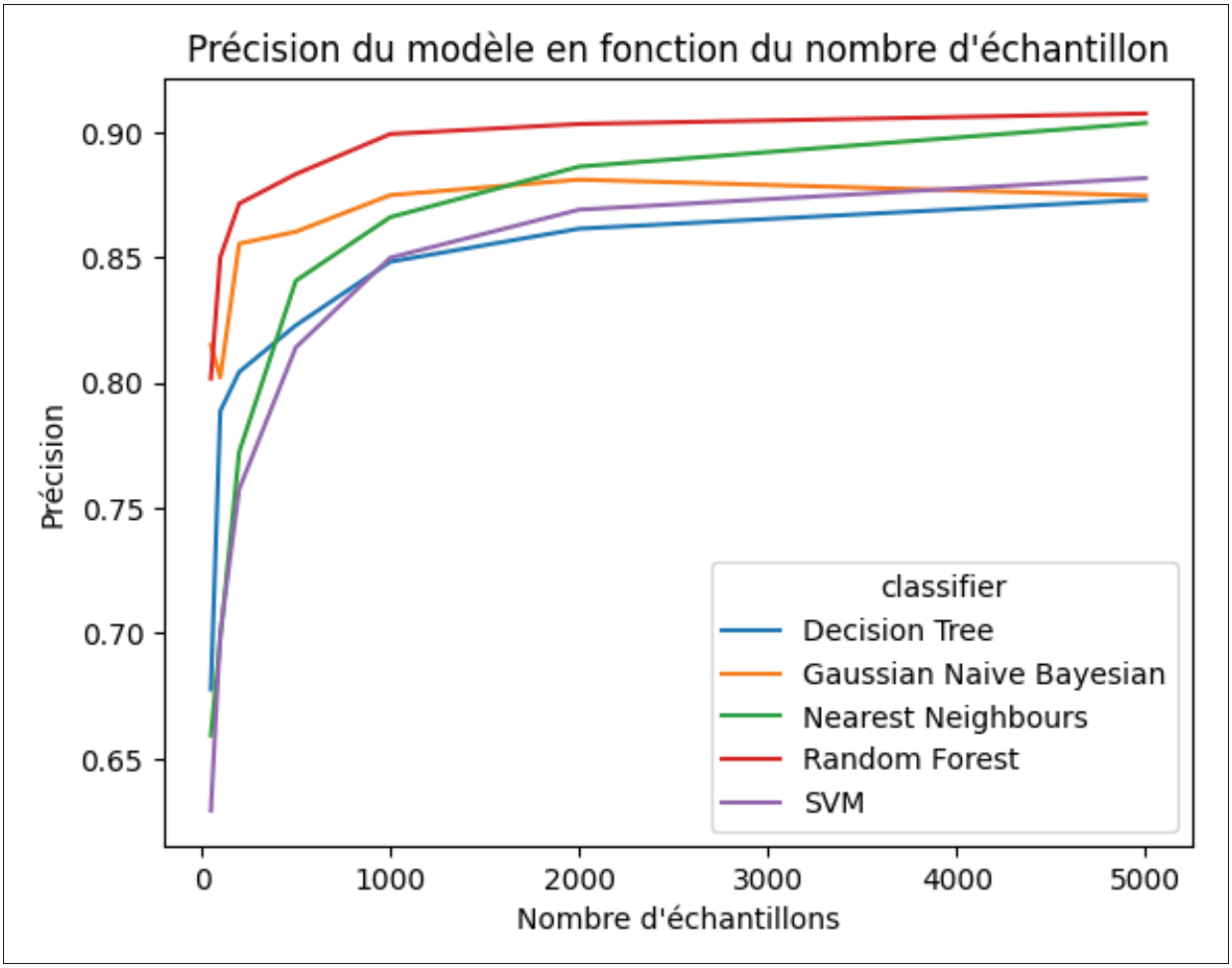

Figure-A I-9 Accuracy performance for different classification algorithms using three accurate models for different subsets of the data. Outputs from the models are used as inputs in the classification algorithm. The classifier ends up with a generalized solution.

Moreover, the combination of multiple models and additional machine learning using models trained for 10 rounds reached an accuracy of 90% on a balanced dataset, while the FedAvg implementation achieved it in 100 rounds. Although, to be fair, it is hard to compare one round of MultiConfederated Learning to one round of FedAvg, as MultiConfederated Learning trains a variable amount of models which makes the round longer. Nonetheless, as a result of the 10 times fewer rounds, we consider MultiConfederated Learning faster as long as the learners are not training 10 models per round. Furthermore, all our experiments had from 1 to 25 clusters,

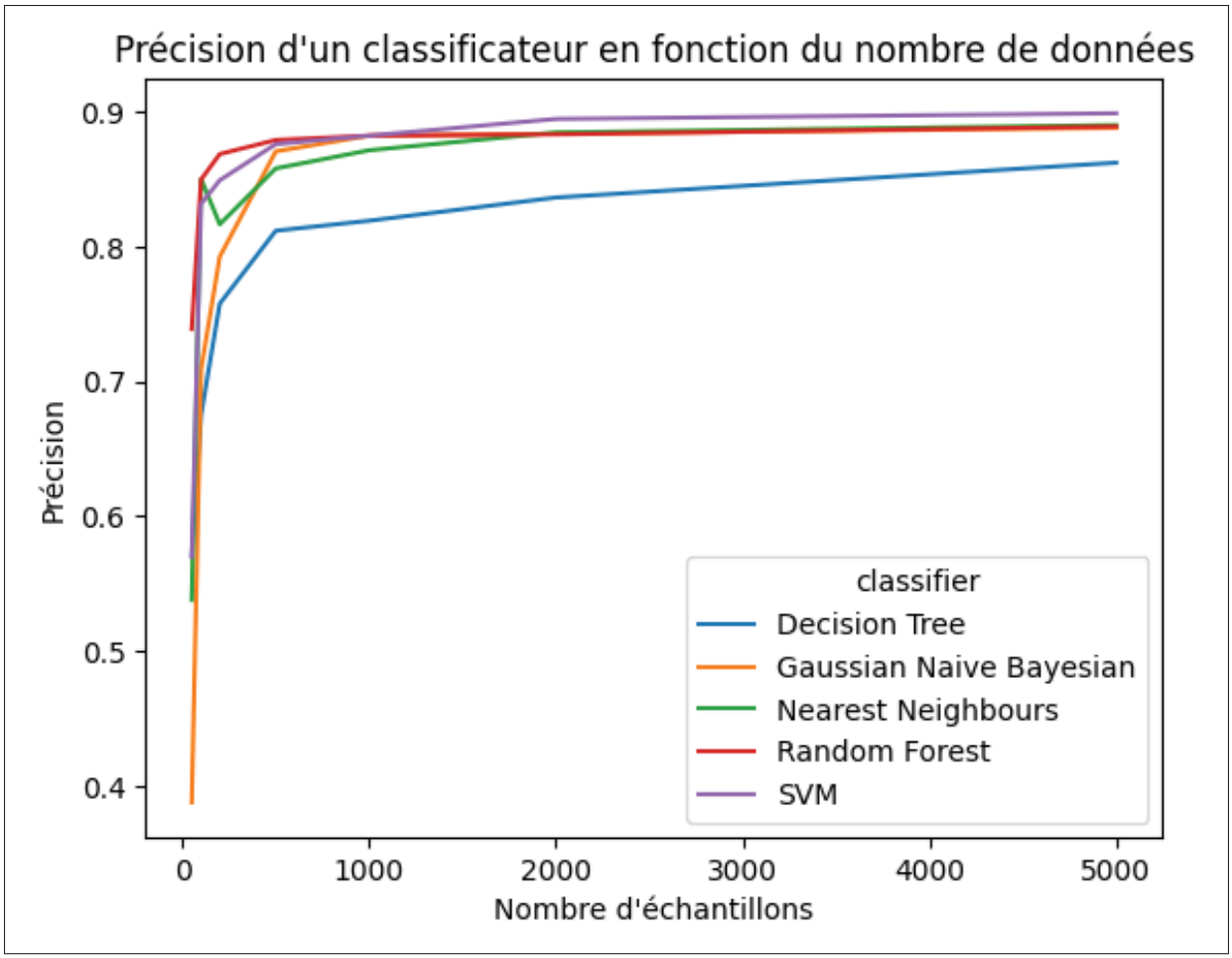

Figure-A I-10 Accuracy performance for different classification algorithm using one cluster for MNIST and one for FMNIST, each have 10 outputs. The local training classify outputs from them to relabel them to 20 classes.

which would make learners train between 1 to 5 models. Thus, we consider our approach faster by at least a factor of two.

One of the other advantages standing out with this approach is that having multiple models can prevent some issues with neural networks not being able to learn all the features of the dataset optimally. As shown in Figure I-8, the global model can learn both tasks, but not as well as the two models.

Moreover, it is impossible to know what the data distribution is for other learners. As a result, it can be hard to create a model that can learn all the dataset features. Due to this limitation, it could very well happen that a FedAvg network is training a model that is too small to learn all tasks. However, the proposed approach is mitigating this issue by forking models that focus on different tasks.

