Neural network compression in the context of federated learning and edge devices

Lucas Grativol Ribeiro

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS DE LA LOIRE – IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Électronique*

Par

Lucas GRATIVOL RIBEIRO

Neural Network Compression in the Context of Federated Learning and Edge Devices

Thèse présentée et soutenue à IMT Atlantique, Brest, le 05/12/2024
Unité de recherche : Lab-STICC, CNRS UMR 6285
Thèse N° : 2024IMTA0444

Rapporteurs avant soutenance :

Frédéric PÉTROT	Professeur des Universités, Grenoble INP
Van-Tam NGUYEN	Professeur, Télécom Paris

Composition du Jury :

Président :	Jean-François NEZAN	Professeur des Universités, INSA Rennes
Examinateurs :	Stefan DUFFNER	Maître de conférences, INSA Lyon
	Frédéric PÉTROT	Professeur des Universités, Grenoble INP
	Van-Tam NGUYEN	Professeur, Télécom Paris
Dir. de thèse :	Matthieu ARZEL	Professeur, IMT Atlantique
Co-dir. de thèse :	Virginie FRESSE	Maître de conférences, Télécom Saint-Étienne
Encadrants de thèse :	Mathieu LÉONARDON	Maître de conférences, IMT Atlantique
	Guillaume MULLER	Maître de conférences, Mines Saint-Étienne

Invité(s) :

Erwan PIRIOU	System and embedded software engineer, CEA LIST

Contents

Acknowledgments

I want to express my sincere thanks to the members of my thesis defense jury: Associate Professor Stefan Duffener at INSA Lyon, Professor Frédéric Petrot at Grenoble INP, Professor Van-Tam Nguyen at Télécom Paris, and Dr. Erwan Piriou at CEA LIST for their review of my work, their interesting discussions, and their feedback. I also wish to thank Professor Jean-François Nezan at INSA Rennes for presiding over the jury.

Thank you, Matthieu Arzel, Mathieu Léonardon, Virginie Fresse, and Guillaume Muller, for believing in me and for offering me this great opportunity to do a thesis with you all. These past three years have been truly life-changing for me, not only because of the exciting and interesting academic discussions but also because of the ever-present good humor and warmth in our exchanges. Thank you all for always pushing me forward, for recognizing when I needed help, and for offering me such valuable advice. I have always looked up to you with great respect and admiration for the professionals and humans you are, and I hope I can reflect those qualities in my own path. I also hope that in the future, we will stay in touch and continue working together.

For the past three years, I have had the great pleasure of meeting, exchanging ideas with, and learning from so many extraordinary people, including those from the BRAIn team, the 2AI team at IMT Atlantique, Télécom Saint-Étienne, and the friends I made along the way. I want to thank you all for the good and the bad moments, the Smash sessions, the laughter, the toasts, the adventures, the help, and those cups of tea we shared. My heartfelt thanks to Albane, Alix, Amer, Aurelie, Axel, Aymane, Bastien, Brahim, Brainbzh, Camila, Cédric, Cynthia, Daniela, Erwan, Fernando, Gerard, Ghaith, Giulia, Hafsa, Hamoud, Hugo LB, Hugo T., Ilyass, Ismail, Jean-Noel, Jérémy, Jonathan, Karl, Lubin, Lucas B., Lucas F., Magali, Manon, Mariana, Marisa, Matheus, Max, Myriam, Nicolas, Pedro, Raphael B., Raphael L., Reda, Thibault, Timotée, Tom, Venkatesh, Vincent, Yassine, Yassir, and Yoann. I wish you all the very best and every success in your future.

Thanks to my family for all their unconditional support and love. Even though we live far apart and do not often speak, you have always been there for me, and for that I will always be grateful to you all.

Finally, a special thanks to An, who has made these last stressful years more fun, chaotic, and interesting. I could never have reached this point without you. Thank you for sharing my life with me.

I want to dedicate this thesis to my grandparents, who have always been my role models in life.

Résumé Long

Introduction

L'apprentissage automatique, et plus particulièrement l'apprentissage profond, est devenu un outil incontournable dans des domainees tels que l'ingénierie, la compréhension du langage, la médecine, et bien d'autres encore. En raison de la complexité des modèles d'apprentissage, qui peuvent comporter des milliers à des milliards de paramètres, l'entraînement de ces modèles nécessite une grande quantité de données prétraitées ainsi que des ressources de calcul importantes. Ce paradigme repose historiquement sur la centralisation des données, ce qui a permis le développement de réseaux de neurones et d'algorithmes d'apprentissage profond à grande échelle grâce à l'utilisation de cartes graphiques, de bibliothèques algébriques optimisées, et à des initiatives de «source ouverte».

Cependant, dans des contextes où les données sont sensibles, comme dans la santé ou les services personnalisés impliquant des interactions avec les données des utilisateurs, la collecte centralisée de ces données pose un problème de confidentialité. L'apprentissage fédéré a émergé comme une solution pour résoudre ce dilemme en déplaçant l'entraînement des modèles directement sur les appareils des utilisateurs, tout en ne partageant que les résultats de l'entraînement plutôt que les données brutes.

L'apprentissage fédéré s'avère particulièrement bénéfique dans des domainees comme la médecine, où les institutions peuvent collaborer sans partager des données sensibles (patients). Néanmoins, cette approche pose des défis techniques majeurs, en particulier pour les dispositifs embarqués, tels que les téléphones mobiles, les drones ou les appareils de l'Internet des Objets, dont les ressources matérielles sont limitées en termes de mémoire, de stockage et de puissance de traitement.

Cette thèse s'est concentrée sur l'amélioration de l'efficacité et de l'accessibilité de l'apprentissage fédéré, compte tenu des contraintes des dispositifs en termes de ressources matérielles, en explorant comment adapter des techniques de compression de modèles centralisées à des techniques fédérées. Au cours de nos études dans le domaine du fédéré, nous avons également été amenés à nous interroger sur la manière dont les systèmes sans capacités de communication pourraient également apprendre et tirer parti des données publiques disponibles.

Apprentissage Profond pour la Classification d'Images

Les algorithmes d'apprentissage profond sont définis par une architecture de réseaux de neurones entraînée sur un ensemble de données spécifique pour résoudre une tâche donnée. La tâche étudiée dans ce travail est la classification d'images, qui consiste à associer des images à des classes correspondantes. Dans le cadre centralisé tradition-

nel, l'architecture du modèle et l'ensemble de données sont disponibles sur un même nœud de calcul. Ce concept contraste avec le paradigme d'apprentissage distribué, où les données, les modèles et/ou les ressources de calcul sont répartis. Ce dernier cas correspond à l'apprentissage fédéré.

La classification d'images est l'une des tâches les plus courantes en apprentissage automatique. Dans cette tâche, une image $X \in \mathbb{R}^{C \times H \times S}$ est fournie en entrée à un modèle d'apprentissage profond, et la sortie $Y \in \mathbb{R}^n$ représente une distribution de probabilité sur les n classes possibles. L'image X est traitée par le modèle $\mathcal{F}$, et la sortie Y représente les logits, c'est-à-dire les niveaux de confiance bruts et non normalisés du modèle pour chaque classe. Ces logits sont ensuite passés à travers une fonction d'activation, telle que la softmax, pour les normaliser en une distribution de probabilité.La classe prédite correspond à l'index ayant la probabilité la plus élevée.

Pour évaluer les algorithmes d'apprentissage profond pour la classification d'images, plusieurs ensembles de données curés sont utilisés. Ces ensembles de données sont spécifiques à la tâche et conçus pour représenter la diversité possible des données du monde réel. Dans ce travail, nous avons examiné plusieurs ensembles de données publics bien connus utilisés comme références :

- CIFAR-10 : Un ensemble de 60 000 images couleur de 32x32 réparties en 10 classes.
- CIFAR-100 : Similaire à CIFAR-10 mais contenant 100 classes, ce qui en fait un ensemble de données plus complexe pour les modèles d'apprentissage profond.
- ImageNet-1K : Un ensemble de données à grande échelle utilisé pour les tâches de vision par ordinateur, contenant 1,2 million d'images d'entraînement réparties en 1 000 catégories. Cet ensemble est largement considéré comme une référence pour évaluer les modèles de classification d'images.

En plus de ces ensembles de données standard, d'autres ensembles dérivés de CIFAR-10 et ImageNet-1K, tels que CINIC-10 et MiniImageNet, ont également fait l'objet d'expérimentations.

En termes d'architectures de réseaux de neurones, un modèle est généralement composé d'un empilement de couches, chaque couche extrayant des caractéristiques de plus en plus abstraites des données d'entrée. Ce conception hiérarchique est motivé par le théorème d'approximation universelle, qui suggère que même les réseaux peu profonds peuvent approximer des fonctions continues. Dans ce travail, nous avons particulièrement considéré l'utilisation des réseaux de neurones convolutifs (CNNs), qui sont des modèles classiques pour la vision par ordinateur.

Les deux principaux modèles étudiés sont :

- VGG (*Visual Geometry Group*) : Une architecture connue pour sa simplicité et sa profondeur, utilisant une séquence de couches de convolution suivies de couches de regroupement (*pooling*).
- ResNet (*Residual Networks*) : Une architecture améliorée qui résout le problème du gradient évanescent en introduisant des connexions résiduelles, permettant aux gradients de se propager plus efficacement dans le réseau et permettant l'entraînement de réseaux plus profonds.

Apprentissage Fédéré

L'apprentissage fédéré est une approche d'apprentissage automatique distribué conçue pour répondre aux préoccupations relatives à la confidentialité des données en conser-

vant ces dernières localement sur les appareils. Dans cette approche, un serveur central coordonne le processus d'entraînement sans accéder directement aux données. Le serveur envoie le modèle à un sous-ensemble de clients participants, qui entraînent le modèle sur leurs données locales et renvoient les résultats de l'entraînement, comme les poids ou gradients du modèle, au serveur. Le serveur agrège ces mises à jour et les applique au modèle global, qui est ensuite redistribué aux clients pour un nouvel entraînement.

Quelques caractéristiques principales de l'apprentissage fédéré incluent :

- Orchestration des clients : Centralisée, avec un serveur central gérant le processus, ou décentralisée, avec des clients communiquant directement entre eux.
- Synchronisation : Synchrone, où tous les clients mettent à jour le modèle global simultanément, ou asynchrone, où les mises à jour sont appliquées au fur et à mesure de leur arrivée.
- Type de client : Inter-appareils (*cross-device*), avec de nombreux appareils peu puissants et peu fiables, ou inter-silos (*cross-silo*), avec quelques nœuds informatiques puissants et fiables provenant d'organisations comme des universités ou des entreprises privées.
- Distribution des données : IID (Indépendamment et Identiquement Distribuées) ou non-IID, où les distributions de données varient significativement entre les clients.

La Boucle d'Entraînement

Une boucle d'entraînement typique en apprentissage fédéré comprend plusieurs étapes clés :

1. Échantillonnage des clients : Le serveur sélectionne un sous-ensemble de clients pour participer à l'entraînement.
2. Téléchargement du modèle : Les clients reçoivent le modèle global le plus récent.
3. Entraînement local : Les clients entraînent le modèle reçu sur leurs données locales.
4. Téléversement de la mise à jour du modèle : Les clients renvoient le modèle mis à jour au serveur.
5. Agrégation des connaissances : Le serveur agrège les mises à jour des clients et affine le modèle global.

Federated Averaging (FedAvg) [117] est l'algorithme le plus utilisé pour agréger les clients en apprentissage fédéré, où le serveur effectue une somme pondérée des poids des clients et calcule la moyenne pour créer un modèle global. FedAvg opère en minimisant la somme pondérée des pertes locales sur les clients. Cette approche est efficace dans des environnements IID, mais des défis surviennent dans des scénarios non-IID, où la distribution des données entre clients peut être significativement différente.

Défis Ouverts en Apprentissage Fédéré

En raison du choix de conception où les données ne quittent jamais les clients, l'apprentissage fédéré présente plusieurs défis ouverts, notamment :

- Confidentialité : Malgré la nature locale des données, les mises à jour des modèles peuvent encore révéler des informations sensibles, conduisant à des violations de données au niveau du serveur. Des techniques comme le chiffrement homomorphe sont explorées pour atténuer ce risque.
- Optimisation Distribuée : Les données non-IID entre les clients complexifient le processus d'entraînement, avec des stratégies comme FedAvg avec momentum [80] et FedProx [107] tentant de résoudre ces défis.
- Sécurité : Les données des clients et les mises à jour des modèles sont vulnérables à des attaques telles que l'empoisonnement des données et des modèles. Des techniques d'agrégation robuste et de confidentialité différentielle sont des domainees clés de recherche pour renforcer la sécurité.
- Équité : Garantir que le modèle fonctionne bien pour tous les clients et que chaque client contribue équitablement au modèle global reste un défi, en particulier lorsque les clients ont des quantités de données, des disponibilités, des contributions et des ressources matérielles différentes.

En résumé, l'apprentissage fédéré offre une solution prometteuse aux préoccupations de confidentialité en apprentissage automatique, mais introduit de nouveaux défis, notamment en matière de distribution des données, de sécurité et d'optimisation. Des recherches supplémentaires sont nécessaires pour relever ces défis et améliorer la praticité de l'apprentissage fédéré dans des applications réelles.

Compression dans l'Apprentissage Profond

La surparamétrisation est une caractéristique déterminante des modèles modernes d'apprentissage profond. Bien que cela permette aux modèles de résoudre des problèmes de plus en plus complexes, cela engendre également une augmentation du stockage, des exigences computationnelles, et une inefficacité de leur exécution. Les méthodes de compression répondent à ces défis en réduisant la taille des modèles sans compromettre significativement leur performance/précision. Ces méthodes sont non seulement cruciales pour réduire la surparamétrisation, mais également pour répondre aux contraintes physiques et computationnelles inhérentes au déploiement de modèles à grande échelle.

La quantification et l'élagage sont deux techniques de compression majeures pour relever ces contraintes. La quantification réduit la précision de la représentation des données, simplifiant ainsi les calculs et diminuant les besoins en stockage. L'élagage, en revanche, supprime les paramètres ou éléments redondants, aboutissant à des modèles plus petits et plus efficaces. Ensemble, ces méthodes jouent un rôle essentiel dans l'optimisation de l'apprentissage profond pour des environnements contraints.

Méthodes de Quantification

La quantification simplifie la représentation des données en réduisant le nombre d'états possibles qu'elles peuvent prendre. Bien que cela entraîne souvent une perte d'information, le compromis réside dans une réduction significative de l'utilisation de la mémoire, de la consommation d'énergie et de la latence. La quantification est particulièrement importante pour les opérations gourmandes en ressources, comme les multiplications de matrices dans les couches d'apprentissage profond. En adoptant des formats de données plus petits et plus efficaces, la complexité computationnelle globale est réduite.

Lorsqu'on se réfère au processus de quantification dans le contexte des réseaux de neurones, en particulier pour les conversions entre les formats en virgule flottante et les entiers, la quantification comporte deux types principaux : la Quantification Après Entraînement (QAE) et la Quantification Sensible à l'Entraînement (QSE). QAE est une méthode directe où la quantification est appliquée à un modèle pré-entraîné, souvent avec une phase de calibration. Elle est plus rapide, mais entraîne généralement une plus grande dégradation de précision que le second type. En revanche, la QSE intègre le processus de quantification dans l'entraînement, permettant au modèle d'apprendre et de compenser les erreurs de quantification. Cela produit des modèles performants avec des précisions réduites, mais au prix d'un effort computationnel accru pendant l'entraînement.

Méthodes d'Élagage

L'élagage est une méthode permettant de réduire le nombre de paramètres ou d'éléments dans un modèle. En identifiant et supprimant les parties qui contribuent peu à la performance du modèle, l'élagage conduit à des modèles plus petits et plus efficaces. Cela est particulièrement utile pour le déploiement de modèles sur des matériels aux ressources computationnelles limitées.

Il existe deux principaux types d'élagage : non structuré et structuré. L'élagage non structuré consiste à mettre des poids (paramètres) individuels à zéro, augmentant ainsi la sparsité au sein du modèle. Bien que cette approche puisse réduire considérablement le nombre de paramètres, elle nécessite du matériel ou des techniques logicielles spécialisés pour exploiter efficacement cette parcimonie. L'élagage structuré, en revanche, supprime des structures entières, telles que des couches ou des filtres de convolution, réduisant potentiellement la complexité computationnelle.

Défis de Communication et de Calcul

L'apprentissage fédéré introduit des défis uniques en raison de sa nature décentralisée. En plus des défis ouverts déjà mentionnés, deux défis fondamentaux abordés dans cette thèse sont les défis de communication et de calcul. Les clients doivent entraîner des modèles localement et communiquer les mises à jour à un serveur central. Ce processus engendre des coûts de communication significatifs, en particulier lorsqu'il s'agit de grands modèles ou d'appareils à ressources limitées. En outre, les exigences computationnelles pour l'entraînement et l'inférence peuvent être prohibitivement élevées pour des appareils aux capacités limitées, comme les systèmes embarqués.

Les techniques de compression, telles que la quantification et l'élagage, offrent des solutions à ces défis. En réduisant la taille et la complexité des modèles, ces méthodes diminuent les frais de communication et les exigences computationnelles. Cela permet à l'apprentissage fédéré d'être plus accessible et efficace, en particulier dans des scénarios impliquant des appareils embarqués, qui fonctionnent souvent sous des contraintes strictes de ressources.

Quantification dans l'Apprentissage Fédéré

Dans l'apprentissage fédéré, la quantification a été adaptée pour relever à la fois les limitations de communication et de calcul. Contrairement à l'apprentissage profond centralisé, les environnements fédérés manquent souvent d'un jeu de données de calibration centralisé, ce qui rend la quantification plus complexe. Des techniques comme

la quantification stochastique et les mécanismes de rétroaction d'erreur ont été développées pour atténuer ces problèmes.

Par exemple, FedPAQ [141] applique la quantification stochastique aux mises à jour des modèles, réduisant ainsi le nombre de bits transmis pendant les rondes de communication. De même, BHFL [175] permet aux clients d'entraîner des modèles en utilisant des formats entiers, avec une déquantification côté serveur pour assurer la cohérence entre les modèles. Ces approches mettent en lumière l'adaptabilité des méthodes de quantification dans les contextes d'apprentissage fédéré.

Élagage dans l'Apprentissage Fédéré

L'élagage a également été adapté pour l'apprentissage fédéré, avec un accent sur la réduction des coûts de communication et de calcul. Des techniques comme PruneFL [88] et FedDST [18] permettent aux clients d'entraîner des modèles élagués, plus petits et plus efficaces. PruneFL détermine dynamiquement le taux d'élagage optimal pour chaque client, tandis que FedDST utilise un entraînement parcimonieux par couche pour minimiser les exigences computationnelles. Ces méthodes démontrent le potentiel de l'élagage pour relever les défis uniques de l'apprentissage fédéré.