The proposed approach delivers the best of both worlds with a personalized model for each group and the ability to generalize knowledge in case the learners need it. In fact, the learner can choose to use their cluster model or train a generalizable solution locally if they have the labeled data.

Finally, the additional local machine learning algorithm lets the learners personalize the outputs of the models by allowing them to relabel the data according to their specifications.

6. Conclusions

In this paper, we presented MultiConfederated Learning a decentralized Federated Learning solution without a central aggregator designed for handling non-IID data. The groups formed during training were able to train a model that fit their data quickly. With more time and if possible, they can converge toward a single model. The key elements to achieve this are training multiple models, transfer-learning, well-thought selection rules and limiting weight divergence between updates. The results showed that the proposed approach worked effectively in two challenging non-IID scenarios. Moreover, a simple solution to use multiple models to obtain a generalizable solution was demonstrated to be viable.

For future work, we wish to explore how to train models to generate synthetic datasets to train new models locally. Additionally, we are interested in using synthetic datasets and their generators to perform knowledge distillation to create a global model locally. Finally, we think that our work sets a good path forward for a robust byzantine-resistant decentralized machine learning network.

ANNEXE II

CONTRATS INTELLIGENTS

1. Trainers.vy

```
# @version 0.3.4

from vyper.interfaces import ERC165
from vyper.interfaces import ERC721

implements: ERC165
implements: ERC721

############ ERC-165 #############
# @dev Static list of supported ERC165 interface ids
SUPPORTED_INTERFACES: constant(bytes4[4]) = [
    0x01ffc9a7,  # ERC165 interface ID of ERC165
    0x80ac58cd,  # ERC165 interface ID of ERC721
    0x5b5e139f,  # ERC165 interface ID of ERC721 Metadata Extension
    0x5604e225,  # ERC165 interface ID of ERC4494
]

############ ERC-721 #############

# Interface for the contract called by safeTransferFrom()
interface ERC721Receiver:
    def onERC721Received(
            operator: address,
            owner: address,
            tokenId: uint256,
            data: Bytes[1024]
        ) -> bytes4: view

# Interface for ERC721Metadata

```

```
31 interface ERC721Metadata:
32   def name() -> String[64]: view
33
34   def symbol() -> String[32]: view
35
36   def tokenURI(
37     _tokenId: uint256
38   ) -> String[128]: view
39
40 interface ERC721Enumerable:
41
42   def totalSupply() -> uint256: view
43
44   def tokenByIndex(
45     _index: uint256
46   ) -> uint256: view
47
48   def tokenOfOwnerByIndex(
49     _address: address,
50     _index: uint256
51   ) -> uint256: view
52
53
54 # @dev Emits when ownership of any NFT changes by any mechanism. This
      event emits when NFTs are
55 #      created (`from` == 0) and destroyed (`to` == 0). Exception:
      during contract creation, any
56 #      number of NFTs may be created and assigned without emitting
      Transfer. At the time of any
57 #      transfer, the approved address for that NFT (if any) is reset to
      none.
58 # @param owner Sender of NFT (if address is zero address it indicates
      token creation).
59 # @param receiver Receiver of NFT (if address is zero address it
      indicates token destruction).
```

```
60 # @param tokenId The NFT that got transfered.
61 event Transfer:
62     sender: indexed(address)
63     receiver: indexed(address)
64     tokenId: indexed(uint256)
65
66 # @dev This emits when the approved address for an NFT is changed or
       reaffirmed. The zero
67 #      address indicates there is no approved address. When a Transfer
       event emits, this also
68 #      indicates that the approved address for that NFT (if any) is
       reset to none.
69 # @param owner Owner of NFT.
70 # @param approved Address that we are approving.
71 # @param tokenId NFT which we are approving.
72 event Approval:
73     owner: indexed(address)
74     approved: indexed(address)
75     tokenId: indexed(uint256)
76
77 # @dev This emits when an operator is enabled or disabled for an owner.
       The operator can manage
78 #      all NFTs of the owner.
79 # @param owner Owner of NFT.
80 # @param operator Address to which we are setting operator rights.
81 # @param approved Status of operator rights(true if operator rights are
       given and false if
82 # revoked).
83 event ApprovalForAll:
84     owner: indexed(address)
85     operator: indexed(address)
86     approved: bool
87
88
89 event TrainerRegistered:
```

```
90      trainer: indexed(address)
91      trainerId: indexed(uint256)
92
93  owner: public(address)
94
95  totalSupply: public(uint256)
96
97  # @dev TokenID => owner
98  idToOwner: public(HashMap[uint256, address])
99
100 ownerToId: public(HashMap[address, uint256])
101
102 # @dev Mapping from owner address to mapping of operator addresses.
103 isApprovedForAll: public(HashMap[address, HashMap[address, bool]])
104
105 # @dev Mapping from NFT ID to approved address.
106 idToApprovals: public(HashMap[uint256, address])
107 ############ ERC-4494 ############
108
109 # @dev Mapping of TokenID to nonce values used for ERC4494 signature
        verification
110 nonces: public(HashMap[uint256, uint256])
111
112 DOMAIN_SEPARATOR: public(bytes32)
113
114 EIP712_DOMAIN_TYPEHASH: constant(bytes32) = keccak256(
115     "EIP712Domain(string name,string version,uint256 chainId,address
        verifyingContract)"
116 )
117 EIP712_DOMAIN_NAMEHASH: constant(bytes32) = keccak256("Owner NFT")
118 EIP712_DOMAIN_VERSIONHASH: constant(bytes32) = keccak256("1")
119
120
121 # ERC20 Token Metadata
122 NAME: constant(String[20]) = "trainer"
```

```
123 SYMBOL: constant(String[5]) = "TRAIN"
124 baseURI: public(String[100])
125
126 @external
127 def __init__():
128     """
129     @dev Contract constructor.
130     """
131     self.owner = msg.sender
132     self.baseURI = "ipfs://
        QmaZm1rAkt6kHTKTFX8GwEhtPMVMeAGJYMBvoAcJWTddwb"
133     # ERC712 domain separator for ERC4494
134     self.DOMAIN_SEPARATOR = keccak256(
135         _abi_encode(
136             EIP712_DOMAIN_TYPEHASH,
137             EIP712_DOMAIN_NAMEHASH,
138             EIP712_DOMAIN_VERSIONHASH,
139             chain.id,
140             self,
141         )
142     )
143
144 @external
145 def register() -> bool:
146     """
147     @notice Les entra neurs doivent s'enregistrer pour pouvoir
        participer et passer    la prochaine ronde
148     @dev Create a new Owner NFT
149     @return bool confirming that the minting occurred
150     """
151     assert not self._isRegistered(msg.sender)
152     self.totalSupply += 1
153     assert self.idToOwner[self.totalSupply] == empty(address)  # Sanity
        check
154
```

```
155     self.idToOwner[self.totalSupply] = msg.sender
156     self.ownerToId[msg.sender] = self.totalSupply
157     log TrainerRegistered(msg.sender, self.totalSupply)
158     log Transfer(empty(address), msg.sender, self.totalSupply)
159
160     return True
161
162 @view
163 @internal
164 def _isRegistered(trainer: address) -> bool:
165     return self.ownerToId[trainer] > 0
166
167 @view
168 @external
169 def isRegistered(trainer: address) -> bool:
170     return self._isRegistered(trainer)
171
172 @view
173 @external
174 def getTrainerId(trainer: address) -> uint256:
175     return self.ownerToId[trainer]
176
177 # ERC721 Metadata Extension
178 @pure
179 @external
180 def name() -> String[40]:
181     return NAME
182
183 @pure
184 @external
185 def symbol() -> String[5]:
186     return SYMBOL
187
188 @view
189 @external
```

```
190 def tokenURI(tokenId: uint256) -> String[179]:
191     return concat(self.baseURI, "/" , uint2str(tokenId))
192
193 @external
194 def setBaseURI(_baseURI: String[100]):
195     assert msg.sender == self.owner
196     self.baseURI = _baseURI
197
198 @external
199 def setDomainSeparator():
200     """
201     @dev Update the domain separator in case of a hardfork where chain
        ID changes
202     """
203     self.DOMAIN_SEPARATOR = keccak256(
204         _abi_encode(
205             EIP712_DOMAIN_TYPEHASH,
206             EIP712_DOMAIN_NAMEHASH,
207             EIP712_DOMAIN_VERSIONHASH,
208             chain.id,
209             self,
210         )
211     )
212
213 ############ ERC-165 #############
214
215 @pure
216 @external
217 def supportsInterface(interface_id: bytes4) -> bool:
218     """
219     @dev Interface identification is specified in ERC-165.
220     @param interface_id Id of the interface
221     """
222     return interface_id in SUPPORTED_INTERFACES
223
```

```
224
225 ##### ERC-721 VIEW FUNCTIONS #####
226
227 @view
228 @external
229 def ownerOf(tokenId: uint256) -> address:
230     """
231     @dev Returns the address of the owner of the NFT.
232          Throws if `tokenId` is not a valid NFT.
233     @param tokenId The identifier for an NFT.
234     """
235     owner: address = self.idToOwner[tokenId]
236     # Throws if `tokenId` is not a valid NFT
237     assert owner != empty(address)
238     return owner
239
240
241 @view
242 @external
243 def getApproved(tokenId: uint256) -> address:
244     """
245     @dev Get the approved address for a single NFT.
246          Throws if `tokenId` is not a valid NFT.
247     @param tokenId ID of the NFT to query the approval of.
248     """
249     # Throws if `tokenId` is not a valid NFT
250     assert self.idToOwner[tokenId] != empty(address)
251     return self.idToApprovals[tokenId]
252
253 @view
254 @external
255 def balanceOf(owner: address) -> uint256:
256     if self.ownerToId[owner] > 0:
257         return 1
258     return 0
```

```
259
260
261 ### TRANSFER FUNCTION HELPERS ###
262 ### Royalty integration under the ERC-2981: NFT Royalty Standard
263 @view
264 @external
265 def royaltyInfo(_tokenId: uint256, _salePrice: uint256) -> (address,
       uint256):
266     """
267     /// @notice Called with the sale price to determine how much royalty
268     //          is owed and to whom. Important; Not all marketplaces
       respect this, e.g. OpenSea
269     /// @param _tokenId - the NFT asset queried for royalty information
270     /// @param _salePrice - the sale price of the NFT asset specified by
        _tokenId
271     /// @return receiver - address of who should be sent the royalty
       payment
272     /// @return royaltyAmount - the royalty payment amount for
       _salePrice
273     """
274     return self.owner, 0
275
276 @view
277 @internal
278 def _isApprovedOrOwner(spender: address, tokenId: uint256) -> bool:
279     """
280     @dev Returns whether the given spender can transfer a given token ID
281     @param spender address of the spender to query
282     @param tokenId uint256 ID of the token to be transferred
283     @return bool whether the msg.sender is approved for the given token
       ID,
284         is an operator of the owner, or is the owner of the token
285     """
286     owner: address = self.idToOwner[tokenId]
287
```

```
288     if owner == spender:
289         return True
290
291     if spender == self.idToApprovals[tokenId]:
292         return True
293
294     if (self.isApprovedForAll[owner])[spender]:
295         return True
296
297     return False
298
299
300 @internal
301 def _transferFrom(owner: address, receiver: address, tokenId: uint256,
        sender: address):
302     """
303     @dev Execute transfer of a NFT.
304          Throws unless `msg.sender` is the current owner, an authorized
        operator, or the approved
305          address for this NFT. (NOTE: `msg.sender` not allowed in
        private function so pass `_sender`.)
306          address for thisassert self.idToOwner[tokenId] == owner NFT. (
        NOTE: `msg.sender` not allowed in private function so pass `_sender
        `.)
307          Throws if `receiver` is the zero address.
308          Throws if `owner` is not the current owner.
309          Throws if `tokenId` is not a valid NFT.
310
311     """
312     # Check requirements
313     assert self._isApprovedOrOwner(sender, tokenId)
314     assert receiver != empty(address)
315     assert owner != empty(address)
316     assert self.idToOwner[tokenId] == owner
317     assert not self._isRegistered(receiver)
```

```
318
319     # Reset approvals, if any
320     if self.idToApprovals[tokenId] != empty(address):
321         self.idToApprovals[tokenId] = empty(address)
322     # EIP-4494: increment nonce on transfer for safety
323     self.nonces[tokenId] += 1
324
325     # Change the owner
326     self.idToOwner[tokenId] = receiver
327     self.ownerToId[owner] = 0
328     self.ownerToId[receiver] = tokenId
329     # Log the transfer
330     log Transfer(owner, receiver, tokenId)
331
332
333 @external
334 def transferFrom(owner: address, receiver: address, tokenId: uint256):
335     """
336     @dev Throws unless `msg.sender` is the current owner, an authorized
      operator, or the approved
337          address for this NFT.
338          Throws if `owner` is not the current owner.
339          Throws if `receiver` is the zero address.
340          Throws if `tokenId` is not a valid NFT.
341     @notice The caller is responsible to confirm that `receiver` is
      capable of receiving NFTs or else
342             they maybe be permanently lost.
343     @param owner The current owner of the NFT.
344     @param receiver The new owner.
345     @param tokenId The NFT to transfer.
346     """
347     self._transferFrom(owner, receiver, tokenId, msg.sender)
348
349
350 @external
```

```
351 def safeTransferFrom(
352         owner: address,
353         receiver: address,
354         tokenId: uint256,
355         data: Bytes[1024]=b""
356     ):
357     """
358     @dev Transfers the ownership of an NFT from one address to another address.
359          Throws unless `msg.sender` is the current owner, an authorized operator, or the
360          approved address for this NFT.
361          Throws if `owner` is not the current owner.
362          Throws if `receiver` is the zero address.
363          Throws if `tokenId` is not a valid NFT.
364          If `receiver` is a smart contract, it calls `onERC721Received` on `receiver` and throws if
365          the return value is not `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`.
366          NOTE: bytes4 is represented by bytes32 with padding
367     @param owner The current owner of the NFT.
368     @param receiver The new owner.
369     @param tokenId The NFT to transfer.
370     @param data Additional data with no specified format, sent in call to `receiver`.
371     """
372     self._transferFrom(owner, receiver, tokenId, msg.sender)
373     if receiver.is_contract: # check if `receiver` is a contract address
374         returnValue: bytes4 = ERC721Receiver(receiver).onERC721Received(msg.sender, owner, tokenId, data)
375         # Throws if transfer destination is a contract which does not implement 'onERC721Received'
376         assert returnValue == method_id("onERC721Received(address,address,uint256,bytes)", output_type=bytes4)
377
```

```
378
379 @external
380 def approve(operator: address, tokenId: uint256):
381     """
382     @dev Set or reaffirm the approved address for an NFT. The zero
    address indicates there is no approved address.
383          Throws unless `msg.sender` is the current NFT owner, or an
    authorized operator of the current owner.
384          Throws if `tokenId` is not a valid NFT. (NOTE: This is not
    written the EIP)
385          Throws if `operator` is the current owner. (NOTE: This is not
    written the EIP)
386     @param operator Address to be approved for the given NFT ID.
387     @param tokenId ID of the token to be approved.
388     """
389     # Throws if `tokenId` is not a valid NFT
390     owner: address = self.idToOwner[tokenId]
391     assert owner != empty(address)
392
393     # Throws if `operator` is the current owner
394     assert operator != owner
395
396     # Throws if `msg.sender` is not the current owner, or is approved
    for all actions
397     assert owner == msg.sender or (self.isApprovedForAll[owner])[msg.
    sender]
398
399     self.idToApprovals[tokenId] = operator
400     log Approval(owner, operator, tokenId)
401 @external
402 def permit(spender: address, tokenId: uint256, deadline: uint256, sig:
    Bytes[65]) -> bool:
403     """
404     @dev Allow a 3rd party to approve a transfer via EIP-721 message
405         Raises if permit has expired
```

```
406         Raises if ‘tokenId‘ is unowned
407         Raises if permit is not signed by token owner
408         Raises if ‘nonce‘ is not the current expected value
409         Raises if ‘sig‘ is not a supported signature type
410     @param spender The approved spender of ‘tokenId‘ for the permit
411     @param tokenId The token that is being approved
412         NOTE: signer is checked against this token’s owner
413     @param deadline The time limit for which the message is valid for
414     @param sig The signature for the message, either in vrs or EIP-2098
        form
415     @return bool If the operation is successful
416     """
417     # Permit is still valid
418     assert block.timestamp <= deadline
419
420     # Ensure the token is owned by someone
421     owner: address = self.idToOwner[tokenId]
422     assert owner != empty(address)
423
424     # Nonce for given token (signer must ensure they use latest)
425     nonce: uint256 = self.nonces[tokenId]
426
427     # Compose EIP-712 message
428     message: bytes32 = keccak256(
429         _abi_encode(
430             0x1901,
431             self.DOMAIN_SEPARATOR,
432             keccak256(
433                 _abi_encode(
434                     keccak256(
435                         "Permit(address spender,uint256 tokenId,uint256
        nonce,uint256 deadline)"
436                     ),
437                     spender,
438                     tokenId,
```

```
                    nonce,
                    deadline,
                )
            )
        )
    )

    # Validate signature
    v: uint256 = 0
    r: uint256 = 0
    s: uint256 = 0

    if len(sig) == 65:
        # Normal encoded VRS signatures
        v = convert(slice(sig, 0, 1), uint256)
        r = convert(slice(sig, 1, 32), uint256)
        s = convert(slice(sig, 33, 32), uint256)

    elif len(sig) == 64:
        # EIP-2098 compact signatures
        r = convert(slice(sig, 0, 32), uint256)
        v = convert(slice(sig, 33, 1), uint256)
        s = convert(slice(sig, 34, 31), uint256)

    else:
        raise  # Other schemes not supported

    # Ensure owner signed permit
    assert ecrecover(message, v, r, s) == owner

    self.nonces[tokenId] = nonce + 1
    self.idToApprovals[tokenId] = spender

    return True

```

```
474 @external
475 def setApprovalForAll(operator: address, approved: bool):
476     """
477     @dev Enables or disables approval for a third party ("operator") to
        manage all of
478          ‘msg.sender‘’s assets. It also emits the ApprovalForAll event.
479     @notice This works even if sender doesn’t own any tokens at the time
        .
480     @param operator Address to add to the set of authorized operators.
481     @param approved True if the operators is approved, false to revoke
        approval.
482     """
483     self.isApprovedForAll[msg.sender][operator] = approved
484     log ApprovalForAll(msg.sender, operator, approved)
```