Élagage par Magnitude pour une Compression Double-Sens

En nous appuyant sur le flot standard de l'apprentissage fédéré, nous avons intégré des techniques d'élagage pour augmenter la parcimonie des communications entre le serveur et les clients. Plus précisément, un élagage non structuré basé sur la magnitude des poids est appliqué aux niveaux du serveur et des clients avant la transmission des messages. Ce processus consiste à élaguer les poids globaux en fonction de leurs valeurs absolues et à fixer à zéro les $\theta\%$ des poids les plus faibles. En garantissant une parcimonie cohérente des deux côtés, cette méthode optimise l'efficacité des communications dans le processus d'apprentissage fédéré.

Des méthodes de codage entropique, telles que le codage de Huffman, sont utilisées pour compresser davantage les messages. Cette technique attribue des codes plus courts aux éléments les plus fréquents, ce qui réduit la taille des messages sans nécessiter de techniques dépendantes des données. Par conséquent, le serveur et les clients peuvent indépendamment compresser et décompresser les messages, évitant ainsi les surcoûts liés à une compression unilatérale.

Pour évaluer l'efficacité de notre méthode d'élagage, des expériences ont été menées sur les ensembles de données CIFAR-10 et CIFAR-100 en utilisant un modèle ResNet-12. Les taux d'élagage ont été variés pour évaluer les compromis entre la compression et la précision du modèle. Les résultats ont montré que, bien que l'élagage réduise considérablement les coûts de communication, l'impact sur la précision dépendait du jeu de données.

Une comparaison avec la méthode ZeroFL [136] a révélé que notre approche conservait une précision plus élevée pour des niveaux de compression équivalents. Par exemple, pour des tailles de message similaires, notre méthode a montré moins de dégradation de la précision par rapport à la méthode de référence. Cela souligne la robustesse de notre technique pour compenser la sparsité introduite.

FLoCoRA

Suite à notre première étude sur la sparsité dans l'apprentissage fédéré, nous avons exploré des techniques alternatives de réduction de communication, aboutissant à notre travail FLoCoRA [61](*Federated Learning Compression with Low-Rank Adaptation*). L'Adaptation de Faible Rang (*Low-Rank Adaptation*, LoRA), méthode initialement développée pour le fine-tuning efficace de modèles à grande échelle, a été adaptée dans FLoCoRA pour permettre un compromis entre le nombre de paramètres à communiquer et la dégradation de la précision. Au lieu d'entraîner tous les paramètres du modèle, LoRA introduit des matrices adaptatrices parallèles légères A et B, considérées comme des versions de faible rang de la matrice de poids d'une couche spécifique. L'idée est de geler la matrice de poids originale, qui n'est pas entraînée, et de n'entraîner que les adaptateurs LoRA. Après les phases d'entraînement, les adaptateurs sont intégrés au modèle original, représentant une mise à jour de faible rang du modèle. Cette approche réduit considérablement le nombre de paramètres à échanger lors de l'apprentissage fédéré, diminuant ainsi les coûts de communication.

Dans le cadre de FLoCoRA, les paramètres du modèle original restent figés, et seuls les adaptateurs LoRA sont entraînés et échangés entre les clients et le serveur. Cette approche assure une compatibilité avec les méthodes d'optimisation existantes en apprentissage fédéré et améliore la flexibilité.

L'efficacité de FLoCoRA a été évaluée sur plusieurs ensembles de données, y compris CIFAR-10, CIFAR-100 et CINIC-10, en utilisant diverses configurations de rang et de facteurs d'échelle. Les résultats ont montré des réductions substantielles des coûts de communication, avec des tailles de message réduites jusqu'à $63,9\times$ lorsqu'elles sont combinées avec une quantification affine. Malgré ces réductions, la précision est restée compétitive, avec des baisses limitées à 4% dans les scénarios de compression les plus extrêmes. Ces résultats soulignent le potentiel de FLoCoRA en tant que référence robuste pour un apprentissage fédéré efficace en termes de communication.

Compression vs Modèles Plus Petits

Nos recherches sur la réduction des coûts de communication ont soulevé la question de savoir si les techniques de compression ou des modèles plus petits sont plus efficaces pour réduire ces coûts. En comparant des modèles compressés à des architectures intrinsèquement plus petites, nous avons constaté que les architectures plus optimisées surpassent généralement les modèles surparamétrés. Par exemple, pour la tâche de classification d'images CIFAR-10, un ResNet-20 surpasse un ResNet-18, même avec plus de 10 fois moins de paramètres. Ce comportement est principalement dû au fait que l'architecture ResNet-20 a été optimisée pour la complexité de la tâche CIFAR-10, soulignant l'importance d'adapter les architectures aux tâches. Avec le choix correct d'architectures, nous avons constaté que la compression devient encore plus pertinente. Nous avons appliqué FLoCoRA aux ResNet-20 et ResNet-18 précédents afin d'obtenir la même taille de message, et le ResNet-20 compressé a présenté moins de dégradation de précision que le ResNet-18 compressé.

Modèles Pré-Entraînés pour l'Apprentissage Fédéré

Enfin, dans le cadre de notre étude sur la réduction de communication, nous avons examiné l'application des modèles pré-entraînés à l'apprentissage fédéré. L'utilisation de modèles pré-entraînés offre des avantages significatifs, comme le démontrent les études

récentes dans le domainee du l'apprentissage fédéré. Les modèles pré-entraînés accélèrent non seulement la convergence, mais atténuent également les défis posés par les distributions de données non-IID. En exploitant des poids pré-entraînés comme bases fixes, les coûts computationnels et de communication sont considérablement réduits, permettant aux clients ayant des ressources limitées d'atteindre des performances compétitives.

Des expériences comparant l'apprentissage fédéré avec et sans modèles pré-entraînés ont montré une convergence plus rapide et une précision améliorée pour la configuration pré-entraînée. Cela souligne le potentiel d'intégrer des modèles pré-entraînés dans les cadres d'apprentissage fédéré, en particulier pour des applications personnalisées.

Apprentissage Embarqué avec peu d'exemples

Les systèmes embarqués sont souvent confrontés à des contraintes matérielles importantes et à une disponibilité limitée des données. Une manière de relever ces défis réside dans l'apprentissage fédéré, comme discuté précédemment, ainsi que dans l'utilisation de modèles pré-entraînés. L'apprentissage fédéré permet aux modèles de s'entraîner de manière collaborative sur plusieurs appareils, exploitant ainsi des classes de données inaccessibles aux clients individuels et améliorant les performances globales. Cependant, dans des scénarios où l'apprentissage fédéré est irréalisable — tels que les dispositifs avec des capacités de communication limitées ou des préoccupations élevées en matière de confidentialité — les modèles pré-entraînés peuvent combler l'écart de performance.

Dans ce contexte, l'apprentissage avec peu d'exemples (*Few-Shot Learning*, FSL) constitue une alternative convaincante au réentraînement répété de modèles pré-entraînés, en particulier pour des scénarios où la disponibilité des données est minimale. Il permet aux modèles de s'adapter efficacement à de nouvelles tâches avec seulement quelques exemples. Un avantage notable du FSL est sa capacité à classer de nouvelles classes sans s'appuyer sur des techniques basées sur la descente de gradient stochastique, ce qui réduit considérablement les besoins en mémoire et en calcul. Pour les systèmes embarqués, cela est particulièrement bénéfique.

Lorsqu'il s'agit de plateformes matérielles pour déployer ces algorithmes, des options comme les GPU embarqués (par exemple, les NVIDIA Jetson) et les FPGA offrent des compromis intéressants. Les GPU facilitent le déploiement avec une faible consommation d'énergie, tandis que les FPGA permettent des latences plus faibles et une flexibilité accrue, essentielles pour les systèmes en temps réel.

Dans ce travail, nous avons développé une chaîne de mise en œuvre permettant de déployer un algorithme FSL sur une plateforme du type système sur une puce (*System-on-Chip*, SoC) composée d'un CPU et d'un FPGA.

Apprentissage avec Peu d'Exemples - EASY

L'apprentissage avec peu d'exemples répond au défi de classer des exemples appartenant à des classes inconnues en utilisant un nombre limité d'exemples étiquetés. L'apprentissage profond traditionnel excelle avec de grands ensembles de données, mais le FSL repose sur des routines d'entraînement spécifiques qui privilégient la généralisation plutôt que l'ajustement aux exemples individuels. La force principale du FSL réside dans l'utilisation de*backbones* entraînés comme extracteurs de caractéristiques universels. Ces *backbones* garantissent que les caractéristiques des classes inconnues sont bien

séparées dans l'espace des caractéristiques, facilitant ainsi la classification par comparaison.

Le cadre FSL utilisé dans ce travail commence par un entraînement générique sur un ensemble de données de base, où le *backbone* est entraîné. Des ensembles de validation, contenant des classes distinctes, sont utilisés pour évaluer la généralisation. Une fois entraîné, le *backbone* est figé et sert d'extracteur de caractéristiques universel. Les performances du modèle sont ensuite évaluées sur un nouvel ensemble de données comprenant un nombre défini de classes (*ways*) et d'exemples étiquetés (*shots*). La classification repose sur des comparaisons des moyennes de classes les plus proches (*Nearest Class Mean*, NCM) dans l'espace des caractéristiques.

Nous avons adopté l'algorithme EASY [15] de FSL, qui nous a permis d'entraîner des *backbones* robustes et polyvalents. Il utilise une architecture en forme de Y combinant une perte de classification supervisée avec une perte auto-supervisée. La perte auto-supervisée implique la reconnaissance des rotations appliquées aux échantillons d'entrée, favorisant l'extraction de caractéristiques significatives. De plus, la régularisation par mélange de variétés (*manifold mixup*) interpole les vecteurs de caractéristiques dans l'espace latent, favorisant des représentations généralisées.

Le processus d'entraînement inclut des ajustements dynamiques du taux d'apprentissage avec des redémarrages progressifs, garantissant une convergence fluide et réduisant le surapprentissage. Un ensemble de *backbones*, chacun initialisé avec des graines aléatoires différentes, est entraîné indépendamment pour créer des représentations de caractéristiques diversifiées et enrichies. Cette approche améliore les performances sans augmenter significativement la complexité du modèle.

Une Plateforme Reconfigurable

Les plateformes SoC intègrent des éléments de traitement traditionnels, tels que des CPU généralistes, avec une matrice FPGA programmable sur une seule puce. Cette combinaison permet aux CPU de gérer les tâches définies par logiciel tandis que la matrice FPGA accélère les fonctions computationnellement intensives comme le traitement de données en temps réel. Des exemples notables incluent la famille AMD-Xilinx Zynq, qui combine des CPU basés sur ARM avec des ressources FPGA, ce qui en fait une solution idéale pour les applications d'apprentissage profond.

La carte de développement PYNQ-Z1, utilisée dans ce travail, est dotée d'une puce Zynq XC7Z020-1CLG400C avec un CPU Cortex-A9 à double cœur, 512 Mo de mémoire et une matrice FPGA. Cette carte sert de plateforme d'entrée polyvalente pour les applications embarquées.

Le déploiement de modèles d'apprentissage profond sur FPGA implique la conversion de descriptions de haut niveau, telles que celles au format ONNX, en code matériel compatible avec le FPGA. Le framework Tensil [157], choisi pour ce travail, utilise une architecture de type tableau systolique optimisée pour les multiplications matricielles et d'autres opérations comme les activations et le pooling. L'adaptabilité et la compatibilité du framework avec la carte FPGA cible en font un choix adapté.

PEFSL : Une Chaîne Open-Source pour l'apprentissage embarqué avec peu d'exemples

La conception des architectures de *backbones* pour le FSL sur FPGA nécessite une optimisation pour équilibrer la précision et le coût computationnel. Nous avons choisi

de travailler avec des architectures peu profondes comme ResNet-9 et ResNet-12, qui offrent une complexité réduite tout en maintenant des performances compétitives. Nous avons exploré l'espace de conception des hyperparamètres des modèles tels que la profondeur du réseau, la taille des images d'entraînement et les cartes de caractéristiques pour déterminer le meilleur compromis entre les performances du système final (précision) et les exigences en ressources.

L'entraînement du *backbone* a été réalisé en utilisant le jeu de données MiniImageNet dans une configuration 5-*way*, 1-*shot*. Les classes diversifiées de ce jeu de données permettent une excellente généralisation pour de nouvelles tâches. Les résultats indiquent que ResNet-9 atteint une précision compétitive tout en maintenant une faible latence, nécessaire pour les applications en temps réel.

La chaîne PEFSL intègre entraînement, compilation et déploiement dans un cadre cohérent. Elle simplifie l'exploration des hyperparamètres et optimise les configurations des modèles pour les tâches FSL. ResNet-9, fonctionnant à 125 MHz, atteint une latence de 30 ms, démontrant l'efficacité de la chaîne dans des scénarios réels.

Démonstrateur

Enfin, un démonstrateur autonome illustre l'application pratique de PEFSL. Logé dans une boîte compacte, il intègre une carte PYNQ-Z1, une caméra et un écran, atteignant une inférence à 16 FPS avec une consommation totale de 6,2 W. La conception modulaire et la mise en œuvre efficace en termes de ressources mettent en évidence son potentiel pour des déploiements industriels.

Conclusions

Cette thèse a abordé le développement et les défis de la compression de modèles en apprentissage profond, en mettant l'accent sur l'apprentissage fédéré et les systèmes embarqués. Notre exploration a inclus l'utilisation de techniques de compression pour réduire les coûts de communication dans l'apprentissage fédéré. Par ailleurs, nous avons également traité des coûts computationnels avec une approche de co-conception visant à optimiser les modèles pré-entraînés pour des systèmes d'inférence efficaces utilisant une méthode d'apprentissage par peu d'exemples (*few-shot learning*).

Perspectives

Les résultats et discussions qui ont contribué à la réalisation de ce travail ont également ouvert des perspectives pour des travaux futurs.

Entraînement Partiel Compressé

Les travaux futurs pourraient intégrer des techniques de compression avec l'entraînement partiel, où les clients s'entraînent sur des sous-ensembles du modèle global, afin de relever les défis de communication et de calcul. Des stratégies itératives de sortie anticipée (*early-exit*) combinées à la compression pourraient offrir un compromis entre l'efficacité de l'entraînement et les coûts de communication. L'exploration de couches en précision mixte et la quantification de l'impact de la compression sur l'entraînement partiel représentent des directions prometteuses.

Plateforme d'Inférence Efficace

Adapter l'apprentissage fédéré à des modèles personnalisés et compressés spécifiques au matériel constitue une voie potentielle. Combiner des structures globales (*global backbones*) avec des têtes de classification personnalisées et utiliser des techniques de quantification comme FedMPQ pourrait permettre un déploiement efficace. La refonte de cadres comme Tensil pour optimiser des structures en précision mixte améliorerait leur applicabilité dans des contextes réels.

Apprentissage Fédéré Hétérogène de Modèles

Une idée prospective consiste à prendre en charge des modèles clients auto-définis avec des méthodes d'agrégation adaptées pour gérer l'hétérogénéité. La distillation de connaissances, comme dans FedDF [111], pourrait permettre la collaboration entre des modèles divers, avec des modèles de langage large ou multimodaux guidant la fusion des connaissances. Cette approche vise à développer un cadre d'apprentissage fédéré entièrement hétérogène.

Résumé

Les approches traditionnelles centralisées de l'apprentissage automatique exigent la collecte de grands ensembles de données, contenant souvent des informations privées. Cela a conduit à une appréhension croissante concernant la sécurité des données, incitant au développement de nouvelles réglementations, telles que le GDPR, pour protéger la vie privée des utilisateurs. En réponse à ces défis, l'apprentissage fédéré a été développé comme une alternative prometteuse, permettant un cadre d'entraînement collaboratif et décentralisé. Différentes entités peuvent participer à l'entraînement fédéré d'un modèle d'apprentissage automatique sans partager leurs données. Cette technique est particulièrement intéressante pour des domaines comme la santé, où la sensibilité et la quantité des données rendent difficile l'entraînement de modèles à grande échelle.

Cette thèse aborde les défis consistant à rendre l'apprentissage fédéré à la fois efficace et accessible, en se concentrant sur des dispositifs avec des contraintes matérielles diverses. L'apprentissage fédéré améliore la confidentialité en permettant à plusieurs participants de collaborer pour entraîner un modèle sans partager leurs données. Chaque participant entraîne le modèle localement avec ses propres données, ne partageant que les mises à jour du modèle entraîné. Cependant, les différences de ressources matérielles entre les participants posent des défis liés aux coûts de communication et aux limitations computationnelles. Ce travail explore des techniques de compression existantes ainsi que de nouvelles approches pour réduire la charge des coûts de communication tout en maintenant les performances du modèle. Toutefois, l'apprentissage fédéré présente des limitations en terme de confidentialité. D'autres méthodes, comme l'apprentissage avec peu d'exemples, constituent de bonnes alternatives. Par conséquent, nous explorons également comment ces modèles peuvent être déployés efficacement sur des appareils aux ressources limitées, offrant ainsi des perspectives pour réduire les coûts liés à l'apprentissage fédéré.

La première contribution de cette thèse se concentre sur la réduction de la taille des messages dans l'apprentissage fédéré pour minimiser l'utilisation d'énergie et de bande passante. Cette méthode intègre l'élagage des poids de faible magnitude avec l'encodage par entropie, réalisant une réduction de 50% de la taille des messages avec moins de 1% de perte en précision. En élaguant les plus petits poids basés sur les valeurs absolues avant de transmettre les messages, cette approche simplifie le processus de compression et permet aux participants d'adapter l'élagage à leurs jeux des données. Le résultat est une réduction significative des besoins en bande passante tout en préservant les performances du modèle.

La deuxième contribution introduit l'application de l'adaptation de basse rang (LoRA) dans l'apprentissage fédéré pour réduire les coûts de communication sans se reposer uniquement sur les techniques traditionnelles de compression de modèles. La méthode proposée, appelée FLoCoRA, intègre des adaptateurs LoRA dans le cadre de l'appren-

tissage fédéré, démontrant que de petits modèles de vision peuvent être entraînés à partir de zéro. Notre méthode réduit la taille des messages jusqu'à 4,8 fois avec une perte de précision négligeable. De plus, une technique de quantification affine compresse davantage les tailles de messages de 18,6 à 37,3 fois. Cette méthode gèle les paramètres du modèle original et ne met à jour que les adaptateurs LoRA pendant l'entraînement, réduisant les coûts de communication de l'apprentissage fédéré.