2. Updates.vy

```
1 # @version 0.3.4
2 from vyper.interfaces import ERC165
3 from vyper.interfaces import ERC721
4
5 from .Models import Models
6
7 implements: ERC165
8 implements: ERC721
9
10 ############ ERC-165 #############
11 # @dev Static list of supported ERC165 interface ids
12 SUPPORTED_INTERFACES: constant(bytes4[5]) = [
13     0x01ffc9a7,  # ERC165 interface ID of ERC165
14     0x80ac58cd,  # ERC165 interface ID of ERC721
15     0x5b5e139f,  # ERC165 interface ID of ERC721 Metadata Extension
16     0x5604e225,  # ERC165 interface ID of ERC4494
17     0x2a55205a,  # ERC165 interface ID of ERC2981
18 ]
```

```
19
20 ############ ERC-721 #############
21
22 # Interface for the contract called by safeTransferFrom()
23 interface ERC721Receiver:
24     def onERC721Received(
25             operator: address,
26             owner: address,
27             tokenId: uint256,
28             data: Bytes[1024]
29         ) -> bytes4: view
30
31 # Interface for ERC721Metadata
32
33 interface ERC721Metadata:
34   def name() -> String[64]: view
35
36   def symbol() -> String[32]: view
37
38   def tokenURI(
39     _tokenId: uint256
40   ) -> String[256]: view
41
42 interface ERC721Enumerable:
43
44   def totalSupply() -> uint256: view
45
46   def tokenByIndex(
47     _index: uint256
48   ) -> uint256: view
49
50   def tokenOfOwnerByIndex(
51     _address: address,
52     _index: uint256
53   ) -> uint256: view
```

```
54
55
56 # @dev Emits when ownership of any NFT changes by any mechanism. This event
       emits when NFTs are
57 #      created (`from` == 0) and destroyed (`to` == 0). Exception: during
      contract creation, any
58 #      number of NFTs may be created and assigned without emitting Transfer
      . At the time of any
59 #      transfer, the approved address for that NFT (if any) is reset to
      none.
60 # @param owner Sender of NFT (if address is zero address it indicates token
       creation).
61 # @param receiver Receiver of NFT (if address is zero address it indicates
      token destruction).
62 # @param tokenId The NFT that got transfered.
63 event Transfer:
64     sender: indexed(address)
65     receiver: indexed(address)
66     tokenId: indexed(uint256)
67
68 # @dev This emits when the approved address for an NFT is changed or
      reaffirmed. The zero
69 #      address indicates there is no approved address. When a Transfer
      event emits, this also
70 #      indicates that the approved address for that NFT (if any) is reset
      to none.
71 # @param owner Owner of NFT.
72 # @param approved Address that we are approving.
73 # @param tokenId NFT which we are approving.
74 event Approval:
75     owner: indexed(address)
76     approved: indexed(address)
77     tokenId: indexed(uint256)
78
```

```
79 # @dev This emits when an operator is enabled or disabled for an owner. The
       operator can manage
80 #       all NFTs of the owner.
81 # @param owner Owner of NFT.
82 # @param operator Address to which we are setting operator rights.
83 # @param approved Status of operator rights(true if operator rights are
      given and false if
84 # revoked).
85 event ApprovalForAll:
86     owner: indexed(address)
87     operator: indexed(address)
88     approved: bool
89
90 event UpdateProposed:
91     proposer: indexed(address)
92     parent: indexed(bytes32)
93     updateId: uint256
94     URI: String[256]
95
96 models: Models
97
98 totalSupply: public(uint256)
99
100 # @dev TokenID => owner
101 idToOwner: public(HashMap[uint256, address])
102
103 # @dev Mapping from owner address to count of their tokens.
104 balanceOf: public(HashMap[address, uint256])
105
106 # @dev Mapping from owner address to mapping of operator addresses.
107 isApprovedForAll: public(HashMap[address, HashMap[address, bool]])
108
109 # @dev Mapping from NFT ID to approved address.
110 idToApprovals: public(HashMap[uint256, address])
111
```

```
112 parentModel: public(HashMap[uint256, bytes32])
113
114 ############ ERC-4494 ############
115
116 # @dev Mapping of TokenID to nonce values used for ERC4494 signature
        verification
117 nonces: public(HashMap[uint256, uint256])
118
119 DOMAIN_SEPARATOR: public(bytes32)
120
121 EIP712_DOMAIN_TYPEHASH: constant(bytes32) = keccak256(
122     "EIP712Domain(string name,string version,uint256 chainId,address
        verifyingContract)"
123 )
124 EIP712_DOMAIN_NAMEHASH: constant(bytes32) = keccak256("Owner NFT")
125 EIP712_DOMAIN_VERSIONHASH: constant(bytes32) = keccak256("1")
126
127
128 # ERC20 Token Metadata
129 NAME: constant(String[6]) = "Models"
130 SYMBOL: constant(String[3]) = "MOD"
131
132 tokenURI: public(HashMap[uint256, String[256]])
133
134 @external
135 def __init__(models: address):
136     """
137     @dev Contract constructor.
138     """
139     self.models = Models(models)
140     # ERC712 domain separator for ERC4494
141     self.DOMAIN_SEPARATOR = keccak256(
142         _abi_encode(
143             EIP712_DOMAIN_TYPEHASH,
144             EIP712_DOMAIN_NAMEHASH,
```

```
145             EIP712_DOMAIN_VERSIONHASH,
146             chain.id,
147             self,
148         )
149     )
150
151 @external
152 def aggregate(updates: uint256[50], URI: String[128]):
153     """
154     @notice Lorsque les entra neurs ont d  cid  de leur combinaison, ils
155         appellent cette fonction pour signaler leur choix
156         La fonction s'assure que les MAJ sont tri s en ordre croissant
157         et ont le m me parent pour que la g n ration du prochain ID
158         soit correct
159     """
160     parentHash: bytes32 = self.parentModel[updates[0]]
161     for i in range(1, 50):
162         if updates[i] == empty(uint256):
163             break
164         assert updates[i-1] < updates[i], "Update list is not ordered"
165         assert parentHash == self.parentModel[updates[i]], "Updates do not
      have the same parent"
166     self.models.appendGeneration(parentHash, updates, URI)
167
168
169 @external
170 def propose(parentModel: bytes32, URI: String[256]) -> uint256:
171     """
172     @notice Les entra neurs appellent cette m thode pour enregistrer leur
       MAJ.
173     @param URI pointe vers l'espace de stockage du mod le
174     """
175     id: uint256 = self._mint(msg.sender, URI)
176     self.parentModel[id] = parentModel
177     log UpdateProposed(msg.sender, parentModel, id, URI)
```

```
178     return id
179
180
181 @internal
182 def _mint(receiver: address, URI: String[256]) -> uint256:
183     """
184     @dev Create a new Owner NFT
185     @return bool confirming that the minting occurred
186     """
187     self.totalSupply += 1
188     # Change count tracking, `totalSupply` represents id for `tokenId`
189     id: uint256 = self.totalSupply
190     # Throws if `totalSupply` count NFTs tracked by this contract is owned
      by someone
191     assert self.idToOwner[id] == empty(address), "already minted"
192     # Create new owner to allocate token
193     self.idToOwner[id] = receiver
194     self.tokenURI[id] = URI
195     # Update balance of minter
196     self.balanceOf[receiver] += 1
197     return id
198
199 # ERC721 Metadata Extension
200 @pure
201 @external
202 def name() -> String[40]:
203     return NAME
204
205 @pure
206 @external
207 def symbol() -> String[5]:
208     return SYMBOL
209
210 @external
211 def setDomainSeparator():
```

```
212     """
213     @dev Update the domain separator in case of a hardfork where chain ID
        changes
214     """
215     self.DOMAIN_SEPARATOR = keccak256(
216         _abi_encode(
217             EIP712_DOMAIN_TYPEHASH,
218             EIP712_DOMAIN_NAMEHASH,
219             EIP712_DOMAIN_VERSIONHASH,
220             chain.id,
221             self,
222         )
223     )
224
225 ############ ERC-165 #############
226
227 @pure
228 @external
229 def supportsInterface(interface_id: bytes4) -> bool:
230     """
231     @dev Interface identification is specified in ERC-165.
232     @param interface_id Id of the interface
233     """
234     return interface_id in SUPPORTED_INTERFACES
235
236
237 ##### ERC-721 VIEW FUNCTIONS #####
238
239 @view
240 @external
241 def ownerOf(tokenId: uint256) -> address:
242     """
243     @dev Returns the address of the owner of the NFT.
244          Throws if `tokenId` is not a valid NFT.
245     @param tokenId The identifier for an NFT.
```

```
246     """
247     owner: address = self.idToOwner[tokenId]
248     # Throws if `tokenId` is not a valid NFT
249     assert owner != empty(address)
250     return owner
251
252
253 @view
254 @external
255 def getApproved(tokenId: uint256) -> address:
256     """
257     @dev Get the approved address for a single NFT.