La troisième contribution présente une chaîne de co-conception de modèles d'apprentissage par peu d'exemples sur une plate-forme FPGA pour la classification d'objets avec des contraintes temps réel, avec des extensions possibles pour l'apprentissage fédéré. Ce système fournit une solution open-source pour la conception, l'entraînement et le déploiement de modèles d'apprentissage profond sur des dispositifs FPGA à faible consommation, atteignant une faible latence (30 ms) et une consommation d'énergie (6,2 W) sur une carte PYNQ-Z1. Cette méthode aborde le défi de la rareté des données, permettant aux modèles de bien généraliser même lorsque certains participants ne peuvent pas intégrer à l'apprentissage fédéré en raison de contraintes matérielles. En exploitant l'apprentissage par peu d'exemples sur des FPGA, le système assure une flexibilité pour les exemples encore non vus et constitue également un candidat pour déployer un modèle fédéré pour l'inférence.

Abstract

Traditional centralized approaches to machine learning demand the collection of large datasets, often containing private data. This has led to growing apprehension around data security, prompting the development of new regulations, such as the GDPR, to protect users' privacy. In response to these challenges, federated learning was developed as a promising alternative, allowing for a collaborative, decentralized training framework. Different entities can participate in federated training of a machine learning model without sharing their data. This technique is particularly valuable for domains like healthcare, where data sensitivity and scarcity make large-scale model training difficult.

This thesis addresses the challenges of making federated learning both efficient and accessible, focusing on environments with diverse hardware capabilities. Federated learning enhances privacy by allowing multiple participants to collaboratively train a model without sharing their data. Each participant trains the model locally with their own data, only sharing the trained model updates. However, variations in hardware resources among participants introduce challenges related to communication costs and computational limitations. This work explores existing compression techniques and new approaches to reduce communication costs load while maintaining model performance. However, federated learning has shortcomings in terms of privacy. Other methods like few-shot learning are good alternatives. Therefore, we also investigate how few-shot learning models can be effectively deployed on hardware-constrained devices, offering insights into reducing costs for federated learning.

The first contribution of this thesis focuses on reducing message sizes in federated learning to minimize energy and bandwidth usage. This method integrates weight magnitude pruning with entropy encoding, achieving a 50% reduction in message size with less than 1% loss in accuracy by pruning the smallest weights based on absolute values before transmitting messages. This approach simplifies the compression process and allows federated learning participants to tailor pruning to their specific datasets. The result is a significant reduction in bandwidth requirements while preserving model performance.

The second contribution introduces the application of Low-Rank Adaptation (LoRA) within federated learning to reduce communication costs without relying solely on traditional model compression techniques. The proposed method, called FLoCoRA, integrates LoRA adapters into the federated learning framework, demonstrating that small vision models can be trained from scratch while reducing message sizes by up to 4.8 times with minimal accuracy loss. Additionally, an affine quantization scheme further compresses message sizes by 18.6 to 37.3 times. This method freezes the original model parameters and updates only the LoRA adapters during training, reducing the communication costs between federated learning participants.

The third contribution introduces a pipeline for the co-design of few-shot learning mod-

els in an FPGA platform for real-time object classification, with possible extensions for federated learning. This system provides an open-source solution for designing, training, and deploying deep learning models on low-power FPGA devices, achieving low latency (30 ms) and power consumption (6.2 W) on a PYNQ-Z1 board. This method addresses the challenge of data scarcity, allowing models to generalize well even when some participants cannot participate in federated training due to hardware constraints. By leveraging few-shot learning on FPGAs, the system ensures flexibility for unseen examples and has also been a candidate for efficiently deploying a federated learning model for inference.

Acronyms

ASIC Application-Specific Integrated Circuit.
CE Cross-Entropy.
CIFAR Canadian Institute For Advanced Research.
CINIC-10 CINIC-10 Is Not ImageNet or CIFAR-10.
CNN Convolutional Neural Network.
CPU Central Processing Unit.
CSR Compressed Sparse Row.
DDR Double Data Rate.
DNN Deep Neural Network.
DPU Deep Processing Unit.
DSP Digital Signal Processing.
EASY Ensemble Augmented-Shot Y-shaped Learning.
FC Fully Connected.
FedAvg Federated Averaging.
FedDF Federated Distillation Fusion.
FedOpt Federated Optimization.
FedPAQ Federated Learning method with Periodic Averaging and Quantization.
FF Flip-FLop/Register.
FLoCoRA Federated Learning Compression with Low-Rank Adaptation.
FLOP Floating Point Operation.
FP Floating-Point.
FPGA Field-Programmable Gate Array.
FPS Frames Per Second.
FSL Few-Shot Learning.
GDPR General Data Protection Regulations.
GPIO General-Purpose Input/Output.
GPU Graphical Processing Unit.
IID Independent and Identically Distributed.
ILSVRC Large Scale Visual Recognition Challenge.
IOB Input/Output Block.
IoT Internet of Things.
IP Intellectual Propriety.
KL Kullback-Leibler.
LDA Latent Dirichlet Allocation.
LLM Large-Language Model.
LoRA Low-Rank Adaptation.

LSTM Long Short-Term Memory.
LUT Look-Up Table.
MAC Multiply-Accumulate.
MLP Multi-Layer Perceptron.
MUX Multiplexer.
NAS Neural Architecture Search.
NCM Nearest Class Mean.
ONNX Open Neural Network Exchange.
PCIe Peripheral Component Interconnect Express.
PEFSL Pipeline for Embedded Few-Shot Learning.
PEFT Parameter-Efficient Fine-Tuning.
PTQ Post-Training Quantization.
QAP Quantization Aware Pruning.
QAT Quantization-Aware Training.
ReLU Rectified Linear Unit.
ResNet Residual Neural Network.
RTL Register Transfer Level.
SGD Stochastic Gradient Descent.
SLT Successive Layer Training.
SoC Sytem-on-Chip.
SoTA State-of-The-Art.
SVD Singular Value Decomposition.
SWAT Sparse Weight Activation Training.
TCC Total Communication Cost.
TDP Thermal Design Power.
UAT Universal Approximation Theorem.
VGG Visual Geometry Group.

Chapter 1

Introduction

Contents

Preamble

As deep learning development is data-driven, the classic view of centralizing data for training and its privacy implications have been questioned by different research entities and governments [57]. Several solutions address these concerns, such as differential privacy [1], which adds noise to data to protect individual information, and homomorphic encryption [102], which allows computation on encrypted data. Both aim to anonymize and secure data. Another approach that promotes a more privacy-aware framework in machine learning is federated learning. However, this increased privacy comes with a trade-off: the training process is shifted to data owners, allowing them to train models without directly sharing their data. These modifications result in numerous design challenges for this framework. This work focuses on the impacts on hardware-constrained devices caused by the constant communications in the federation and the on-device training of convolutional neural networks.

The introduction is divided into three parts. The first part provides the context and addresses the problems discussed in this thesis. The second part summarizes the chapters of this manuscript, and finally, the third part lists the contributions in terms of methods and publications achieved.

1.1 Context

Machine learning has become a common tool in many fields, including engineering, language understanding, medicine, and many other areas. As the result of a stack of layers, each representing multiple compositions of mathematical functions, such models are scaled up by adding more layers and modifying the layers' composition. As such, models can be composed of a few thousand parameters up to trillions of parameters [165]. To run a model, one needs large amounts of preprocessed data and computational power proportional to the model's complexity. This paradigm has been the main conduit in the application of neural networks and, later, deep learning from the 80s and 90s to today, where the development of powerful hardware accelerators in the form of Graphical Processing Units (GPUs), faster memory and storage solutions, optimized algebraic libraries, easy-to-use research/production frameworks, and extensive data collection have made large-scale machine learning research and deployment possible.

Considering services that interact directly with human data, such as language models, personalized services for image generation, or healthcare services, to train a model, one needs to collect not only a considerable amount of data but also real-world data. This creates a direct problem of dealing with potentially sensible data. In recent years, there has been increasing concern about data privacy and confidentiality. Specifically, there have been numerous new laws and agreements in regions like North America and Europe regarding data protection and efforts to keep users' data and activity private. This scenario has fueled recent debates over the scaling of deep learning models that need more high-quality data and users' right to data privacy and ownership.

In light of current concerns about the General Data Protection Regulations (GDPR), researchers have proposed shifting from the centralized data paradigm to a federated approach [94, 117]. In the federated paradigm, the computing units are distributed, with each unit only able to access its own local data. The principle is that each unit can independently train its own model on its own data. By exchanging the outcomes of this training, rather than the data itself, a third party involved in the federation, or

even one or more units, can explore methods to combine the various models into one global model. The main intent of this approach is to increase data privacy as much as possible while allowing models to be trained with data that would otherwise be hardly accessible. Its most notable application is the development of medical-related machine learning models and applications. Unlike standard image classification or segmentation tasks, for which several public datasets have been curated for benchmarking purposes and the Internet serves as a contentious source for web scraping, medical data presents substantially higher sensitivity and constitutes the sole repository for training relevant applications. A single medical institution may not possess or generate sufficient data, particularly for rare diagnoses or occurrences. Federated learning is an option that allows several institutions to enroll in collaborative training to construct a more representative model.

In a broader context, Federated learning constitutes a collaborative framework in which, in its original formulation, a group of entities, termed clients, and a central aggregation node, referred to as the server, jointly train a machine learning model, keeping training data local to each client. This increase in data protection comes with a series of design challenges that are part of ongoing research topics in the framework. Among many design challenges, we focus on the framework's consequences to clients. From a system perspective, offloading computation to individual clients introduces challenges associated with heterogeneous and constraining computational resources, a drawback less prevalent in centralized configurations. Clients' hardware can range from private or public entities with the capacity to invest in dedicated compute nodes to standard desktop computers characterized by limited memory, storage, and processing power, extending to mobile phones, drones, or Internet of Things (IoT) devices, wherein hardware limitations are intrinsic to their design. As multiple parties participate in the federation, there is a constant need for information exchange to collaborate on the global aggregation of knowledge, resulting in extra costs in terms of communication means and bandwidth.

Hardware heterogeneity and its related constraints are not new concerns in the design of neural network models, as numerous techniques have been developed over the years to address the needs of model compression. One of the most adopted techniques has been to mitigate the over-parameterization of models by augmenting sparsity, given that parameters vary in their pertinence to the outcome or by deleting entire architectural components. Such strategies are commonly known as "pruning", where the intent is to reduce over-parameterization to diminish network complexity and/or size without compromising performance. Another well-established technique, quantization, rooted in signal processing, involves reducing the precision used to represent parameters and/or input data to a model. The central premise is that the conventional data format, IEEE 754 32-bit floating point [120], can be transformed into less complex and more efficient numerical formats for machine learning, resulting in representations that demand less memory space by reducing the number of bits, or into formats that simplify multiplications and accumulations, as seen in the use of integers, ternary, or even 1-bit representations.

Although federated learning is a more privacy-conscious framework, data attack methods have demonstrated the ability to recover clients' data during training. This raises concerns about the applicability of federated learning, especially considering that not all devices can maintain a stable communication link. Embedded devices seeking to implement federated learning are likely to face limitations in both the quantity and diversity of their data. A promising solution to these limitations is to use techniques based on pre-trained models, which harness the power of publicly available data. Among the

various options, few-shot learning [15] stands out as an exciting alternative. Additionally, few-shot learning offers an online learning method that does not rely on computationally expensive gradient descent algorithms, making it more efficient in terms of energy consumption and latency. This makes it an appealing choice for deployment on embedded devices.

In federated learning, the algorithm executed locally by each participant is computationally equivalent to those used in centralized models. This often requires memory and computationally intensive operations to minimize a cost function, making GPUs a suitable hardware choice. In contrast, few-shot learning relies on comparing high-dimensional vectors, where the classification method learns class representations from only a few examples. These high-dimensional vectors, or features, are extracted using a pre-trained model that can be further optimized for inference using compression techniques. For such tasks, hardware platforms like Field-Programmable Gate Array (FPGA), with their reconfigurable architecture, offer an interesting alternative. These platforms allow for the exploitation of different quantization and optimization strategies for deep learning models [100].

This thesis focuses on making federated learning more efficient and accessible, given device restrictions regarding hardware capabilities, by exploring how to shift from centralized model compression techniques to federated ones. Additionally, we explored the low-power, low-latency alternative for embedded devices based on few-shot learning and a FPGA platform. We showed how a pre-trained model could be efficiently deployed in an embedded platform.

1.2 Manuscript Outline

We present a summary of each chapter in this manuscript :

- Chapter 2, "Deep, Distributed and Federated Learning": introduces the main concepts of deep learning that will be explored and referenced throughout this work. From this base, we elaborate on the domain of distributed deep learning to present the framework of federated learning and its open challenges later.
- Chapter 3, "Compressing the Federation": builds upon the foundation laid in Chapter 2, offering an in-depth exploration of the challenges associated with deploying deep learning models on embedded devices. First, standard compression techniques seen in deep learning are revisited to further elaborate on their adaptation to federated learning, as demonstrated in prior research. The chapter reviews the state-of-the-art compression methods for federated learning, discussing the application of pruning and quantization and other alternatives tailored for federated learning.
- Chapter 4, "Cutting Communication Costs": presents our propositions to reduce communication costs in federated learning through two methods. First, by integrating pruning and entropy coding for message exchanges between the client and server. We discuss our design choices and how a simple technique can be incorporated into the federated learning framework while remaining separate from other solutions addressing different issues. Second, we explore how to utilize a combination of low-rank adaptation and quantization further to decrease communication costs between the client and server. We demonstrate how an initial fine-tuning technique designed for transformer architectures can be used to train small convolutional neural networks from scratch. Based on the results of this technique, we also expand our research to question the role of compression in

federated learning compared to the improved design of models. The chapter concludes by revisiting the significance of pre-training in federated learning.

- Chapter 5, "Embedded Few-Shot": presents our proposition for an embedded solution of a system that does not have any communication capability. Without being able to participate in federated learning training and constrained with a limited dataset, our proposal combines co-design techniques between a few-shot learning training algorithm and an FPGA implementation framework to obtain an efficient platform capable of compensating for new data. Initially, we explored how to address the limitation of access to new data and classes with a few-shot learning approach. We propose to adequate a model architecture for our target platform and how to implement it efficiently.
- Chapter 6, "Conclusions": presents a recapitulation of the context addressed in this thesis and the contributions addressed in Chapters 4 and 5, extending the discussion to the future works and perspectives related to each work.
- Appendix A, show an brief introduction to an FPGA, highlyting the different elements that compose the reconfigurable circuit. It also discusses frameworks for deploying deep learning models to FPGA platforms. This appendix is a complement for the discussion on Chapter 5.

1.3 Contributions

The different contributions of this work are summed up hereafter:

1. To address these hardware constraints, lightweight models and compression techniques such as pruning and quantization are commonly adopted in centralized paradigms. We investigated the impact of compression techniques on federated learning for a typical image classification task. We show that a straightforward method can compress messages up to 50% while having less than 1% of accuracy loss. A technique that integrates pruning and entropy encoding into federated learning. This work is discussed in Chapter 4.
2. Low-Rank Adaptation methods are popular for efficient parameter fine-tuning of models containing hundreds of billions of parameters. We demonstrate the application of low-rank adaptation to train small-vision models in federated learning from scratch. We first propose an aggregation-agnostic method, named FLoCoRA, showing that the technique can reduce communication costs by 4.8 times while having less than 1% accuracy degradation for a CIFAR-10 classification task with a ResNet-8. Next, we showed that the same method can be extended with an affine quantization scheme, dividing the communication cost by 18.6 times. FLoCoRA represents a strong baseline for message size reduction, even when compared to conventional model compression works. This work is discussed in Chapter 4.
3. The development of an end-to-end open-source pipeline for a few-shot learning platform for object classification on an FPGA Sytem-on-Chip (SoC). The pipeline is built on top of the Tensil open-source framework, facilitating the design, training, evaluation, and deployment of Deep Neural Network (DNN) backbones tailored for few-shot learning. We showcase our work's potential by building and deploying a low-power, low-latency demonstrator trained on the MiniImageNet dataset with a systolic-array architecture. The proposed system has a latency of 30 ms while consuming 6.2 W on the PYNQ-Z1 board. This work is discussed in Chapter 5.

These various contributions have been submitted to scientific publications :

- National symposiums presentation:
  - Grativol, L., Gauthier, L., Léonardon, M., Morlier, J., Lavrard-Meyer, A., Muller, G., & Arzel, M. (2024, May). PEFSL: A Deployment Pipeline for Embedded Few-Shot Learning on a FPGA SoC. In Colloque Nationale du GDR SoC2 2024.
- National conferences without proceedings:
  - Grativol, L., Léonardon, M., Muller, G., Fresse, V., & Arzel, M. (2023, August). Compression de réseaux de neurones pour l'apprentissage fédéré. In XXIXème Colloque GRETSI.
- International conferences with proceedings:
  - Grativol, L., Léonardon, M., Muller, G., Fresse, V., & Arzel, M. (2023, December). Federated learning compression designed for lightweight communications. In 2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS) (pp. 1-4). IEEE.
  - Grativol, L., Gauthier, L., Léonardon, M., Morlier, J., Lavrard-Meyer, A., Muller, G., Fresse, V., & Arzel, M. (2024, May). PEFSL: A Deployment Pipeline for Embedded Few-Shot Learning on a FPGA SoC. In 2024 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). IEEE.
  - L. Grativol, M. Léonardon, G. Muller, V. Fresse and M. Arzel, "FLoCoRA: Federated Learning Compression with Low-Rank Adaptation," 2024 32nd European Signal Processing Conference (EUSIPCO), Lyon, France, 2024, pp. 1786-1790.

Chapter 2

Federated Learning for Image Classification

Contents

This thesis focuses on federated learning algorithms, to which we contribute by addressing their design challenges. Although federated learning is intended to be used with a plurality of machine learning algorithms, we focus on deep learning algorithms. Therefore, in this first section, we will introduce the task deep learning models are used to solve in this work. Afterward, we introduce the concept of distributed learning, which is a broader topic on which federated learning is built. Finally, we introduce the concepts of the federated learning framework. The definitions and baselines established in this chapter are reused in the rest of this manuscript.

2.1 Deep Learning for Image Classification

A deep learning algorithm consists of an architecture (see Section 2.1.3) for the model that will be trained on a specific collection of data (see Section 2.1.2) to approximate a specific function that models or solves a given task. Using these notions, in this work, we define a centralized setting as one with access to the architecture and the dataset used to solve a specific task. This definition is later confronted with the distributed learning paradigm of which federated learning is part.

2.1.1 Image Classification Task

As one of the most common tasks in machine learning [113, 139], image classification consists of giving an image as input, and we expect to obtain a corresponding numerical value representing the label of the most important information present in the input. One could describe the input of this task as being an X $\in \mathbb{R}^{C\times H\times S}$ and the output as $Y \in \mathbb{R}^n$, where n represents the number of possible labels of a function $\mathcal{F} : X \to Y$, mapping an image to its corresponding label.