258          Throws if `tokenId` is not a valid NFT.
259     @param tokenId ID of the NFT to query the approval of.
260     """
261     # Throws if `tokenId` is not a valid NFT
262     assert self.idToOwner[tokenId] != empty(address)
263     return self.idToApprovals[tokenId]
264
265
266 ### TRANSFER FUNCTION HELPERS ###
267
268 @view
269 @internal
270 def _isApprovedOrOwner(spender: address, tokenId: uint256) -> bool:
271     """
272     @dev Returns whether the given spender can transfer a given token ID
273     @param spender address of the spender to query
274     @param tokenId uint256 ID of the token to be transferred
275     @return bool whether the msg.sender is approved for the given token ID,
276         is an operator of the owner, or is the owner of the token
277     """
278     owner: address = self.idToOwner[tokenId]
279
280     if owner == spender:
```

```
281         return True
282
283     if spender == self.idToApprovals[tokenId]:
284         return True
285
286     if (self.isApprovedForAll[owner])[spender]:
287         return True
288
289     return False
290
291
292 @internal
293 def _transferFrom(owner: address, receiver: address, tokenId: uint256,
        sender: address):
294     """
295     @dev Execute transfer of a NFT.
296          Throws unless `msg.sender` is the current owner, an authorized
        operator, or the approved
297          address for this NFT. (NOTE: `msg.sender` not allowed in private
        function so pass `_sender`.)
298          address for thisassert self.idToOwner[tokenId] == owner NFT. (NOTE
        : `msg.sender` not allowed in private function so pass `_sender`.)
299          Throws if `receiver` is the zero address.
300          Throws if `owner` is not the current owner.
301          Throws if `tokenId` is not a valid NFT.
302
303     """
304     # Check requirements
305     assert self._isApprovedOrOwner(sender, tokenId)
306     assert receiver != empty(address)
307     assert owner != empty(address)
308     assert self.idToOwner[tokenId] == owner
309
310     # Reset approvals, if any
311     if self.idToApprovals[tokenId] != empty(address):
```

```
312         self.idToApprovals[tokenId] = empty(address)
313
314     # EIP-4494: increment nonce on transfer for safety
315     self.nonces[tokenId] += 1
316     # Change the owner
317     self.idToOwner[tokenId] = receiver
318
319     # Change count tracking
320     self.balanceOf[owner] -= 1
321     # Add count of token to address
322     self.balanceOf[receiver] += 1
323
324     # Log the transfer
325     log Transfer(owner, receiver, tokenId)
326
327
328 @external
329 def transferFrom(owner: address, receiver: address, tokenId: uint256):
330     """
331     @dev Throws unless `msg.sender` is the current owner, an authorized
        operator, or the approved
332          address for this NFT.
333          Throws if `owner` is not the current owner.
334          Throws if `receiver` is the zero address.
335          Throws if `tokenId` is not a valid NFT.
336     @notice The caller is responsible to confirm that `receiver` is capable
         of receiving NFTs or else
337             they maybe be permanently lost.
338     @param owner The current owner of the NFT.
339     @param receiver The new owner.
340     @param tokenId The NFT to transfer.
341     """
342     self._transferFrom(owner, receiver, tokenId, msg.sender)
343
344
```

```
345 @external
346 def safeTransferFrom(
347         owner: address,
348         receiver: address,
349         tokenId: uint256,
350         data: Bytes[1024]=b""
351     ):
352     """
353     @dev Transfers the ownership of an NFT from one address to another address.
354          Throws unless `msg.sender` is the current owner, an authorized operator, or the
355          approved address for this NFT.
356          Throws if `owner` is not the current owner.
357          Throws if `receiver` is the zero address.
358          Throws if `tokenId` is not a valid NFT.
359          If `receiver` is a smart contract, it calls `onERC721Received` on `receiver` and throws if
360          the return value is not `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`.
361          NOTE: bytes4 is represented by bytes32 with padding
362     @param owner The current owner of the NFT.
363     @param receiver The new owner.
364     @param tokenId The NFT to transfer.
365     @param data Additional data with no specified format, sent in call to `receiver`.
366     """
367     self._transferFrom(owner, receiver, tokenId, msg.sender)
368     if receiver.is_contract: # check if `receiver` is a contract address
369         returnValue: bytes4 = ERC721Receiver(receiver).onERC721Received(msg.sender, owner, tokenId, data)
370         # Throws if transfer destination is a contract which does not implement 'onERC721Received'
371         assert returnValue == method_id("onERC721Received(address,address,uint256,bytes)", output_type=bytes4)
```

```
372
373
374 @external
375 def approve(operator: address, tokenId: uint256):
376     """
377     @dev Set or reaffirm the approved address for an NFT. The zero address
        indicates there is no approved address.
378          Throws unless `msg.sender` is the current NFT owner, or an
        authorized operator of the current owner.
379          Throws if `tokenId` is not a valid NFT. (NOTE: This is not written
         the EIP)
380          Throws if `operator` is the current owner. (NOTE: This is not
        written the EIP)
381     @param operator Address to be approved for the given NFT ID.
382     @param tokenId ID of the token to be approved.
383     """
384     # Throws if `tokenId` is not a valid NFT
385     owner: address = self.idToOwner[tokenId]
386     assert owner != empty(address)
387
388     # Throws if `operator` is the current owner
389     assert operator != owner
390
391     # Throws if `msg.sender` is not the current owner, or is approved for
        all actions
392     assert owner == msg.sender or (self.isApprovedForAll[owner])[msg.sender
        ]
393
394     self.idToApprovals[tokenId] = operator
395     log Approval(owner, operator, tokenId)
396
397 @external
398 def permit(spender: address, tokenId: uint256, deadline: uint256, sig:
        Bytes[65]) -> bool:
399     """
```

```
400     @dev Allow a 3rd party to approve a transfer via EIP-721 message
401         Raises if permit has expired
402         Raises if `tokenId` is unowned
403         Raises if permit is not signed by token owner
404         Raises if `nonce` is not the current expected value
405         Raises if `sig` is not a supported signature type
406     @param spender The approved spender of `tokenId` for the permit
407     @param tokenId The token that is being approved
408         NOTE: signer is checked against this token's owner
409     @param deadline The time limit for which the message is valid for
410     @param sig The signature for the message, either in vrs or EIP-2098
        form
411     @return bool If the operation is successful
412     """
413     # Permit is still valid
414     assert block.timestamp <= deadline
415
416     # Ensure the token is owned by someone
417     owner: address = self.idToOwner[tokenId]
418     assert owner != empty(address)
419
420     # Nonce for given token (signer must ensure they use latest)
421     nonce: uint256 = self.nonces[tokenId]
422
423     # Compose EIP-712 message
424     message: bytes32 = keccak256(
425         _abi_encode(
426             0x1901,
427             self.DOMAIN_SEPARATOR,
428             keccak256(
429                 _abi_encode(
430                     keccak256(
431                         "Permit(address spender,uint256 tokenId,uint256
        nonce,uint256 deadline)"
432                     ),
```

```
                    spender,
                    tokenId,
                    nonce,
                    deadline,
                )
            )
        )
    )

    # Validate signature
    v: uint256 = 0
    r: uint256 = 0
    s: uint256 = 0

    if len(sig) == 65:
        # Normal encoded VRS signatures
        v = convert(slice(sig, 0, 1), uint256)
        r = convert(slice(sig, 1, 32), uint256)
        s = convert(slice(sig, 33, 32), uint256)

    elif len(sig) == 64:
        # EIP-2098 compact signatures
        r = convert(slice(sig, 0, 32), uint256)
        v = convert(slice(sig, 33, 1), uint256)
        s = convert(slice(sig, 34, 31), uint256)

    else:
        raise  # Other schemes not supported

    # Ensure owner signed permit
    assert ecrecover(message, v, r, s) == owner

    self.nonces[tokenId] = nonce + 1
    self.idToApprovals[tokenId] = spender