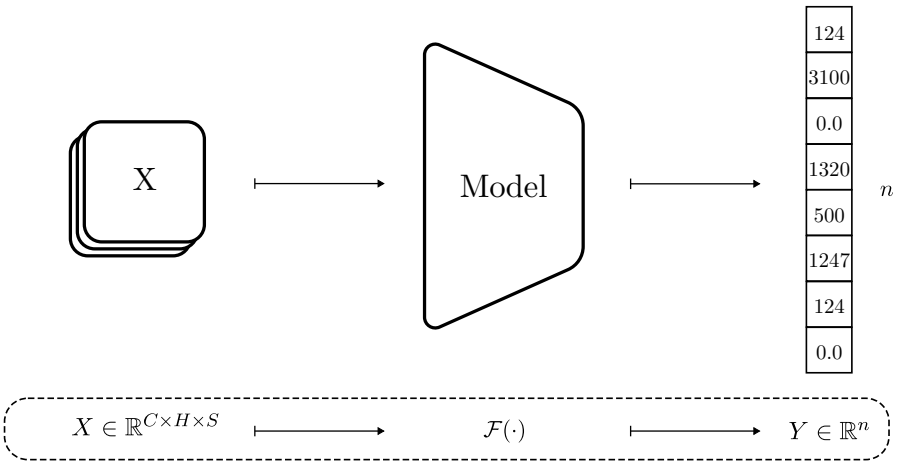


Figure 1 – The input X, which is an image, is processed by the model represented by the function $\mathcal{F}$, from which the corresponding labels are yielded as the output Y.

Figure 1 illustrates an image classification task. By inputting data corresponding to X, the output of the model, Y, is a vector of size n, also called logits. Logits represent the raw unnormalized confidence levels of each of the n possible labels or classes of the input. An activation function is applied to the logits to normalize them, transforming them into a probability distribution over the possible outputs. The index with the highest probability value represents the predicted label for the input.

2.1.2 Datasets

As deep learning is a data-driven field, to measure how good a particular algorithm is for a specific task, collections of curated data were built [133]. These well-known datasets are task-specific, and by definitions of their curating process, they try to represent the possible diversity that a task could have. Here, we introduce three datasets used as benchmarks for the experiments in this work.

2.1.2.1 CIFAR-10

Named after and developed by the **C**anadian **I**nstitute **F**or **A**dvanced **R**esearch in 2008, CIFAR-10 [95] is a collection of images used for computer vision, commonly used in image classification tasks. The dataset comprises 60,000 color images, each with a resolution of 32×32 pixels, divided into 10 distinct classes (hence the name), each with 6000 examples. Commonly, the dataset is split to reserve 50,000 examples for training and 10,000 for validation. Due to the input size, 32×32, and the number of classes, the CIFAR-10 is one of the most well-known datasets in the domain. The input size and the number of examples made CIFAR-10 a resource-accessible dataset. A sample is presented in Figure 2.

Figure 2 – CIFAR-10, classes samples. Source: `https://www.cs.toronto.edu/~kriz/cifar.html`

2.1.2.2 CIFAR-100

As CIFAR-10, CIFAR-100 [95] was built from the "80 Million Tiny Images" dataset, containing 60,000 color images of 32×32 pixels, but divided into 100 classes, resulting in 600

images per class. The CIFAR-100 represents a more challenging task than the CIFAR-10 due to the increased number of classes and the smaller number of examples per class. A sample is presented in Figure 3.

Figure 3 – CIFAR-100, classes samples. Source: same as CIFAR-10.

2.1.2.3 ImageNet

Built initially for the Large Scale Visual Recognition Challenge (ILSVRC), ImageNet [45] is one of the most comprehensive datasets for object recognition and image classification tasks in computer vision. ImageNet contains millions of labeled images spanning thousands of categories, making it a benchmark for evaluating the performance of various machine learning models. The images vary in resolution and cover various everyday objects, animals, and scenes.

A well-known and used subset of ImageNet, the **ImageNet-1K**, consists of 1,000 classes with around 1.2 million training images and 50,000 validation images. The images obtained from scrapping the internet have different resolutions, with the largest having 4288×2848 pixels and the smallest having 75×56 pixels. This subset has become the standard benchmark for deep learning models in image classification, as its large scale and diversity in object categories make it a complex task. The ImageNet-1K dataset benchmark is an important landmark for image classification models being used to push and develop the domain. A sample from ImageNet-1K is illustrated in Figure 4.

Figure 4 – ImageNet-1K, classes samples. Source:`https://www.image-net.org/`

The ImageNet dataset was also used as a base to build other datasets. Notably, we introduce the CINIC-10 and MiniImageNet datasets that are later used in this work.

Built as an expansion of the CIFAR-10 dataset, the **CINIC-10** [42] dataset was created to bridge the gap between the complexity of small-scale datasets like CIFAR-10 and large-scale ones like ImageNet. CINIC-10 contains a combination of CIFAR-10 and downsampled ImageNet images, with 90,000 training images, 90,000 validation images, and 90,000 test images, each split across 10 classes. The dataset was designed to challenge models with more varied visual input while maintaining a manageable size for researchers who lack the resources to work with massive datasets like ImageNet. The CIFAR-100, in comparison to the CIFAR-10, has 100 classes but with "only" 500 examples per class, making it a more challenging task than the CINIC-10.

MiniImageNet [166], a curated subset of ImageNet, was built to facilitate the study of few-shot learning in image classification tasks. The dataset contains 100 classes with 600 images per class, providing a balanced and smaller-scale version of the original ImageNet, ideal for testing algorithms that learn from a few labeled examples. Due to the specificities of the few-shot learning paradigm, more details on the dataset are

discussed in Chapter 5 and Section 5.3.

2.1.3 Architectures

Deep learning algorithms have been driven by the need to handle increasingly complex and diverse data. The concept of stacking layers is used to design different architectures that are capable of learning intricate patterns in different types of signals. By stacking multiple layers, deep networks can progressively abstract higher-level features from the data, increasing the complexity of the task that the model can handle. The motivation for such a design comes twofold, first with the Universal Approximation Theorem (UAT) [41], which states that a shallow neural network, with few layers, can approximate any continuous function. Second, since the UAT does not determine the adequate size or how to train the neural network, stacking becomes a way to construct a hierarchical feature learner, generating a deeper neural network capable of capturing more complex patterns [101]. Also, as shown by [75], the deeper network becomes capable of emulating a shallow model without getting to the scaling problem of a shallow network that should grow indefinitely in its size to accommodate more complex problems [78].

In terms of layer stacking for image classification, for instance, the initial layers might identify edges and textures, the intermediate layers recognize parts of the object, and the deeper layers detect entire objects. This reasoning applies to computer vision tasks and other fields, such as natural language processing, audio, and complex signals.

This thesis has focused on using Convolutional Neural Networks (CNNs) for image classification. As such, we present the different elements of CNN architecture based on the two models. The objective is to introduce the different elements in CNN architectures following the evolution from VGG architecture to ResNet. The descriptions are focused on the **Visual Geometry Group** (**VGG**) [149] family of models and on the **Residual Neural Networks** (**ResNets**) [75] family.

In terms of general notation, the subscript l indicates the layer index of a supposed model, $\mathcal{F}$, containing L layers, as so $l \in \{0, ..., L-1\}$. The input of a layer is represented by X. We also refer to the input of a layer as being an activation. The weights of a specific layer l are represented as W_l, while $\mathcal{W}$ represent the weights of the entire model. In this case, $\mathcal{F}_{\mathcal{W}}$ represents the model $\mathcal{F}$ parametrized by $\mathcal{W}$.

2.1.3.1 VGG-family

The VGG family was proposed following the idea of layer stacking and the success of the AlexNet [96] model. The basis of this architectural family is shown in Figure 5. Next, we explain the different elements that compose this architecture and the motivation for its adoption.

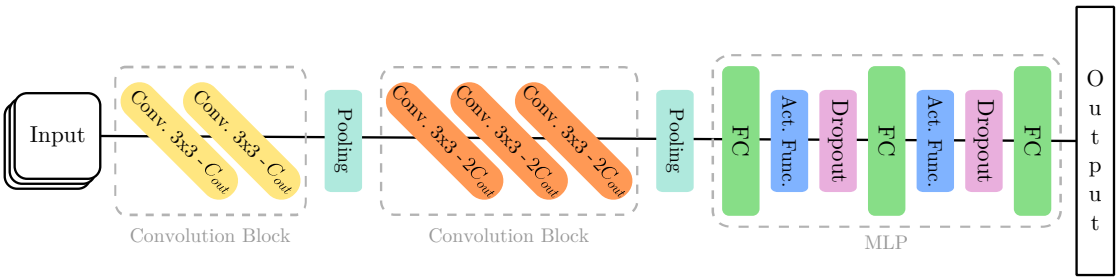


Figure 5 – A simplified VGG architecture.

Convolutional Layers - Going from the input to the output, the VGG family proposed to stack **convolution layers**, which are the principal components of CNNs [55]. These layers apply filters to the input data to detect spatial hierarchies. Figure 6 demonstrates the most simple case of a 2D convolution operation. The input to the current layer corresponds to the outputs generated by a previous layer $l-1$, in the form of $X_{l-1} \in \mathbb{R}^{C_{in} \times H_{in} \times S_{in}}$. Where C_{in} indicates the number of input channels, S_{in} and H_{in} are the width and height of each activation per channel. As well that C_{out}, S_{out}, H_{out} represent the output shape. For each output channel, there is a filter, $W_l^{(i)}$ for $i \in \{1, ..., C_{out}\}$, composed of kernels $K^{(i)}$. Each kernel possesses C_{in} channels of size $k \times k$, forming square kernels. As such, a particular kernel channel can be represented by $W_l^{(i,j)}$, where j represents the kernel channel. In practice, rectangular kernels are also possible, but we limit the discussion to square kernels. The number of parameters in this type of configuration of a convolution layer is $C_{out} \times C_{in} \times k \times k$, plus the bias term, β_l if it is present. In the figure, the bias term is omitted for simplicity. To produce the output, X_l, a 2D cross-correlation operation, represented by $*$, is performed between the input and the different filters. The cross-correlation result between the input and each kernel is summed to produce one corresponding output feature. Note the difference between the input and output sizes, $H_{in} \times S_{in}$ to $H_{out} \times S_{out}$, as the operation cross-correlation can compress/dilate the input features. Equation 1 is a general equation to the operation depicted in Figure 6, for each output channel in $X_l^{(j)}$. The operation is demonstrated in its simplest case, without padding, no dilation, a stride of 1, square kernels, and no grouping [134].