```

```
468     return True
469
470 @external
471 def setApprovalForAll(operator: address, approved: bool):
472     """
473     @dev Enables or disables approval for a third party ("operator") to
        manage all of
474          'msg.sender''s assets. It also emits the ApprovalForAll event.
475     @notice This works even if sender doesn't own any tokens at the time.
476     @param operator Address to add to the set of authorized operators.
477     @param approved True if the operators is approved, false to revoke
        approval.
478     """
479     self.isApprovedForAll[msg.sender][operator] = approved
480     log ApprovalForAll(msg.sender, operator, approved)
```

3. Models.vy

```
1 # @version 0.3.4
2
3 from vyper.interfaces import ERC165
4 from vyper.interfaces import ERC721
5 from .Trainers import Trainers
6
7 implements: ERC165
8 implements: ERC721
9
10 ############ ERC-165 #############
11 # @dev Static list of supported ERC165 interface ids
12 SUPPORTED_INTERFACES: constant(bytes4[5]) = [
13     0x01ffc9a7,  # ERC165 interface ID of ERC165
14     0x80ac58cd,  # ERC165 interface ID of ERC721
15     0x5b5e139f,  # ERC165 interface ID of ERC721 Metadata Extension
16     0x5604e225,  # ERC165 interface ID of ERC4494
17     0x2a55205a,  # ERC165 interface ID of ERC2981
```

```
]

############ ERC-721 #############

# Interface for the contract called by safeTransferFrom()
interface ERC721Receiver:
    def onERC721Received(
            operator: address,
            owner: address,
            tokenId: uint256,
            data: Bytes[1024]
        ) -> bytes4: view

# Interface for ERC721Metadata

interface ERC721Metadata:
  def name() -> String[64]: view

  def symbol() -> String[32]: view

  def tokenURI(
    _tokenId: uint256
  ) -> String[128]: view

interface ERC721Enumerable:

  def totalSupply() -> uint256: view

  def tokenByIndex(
    _index: uint256
  ) -> uint256: view

  def tokenOfOwnerByIndex(
    _address: address,
    _index: uint256
```

```
53   ) -> uint256: view
54
55
56 # @dev Emits when ownership of any NFT changes by any mechanism. This event
       emits when NFTs are
57 #      created (`from` == 0) and destroyed (`to` == 0). Exception: during
      contract creation, any
58 #      number of NFTs may be created and assigned without emitting Transfer
      . At the time of any
59 #      transfer, the approved address for that NFT (if any) is reset to
      none.
60 # @param owner Sender of NFT (if address is zero address it indicates token
       creation).
61 # @param receiver Receiver of NFT (if address is zero address it indicates
      token destruction).
62 # @param tokenId The NFT that got transfered.
63 event Transfer:
64     sender: indexed(address)
65     receiver: indexed(address)
66     tokenId: indexed(uint256)
67
68 # @dev This emits when the approved address for an NFT is changed or
      reaffirmed. The zero
69 #      address indicates there is no approved address. When a Transfer
      event emits, this also
70 #      indicates that the approved address for that NFT (if any) is reset
      to none.
71 # @param owner Owner of NFT.
72 # @param approved Address that we are approving.
73 # @param tokenId NFT which we are approving.
74 event Approval:
75     owner: indexed(address)
76     approved: indexed(address)
77     tokenId: indexed(uint256)
78
```

```
79 # @dev This emits when an operator is enabled or disabled for an owner. The
       operator can manage
80 #      all NFTs of the owner.
81 # @param owner Owner of NFT.
82 # @param operator Address to which we are setting operator rights.
83 # @param approved Status of operator rights(true if operator rights are
      given and false if
84 # revoked).
85 event ApprovalForAll:
86     owner: indexed(address)
87     operator: indexed(address)
88     approved: bool
89
90 event ModelAppended:
91     roundId: indexed(uint256)
92     hash: indexed(bytes32)
93     parent: indexed(bytes32)
94     updates: uint256[50]
95     weight: uint256
96
97 event TrainerVoted:
98     trainerId: indexed(uint256)
99     trainer: indexed(address)
100    round: indexed(uint256)
101
102 event NewRound:
103    prevRound: indexed(uint256)
104    nextRound: indexed(uint256)
105
106 totalSupply: public(uint256)
107 round: public(uint256)
108 nextRoundVotes: public(uint256)
109 roundVoted: public(HashMap[uint256, bool])
110
111 # @dev TokenID => owner
```

```
112 idToOwner: public(HashMap[uint256, address])
113 hashToId: public(HashMap[bytes32, uint256])
114 idWeight: public(HashMap[uint256, uint256])
115 idToRound: public(HashMap[uint256, uint256])
116
117 # @dev Mapping from owner address to count of their tokens.
118 balanceOf: public(HashMap[address, uint256])
119
120 # @dev Mapping from owner address to mapping of operator addresses.
121 isApprovedForAll: public(HashMap[address, HashMap[address, bool]])
122
123 # @dev Mapping from NFT ID to approved address.
124 idToApprovals: public(HashMap[uint256, address])
125
126 ############ ERC-4494 ############
127
128 # @dev Mapping of TokenID to nonce values used for ERC4494 signature
       verification
129 nonces: public(HashMap[uint256, uint256])
130
131 DOMAIN_SEPARATOR: public(bytes32)
132
133 EIP712_DOMAIN_TYPEHASH: constant(bytes32) = keccak256(
134     "EIP712Domain(string name,string version,uint256 chainId,address
       verifyingContract)"
135 )
136 EIP712_DOMAIN_NAMEHASH: constant(bytes32) = keccak256("Owner NFT")
137 EIP712_DOMAIN_VERSIONHASH: constant(bytes32) = keccak256("1")
138
139
140 # ERC20 Token Metadata
141 NAME: constant(String[15]) = "Global Models"
142 SYMBOL: constant(String[3]) = "GLO"
143
144 tokenURI: public(HashMap[uint256, String[128]])
```

```
145 trainers: Trainers
146
147 @external
148 def __init__(trainers: address):
149     """
150     @dev Contract constructor.
151     """
152     self.trainers = Trainers(trainers)
153     # ERC712 domain separator for ERC4494
154     self.DOMAIN_SEPARATOR = keccak256(
155         _abi_encode(
156             EIP712_DOMAIN_TYPEHASH,
157             EIP712_DOMAIN_NAMEHASH,
158             EIP712_DOMAIN_VERSIONHASH,
159             chain.id,
160             self,
161         )
162     )
163     self.totalSupply = 1
164     self.hashToId[empty(bytes32)] = self.totalSupply
165     self.idWeight[self.totalSupply] += 1
166
167 @view
168 @external
169 def root() -> bytes32:
170     """
171     @dev Le premier noeud du graphe
172     """
173     return empty(bytes32)
174
175
176 @external
177 def nextRound():
178     """
```

```
179     @notice Indique que l'entraineur est pret    passer    une nouvelle
        ronde
180     @dev stock si l'entra neur a vot , la valeur bool flip    chaque
        ronde
181     """
182     trainerId: uint256 = self.trainers.getTrainerId(msg.sender)
183     assert trainerId > 0, "trainer not registered"
184     ifVoted: bool = self.round % 2 == 0
185     assert self.roundVoted[trainerId] != ifVoted, "Address has already
        voted"
186     self.roundVoted[trainerId] = not self.roundVoted[trainerId]
187     self.nextRoundVotes += 1
188     log TrainerVoted(trainerId, msg.sender, self.round)
189     if self.nextRoundVotes >= self.trainers.totalSupply():
190         self.round += 1
191         self.nextRoundVotes = 0
192         log NewRound(self.round - 1, self.round)
193
194 @external
195 def appendGeneration(parent: bytes32, updates: uint256[50], URI: String
        [128]) -> bytes32:
196     """
197     @notice Soumet une nouvelle combinaison de mise    jour
198     @returns l'ID du nouveau mod le
199     """
200     childHash: bytes32 = keccak256(_abi_encode(updates))
201     modelHash: bytes32 = self._append(parent, childHash, URI)
202
203     log ModelAppended(self.round, modelHash, parent, updates, self.idWeight
        [self.hashToId[modelHash]])
204     return modelHash
205
206 @internal
207 def _append(parent: bytes32, child: bytes32, URI: String[128]) -> bytes32:
208     """
```

```
209     @dev Ajoute le nouveau mod le au graph.
210     @dev Incremente le poid si la combinaison a  d  j     t   soumise
211     @return bytes32 the hash for the new NFT
212     """
213     assert self._hashExists(parent), "parent hash doesn't exists"
214     # Change count tracking, 'totalSupply' represents id for 'tokenId'
215     self.totalSupply += 1
216     # Throws if 'totalSupply' count NFTs tracked by this contract is owned
      by someone
217     hash: bytes32 = keccak256(_abi_encode(parent, child))
218     id: uint256 = self.totalSupply
219     if self._hashExists(hash):
220         id = self.hashToId[hash]
221     self.hashToId[hash] = id
222     self.tokenURI[id] = URI
223     self.idWeight[id] += 1
224     log Transfer(empty(address), empty(address), self.totalSupply)
225
226     return hash
227
228 @internal
229 @view
230 def _hashExists(hash: bytes32) -> bool:
231     return self.hashToId[hash] > 0
232
233 @view
234 @external
235 def modelExists(hash: bytes32) -> bool:
236     return self._hashExists(hash)
237
238 @external
239 def getBytes() -> bytes32:
240     return convert(10, bytes32)
241
242 # ERC721 Metadata Extension
```

```
243 @pure
244 @external
245 def name() -> String[40]:
246     return NAME
247
248 @pure
249 @external
250 def symbol() -> String[5]:
251     return SYMBOL
252
253 @external
254 def setDomainSeparator():
255     """
256     @dev Update the domain separator in case of a hardfork where chain ID
        changes
257     """
258     self.DOMAIN_SEPARATOR = keccak256(
259         _abi_encode(
260             EIP712_DOMAIN_TYPEHASH,
261             EIP712_DOMAIN_NAMEHASH,
262             EIP712_DOMAIN_VERSIONHASH,
263             chain.id,
264             self,
265         )
266     )
267
268 ############ ERC-165 #############
269
270 @pure
271 @external
272 def supportsInterface(interface_id: bytes4) -> bool:
273     """
274     @dev Interface identification is specified in ERC-165.
275     @param interface_id Id of the interface
276     """
```

```
277     return interface_id in SUPPORTED_INTERFACES
278
279
280 ##### ERC-721 VIEW FUNCTIONS #####
281
282 @view
283 @external
284 def ownerOf(tokenId: uint256) -> address:
285     """
286     @dev Returns the address of the owner of the NFT.
287          Throws if `tokenId` is not a valid NFT.
288     @param tokenId The identifier for an NFT.
289     """
290     owner: address = self.idToOwner[tokenId]
291     # Throws if `tokenId` is not a valid NFT
292     assert owner != empty(address)
293     return owner
294
295
296 @view
297 @external
298 def getApproved(tokenId: uint256) -> address:
299     """
300     @dev Get the approved address for a single NFT.
301          Throws if `tokenId` is not a valid NFT.
302     @param tokenId ID of the NFT to query the approval of.
303     """
304     # Throws if `tokenId` is not a valid NFT
305     assert self.idToOwner[tokenId] != empty(address)
306     return self.idToApprovals[tokenId]
307
308
309 ### TRANSFER FUNCTION HELPERS ###
310
311 @view
```

```
312 @internal
313 def _isApprovedOrOwner(spender: address, tokenId: uint256) -> bool:
314     """
315     @dev Returns whether the given spender can transfer a given token ID
316     @param spender address of the spender to query
317     @param tokenId uint256 ID of the token to be transferred
318     @return bool whether the msg.sender is approved for the given token ID,
319         is an operator of the owner, or is the owner of the token
320     """
321     owner: address = self.idToOwner[tokenId]
322
323     if owner == spender:
324         return True
325
326     if spender == self.idToApprovals[tokenId]:
327         return True
328
329     if (self.isApprovedForAll[owner])[spender]:
330         return True
331
332     return False
333
334
335 @internal
336 def _transferFrom(owner: address, receiver: address, tokenId: uint256,
      sender: address):
337     """
338     @dev Execute transfer of a NFT.
339          Throws unless `msg.sender` is the current owner, an authorized
      operator, or the approved
340          address for this NFT. (NOTE: `msg.sender` not allowed in private
      function so pass `_sender`.)
341          address for thisassert self.idToOwner[tokenId] == owner NFT. (NOTE
      : `msg.sender` not allowed in private function so pass `_sender`.)
342          Throws if `receiver` is the zero address.
```

```
343         Throws if ‘owner‘ is not the current owner.
344         Throws if ‘tokenId‘ is not a valid NFT.
345
346     """
347     # Check requirements
348     assert self._isApprovedOrOwner(sender, tokenId)
349     assert receiver != empty(address)
350     assert owner != empty(address)
351     assert self.idToOwner[tokenId] == owner
352
353     # Reset approvals, if any
354     if self.idToApprovals[tokenId] != empty(address):
355         self.idToApprovals[tokenId] = empty(address)
356
357     # EIP-4494: increment nonce on transfer for safety
358     self.nonces[tokenId] += 1
359     # Change the owner
360     self.idToOwner[tokenId] = receiver
361
362     # Change count tracking
363     self.balanceOf[owner] -= 1
364     # Add count of token to address
365     self.balanceOf[receiver] += 1
366
367     # Log the transfer
368     log Transfer(owner, receiver, tokenId)
369
370
371 @external
372 def transferFrom(owner: address, receiver: address, tokenId: uint256):
373     """
374     @dev Throws unless ‘msg.sender‘ is the current owner, an authorized
       operator, or the approved
375         address for this NFT.
376         Throws if ‘owner‘ is not the current owner.
```

```
377         Throws if `receiver` is the zero address.
378         Throws if `tokenId` is not a valid NFT.
379     @notice The caller is responsible to confirm that `receiver` is capable
        of receiving NFTs or else
380             they maybe be permanently lost.
381     @param owner The current owner of the NFT.
382     @param receiver The new owner.
383     @param tokenId The NFT to transfer.
384     """
385     self._transferFrom(owner, receiver, tokenId, msg.sender)
386
387
388 @external
389 def safeTransferFrom(
390         owner: address,
391         receiver: address,
392         tokenId: uint256,
393         data: Bytes[1024]=b""
394     ):
395     """
396     @dev Transfers the ownership of an NFT from one address to another
        address.
397         Throws unless `msg.sender` is the current owner, an authorized
        operator, or the
398         approved address for this NFT.
399         Throws if `owner` is not the current owner.
400         Throws if `receiver` is the zero address.
401         Throws if `tokenId` is not a valid NFT.
402         If `receiver` is a smart contract, it calls `onERC721Received` on
        `receiver` and throws if
403         the return value is not `bytes4(keccak256("onERC721Received(
        address,address,uint256,bytes)"))`.
404         NOTE: bytes4 is represented by bytes32 with padding
405     @param owner The current owner of the NFT.
406     @param receiver The new owner.
```

```
407     @param tokenId The NFT to transfer.
408     @param data Additional data with no specified format, sent in call to `
    receiver`.
409     """
410     self._transferFrom(owner, receiver, tokenId, msg.sender)
411     if receiver.is_contract: # check if `receiver` is a contract address
412         returnValue: bytes4 = ERC721Receiver(receiver).onERC721Received(msg
    .sender, owner, tokenId, data)
413         # Throws if transfer destination is a contract which does not
    implement 'onERC721Received'
414         assert returnValue == method_id("onERC721Received(address,address,
    uint256,bytes)", output_type=bytes4)
415
416
417 @external
418 def approve(operator: address, tokenId: uint256):
419     """
420     @dev Set or reaffirm the approved address for an NFT. The zero address
    indicates there is no approved address.
421          Throws unless `msg.sender` is the current NFT owner, or an
    authorized operator of the current owner.
422          Throws if `tokenId` is not a valid NFT. (NOTE: This is not written
     the EIP)
423          Throws if `operator` is the current owner. (NOTE: This is not
    written the EIP)
424     @param operator Address to be approved for the given NFT ID.
425     @param tokenId ID of the token to be approved.
426     """
427     # Throws if `tokenId` is not a valid NFT
428     owner: address = self.idToOwner[tokenId]
429     assert owner != empty(address)
430
431     # Throws if `operator` is the current owner
432     assert operator != owner
433
```

```
434     # Throws if `msg.sender` is not the current owner, or is approved for
        all actions
435     assert owner == msg.sender or (self.isApprovedForAll[owner])[msg.sender
        ]
436
437     self.idToApprovals[tokenId] = operator
438     log Approval(owner, operator, tokenId)
439
440 @external
441 def permit(spender: address, tokenId: uint256, deadline: uint256, sig:
        Bytes[65]) -> bool:
442     """
443     @dev Allow a 3rd party to approve a transfer via EIP-721 message
444         Raises if permit has expired
445         Raises if `tokenId` is unowned
446         Raises if permit is not signed by token owner
447         Raises if `nonce` is not the current expected value
448         Raises if `sig` is not a supported signature type
449     @param spender The approved spender of `tokenId` for the permit
450     @param tokenId The token that is being approved
451         NOTE: signer is checked against this token's owner
452     @param deadline The time limit for which the message is valid for
453     @param sig The signature for the message, either in vrs or EIP-2098
        form
454     @return bool If the operation is successful
455     """
456     # Permit is still valid
457     assert block.timestamp <= deadline
458
459     # Ensure the token is owned by someone
460     owner: address = self.idToOwner[tokenId]
461     assert owner != empty(address)
462
463     # Nonce for given token (signer must ensure they use latest)
464     nonce: uint256 = self.nonces[tokenId]
```

```
465
466     # Compose EIP-712 message
467     message: bytes32 = keccak256(
468         _abi_encode(
469             0x1901,
470             self.DOMAIN_SEPARATOR,
471             keccak256(
472                 _abi_encode(
473                     keccak256(
474                         "Permit(address spender,uint256 tokenId,uint256
    nonce,uint256 deadline)"
475                     ),
476                     spender,
477                     tokenId,
478                     nonce,
479                     deadline,
480                 )
481             )
482         )
483     )
484
485     # Validate signature
486     v: uint256 = 0
487     r: uint256 = 0
488     s: uint256 = 0
489
490     if len(sig) == 65:
491         # Normal encoded VRS signatures
492         v = convert(slice(sig, 0, 1), uint256)
493         r = convert(slice(sig, 1, 32), uint256)
494         s = convert(slice(sig, 33, 32), uint256)
495
496     elif len(sig) == 64:
497         # EIP-2098 compact signatures
498         r = convert(slice(sig, 0, 32), uint256)
```

```
499         v = convert(slice(sig, 33, 1), uint256)
500         s = convert(slice(sig, 34, 31), uint256)
501
502     else:
503         raise  # Other schemes not supported
504
505     # Ensure owner signed permit
506     assert ecrecover(message, v, r, s) == owner
507
508     self.nonces[tokenId] = nonce + 1
509     self.idToApprovals[tokenId] = spender
510
511     return True
512
513 @external
514 def setApprovalForAll(operator: address, approved: bool):
515     """
516     @dev Enables or disables approval for a third party ("operator") to
        manage all of
517          `msg.sender`'s assets. It also emits the ApprovalForAll event.
518     @notice This works even if sender doesn't own any tokens at the time.
519     @param operator Address to add to the set of authorized operators.
520     @param approved True if the operators is approved, false to revoke
        approval.
521     """
522     self.isApprovedForAll[msg.sender][operator] = approved
523     log ApprovalForAll(msg.sender, operator, approved)
```

BIBLIOGRAPHIE

Beilharz, J., Pfitzner, B., Schmid, R., Geppert, P., Arnrich, B. & Polze, A. (2021, Dec). Implicit Model Specialization through DAG-based Decentralized Federated Learning. *Proceedings of the 22nd International Middleware Conference*, pp. 310–322. doi : 10.1145/3464298.3493403. [6, 73]

Briggs, C., Fan, Z. & Andras, P. (2020, Jul). Federated learning with hierarchical clustering of local updates to improve training on non-IID data. *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–9. doi : 10.1109/IJCNN48605.2020.9207469. [6, 72]

Bro, R. & Smilde, A. K. (2014). Principal component analysis. *Anal. Methods*, 6(9), 2812–2831. doi : 10.1039/C3AY41907J. [86]

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. & Amodei, D. (2020). Language Models are Few-Shot Learners. (arXiv :2005.14165). Repéré à http://arxiv.org/abs/2005.14165. arXiv :2005.14165 [cs]. [-]

Chandran, P., Bhat, R., Chakravarthi, A. & Chandar, S. (2021). Weight Divergence Driven Divide-and-Conquer Approach for Optimal Federated Learning from non-IID Data. (arXiv :2106.14503). Repéré à http://arxiv.org/abs/2106.14503. arXiv :2106.14503 [cs]. [5, 39, 71]

Ghosh, A., Chung, J., Yin, D. & Ramchandran, K. (2021). An Efficient Framework for Clustered Federated Learning. (arXiv :2006.04088). doi : 10.48550/arXiv.2006.04088. arXiv :2006.04088 [cs, stat]. [6, 72, 80]

Guo, S., Zhang, T., Yu, H., Xie, X., Ma, L., Xiang, T. & Liu, Y. (2021). Byzantine-resilient Decentralized Stochastic Gradient Descent. *arXiv :2002.08569 [cs]*. Repéré à http://arxiv.org/abs/2002.08569. arXiv : 2002.08569. [2, 9, 76]

Handschuh, H. (2005). SHA Family (Secure Hash Algorithm). Dans van Tilborg, H. C. A. (Éd.), *Encyclopedia of Cryptography and Security* (pp. 565–567). Boston, MA : Springer US. doi : 10.1007/0-387-23483-7_388. [82]

Huang, C., Huang, J. & Liu, X. (2022). Cross-Silo Federated Learning : Challenges and Opportunities. (arXiv :2206.12949). Repéré à http://arxiv.org/abs/2206.12949. arXiv :2206.12949 [cs]. [9, 75]

Jeong, H., Son, H., Lee, S., Hyun, J. & Chung, T.-M. FedCC : Robust Federated Learning against Model Poisoning Attacks. [9, 75]

Karimireddy, S. P., Kale, S., Mohri, M., Reddi, S. J., Stich, S. U. & Suresh, A. T. (2021). SCAFFOLD : Stochastic Controlled Averaging for Federated Learning. (arXiv :1910.06378). doi : 10.48550/arXiv.1910.06378. arXiv :1910.06378 [cs, math, stat]. [6, 72]

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D. & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13), 3521–3526. doi : 10.1073/pnas.1611835114. arXiv :1612.00796 [cs, stat]. [5, 10, 72, 76]

Kopparapu, K. & Lin, E. (2020). FedFMC : Sequential Efficient Federated Learning on Non-iid Data. (arXiv :2006.10937). Repéré à http://arxiv.org/abs/2006.10937. arXiv :2006.10937 [cs, stat]. [6, 73, 80]

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4), 541–551. doi : 10.1162/neco.1989.1.4.541. [45, 88]

Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A. & Smith, V. (2020). Federated Optimization in Heterogeneous Networks. (arXiv :1812.06127). doi : 10.48550/arXiv.1812.06127. arXiv :1812.06127 [cs, stat]. [6, 45, 72, 88]

Li, Y., Chen, C., Liu, N., Huang, H., Zheng, Z. & Yan, Q. (2021). A Blockchain-Based Decentralized Federated Learning Framework with Committee Consensus. *IEEE Network*, 35(1), 234–241. doi : 10.1109/MNET.011.2000263. [2, 7, 70, 73]

Liu, B., Ding, M., Shaham, S., Rahayu, W., Farokhi, F. & Lin, Z. (2021). When Machine Learning Meets Privacy : A Survey and Outlook. *ACM Computing Surveys*, 54(2), 31 :1-31 :36. doi : 10.1145/3436755. [69]

Mansour, Y., Mohri, M., Ro, J. & Suresh, A. T. (2020). Three Approaches for Personalization with Applications to Federated Learning. (arXiv :2002.10619). Repéré à http://arxiv.org/abs/2002.10619. arXiv :2002.10619 [cs, stat]. [6, 72, 80]

Roy, A. G., Siddiqui, S., Pölsterl, S., Navab, N. & Wachinger, C. (2019). BrainTorrent : A Peer-to-Peer Environment for Decentralized Federated Learning. (arXiv :1905.06731). Repéré à http://arxiv.org/abs/1905.06731. arXiv :1905.06731 [cs, stat]. [7, 70, 73]

Shoham, N., Avidor, T., Keren, A., Israel, N., Benditkis, D., Mor Yosef, L. & Zeitak, I. (2019). *Overcoming Forgetting in Federated Learning on Non-IID Data*. [5, 72, 76]

Stone, P., Brooks, R., Brynjolfsson, E., Calo, R., Etzioni, O., Hager, G., Hirschberg, J., Kalyanakrishnan, S., Kamar, E., Kraus, S., Leyton-Brown, K., Parkes, D., Press, W., Saxenian, A., Shah, J., Tambe, M. & Teller, A. (2022). Artificial Intelligence and Life in 2030 : The One Hundred Year Study on Artificial Intelligence. (arXiv :2211.06318). doi : 10.48550/arXiv.2211.06318. arXiv :2211.06318 [cs]. [-]

Torrey, L. & Shavlik, J. (2010). Transfer Learning [chapter]. IGI Global. Repéré à https://www.igi-global.com/chapter/transfer-learning/www.igi-global.com/chapter/transfer-learning/36988. [77]

Wang, J., Liu, Q., Liang, H. & Joshi, G. Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization. [45, 88]

Wang, K., Mathews, R., Kiddon, C., Eichner, H., Beaufays, F. & Ramage, D. (2019). Federated Evaluation of On-device Personalization. (arXiv :1910.10252). doi : 10.48550/arXiv.1910.10252. arXiv :1910.10252 [cs, stat]. [10, 76]

Warnat-Herresthal, S., Schultze, H., Shastry, K. L., Manamohan, S., Mukherjee, S., Garg, V., Sarveswara, R., Händler, K., Pickkers, P., Aziz, N. A., Ktena, S., Tran, F., Bitzer, M., Ossowski, S., Casadei, N., Herr, C., Petersheim, D., Behrends, U., Kern, F., Fehlmann, T., Schommers, P., Lehmann, C., Augustin, M., Rybniker, J., Altmüller, J., Mishra, N., Bernardes, J. P., Krämer, B., Bonaguro, L., Schulte-Schrepping, J., De Domenico, E., Siever, C., Kraut, M., Desai, M., Monnet, B., Saridaki, M., Siegel, C. M., Drews, A., Nuesch-Germano, M., Theis, H., Heyckendorf, J., Schreiber, S., Kim-Hellmuth, S., Nattermann, J., Skowasch, D., Kurth, I., Keller, A., Bals, R., Nürnberg, P., Rieß, O., Rosenstiel, P., Netea, M. G., Theis, F., Mukherjee, S., Backes, M., Aschenbrenner, A. C., Ulas, T., Breteler, M. M. B., Giamarellos-Bourboulis, E. J., Kox, M., Becker, M., Cheran, S., Woodacre, M. S., Goh, E. L. & Schultze, J. L. (2021). Swarm Learning for decentralized and confidential clinical machine learning. *Nature*, 594(78627862), 265–270. doi : 10.1038/s41586-021-03583-3. [2, 7, 70, 73, 74]

Xu, R., Baracaldo, N. & Joshi, J. Privacy-Preserving Machine Learning : Methods, Challenges and Directions. [69]

Yang, Q., Liu, Y., Chen, T. & Tong, Y. (2019). Federated Machine Learning : Concept and Applications. *ACM Transactions on Intelligent Systems and Technology*, 10(2), 12 :1-12 :19. doi : 10.1145/3298981. [69]

Yang, T., Andrew, G., Eichner, H., Sun, H., Li, W., Kong, N., Ramage, D. & Beaufays, F. (2018). Applied Federated Learning : Improving Google Keyboard Query Suggestions. *arXiv :1812.02903 [cs, stat]*. Repéré à http://arxiv.org/abs/1812.02903. arXiv : 1812.02903. [1, 74]

Zhao, Y., Li, M., Lai, L., Suda, N., Civin, D. & Chandra, V. (2018). Federated Learning with Non-IID Data. doi : 10.48550/arXiv.1806.00582. arXiv :1806.00582 [cs, stat]. [2, 5, 9, 69, 71, 75, 84]

Zhu, H., Xu, J., Liu, S. & Jin, Y. (2021). Federated learning on non-IID data : A survey. *Neurocomputing*, 465, 371–390. doi : 10.1016/j.neucom.2021.07.098. [9, 75]