In the VGG architecture, the kernel size was fixed 3×3 kernels. Another noticeable architecture design from the AlexNet model is that the number of output channels doubles between the convolution blocks. Convolution blocks in VGGs are formed as stacked convolution layers without any operation in between. Usually, the first convolution block has two convolution layers, with C_{out} channels; subsequent blocks stack three convolution layers, doubling the previous layer's number of channels, $2 \cdot C_{out}$.

$$X_l^{(j)} = \sum_{i=0}^{C_{in}-1} W_l^{(i,j)} * X_{l-1}^{(i)} \tag{1}$$

Pooling Layers - In between the convolution blocks, we have the pooling layers [58], whose function is to reduce the spatial dimensions of the feature maps, retaining essential features while reducing computational load. In the VGG architecture, we find the **max-pooling** layer whose function is to downsample the feature map by taking the max values of the values inside a kernel window. For example, for a certain feature map (activation) of dimensions $C_{in} \times H_{in} \times S_{in}$, a max-pooling of kernel $k_p \times k_p$, would reduce the feature map to $C_{in} \times \frac{S_{in}}{k_p} \times \frac{H_{in}}{k_p}$. We assumed for this example that the activations are divisible by the kernel size; in the case of a mismatch, the final shape would approximated. Another common version of the pooling layer is the average pooling, which takes the average of the elements inside the kernel.

The Multi-Layer Perceptron (MLP) layer is composed of several layers that are detailed next, where the input to the block is the flattened view of the feature maps generated by the last pooling layer.

Fully Connected Layers - Inside the MLP block, the first layer is the Fully Connected (FC) layer, also known as a linear layer. It is typically used to integrate features learned

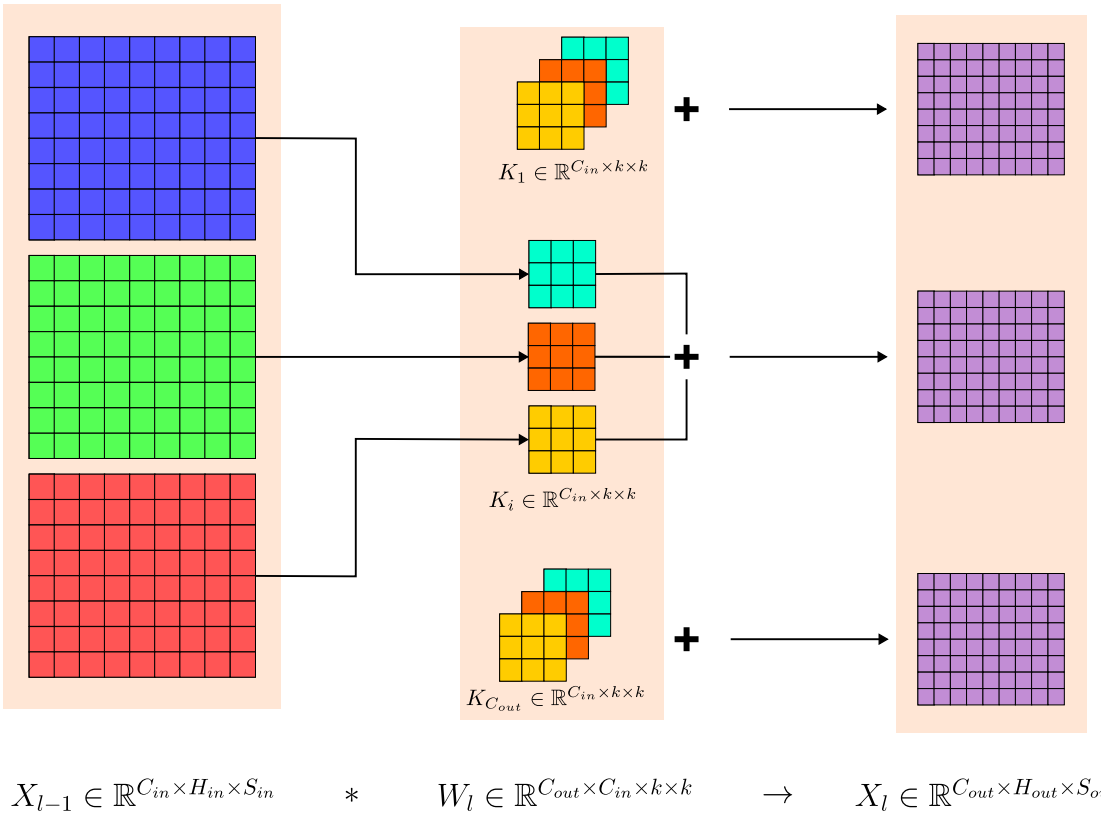


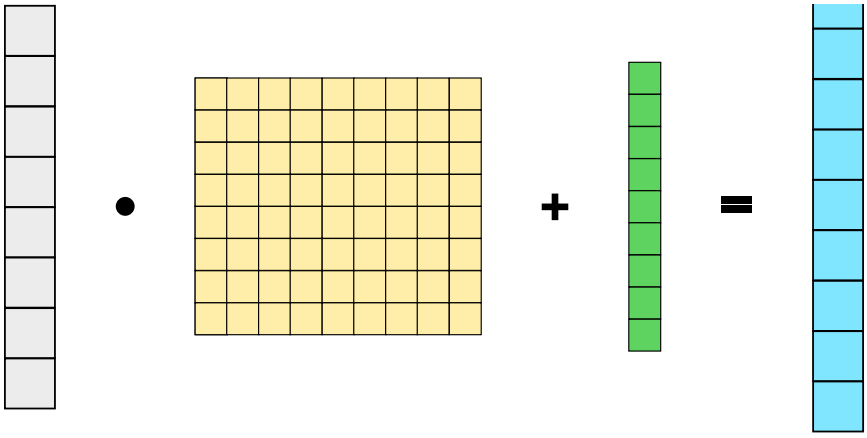
Figure 6 – Convolutional layers representation.

by previous layers. Their role is crucial in combining abstract features for decision-making processes. Figure 7 shows a graphical representation. The layer is characterized by its weight matrix W_l, which contains $I \times O$ elements and a bias vector, β_l, of O elements. To produce the output, X_l, a matrix multiplication operation is performed between X_{l-1} and W_l, and the bias term is added to its results. Equation 2 represents the operation described in the previous figure. Note that essentially, the FC layer is a linear map from $\mathbb{R}^I \rightarrow \mathbb{R}^O$. By stacking multiple FC layers, one can construct a MLP [143], one of the first models in the domain of neural networks.

$$X_l = X_{l-1} W_l + \beta_l \tag{2}$$

Activation Functions - To add some non-linearity into the network, to allow the model to learn complex patterns [50], Activation Functions (Act. Func.) are added. The VGG architecture uses the Rectified Linear Unit (ReLU) function, which can be interpreted as a max operation between the input and zero, $max(0, X_l)$. In modern CNN architectures, activation functions are commonly introduced after convolution layers.

Dropout Layers - Dropout [151] are special layers used only during training, whose function is to prevent the model from overfitting. When overfitting, the model performs well on trained data but poorly on unseen data. Dropout is a regularization technique that randomly "drops" or deactivates a part of a layer for each training iteration, replacing the dropped parts outputs by zeros. This forces the model to learn more robust features by preventing it from relying on any particular path in a layer, enhancing its generalization ability.



$$X_l \in \mathbb{R}^I \quad \cdot \quad W_l \in \mathbb{R}^{I \times O} \quad + \quad \beta_l \in \mathbb{R}^O \quad \rightarrow X_{l-1} \in \mathbb{R}^O$$

Figure 7 – Fully Connected layers representation and equation.

The final output is the logits vector, representing the raw confidence level of the model. To obtain a probabilistic vector representing the label/class pertinence of the input, the logits are fed to a softmax function [25]. The softmax function is also considered an activation function, and Equation 3 represents its operation. For a certain $X_{logits} \in \mathbb{R}^n$, representing the n classes of the task, is normalized based on the exponential sum of the elements in X_{logits}.

In an actual VGG architecture, like the VGG-16, the convolution block is repeated 15 times, totalizing 16 blocks with the first convolution block. In general, the model's names are given based on the number of convolution blocks plus one, and the MLP block at the end has its dimensions adapted in function of it.

$$Softmax(X_{logits}^{(i)}) = \frac{e^{X_{logits}^{(i)}}}{\sum_{j=0}^{j=n-1} e^{X_{logits}^{(j)}}} \tag{3}$$

2.1.3.2 ResNet-family

The ResNet family improved the parameter inefficiency of the VGG family and the problem of the vanishing gradient [75] caused by the depth-scaling style adopted. To solve such problems, ResNets have introduced the concept of residual connections, as seen in Figure 8. These residual connections, also called "shortcuts" within convolutional blocks, connected the input of a block to its output. The primary goal was to enable gradients to flow from the output back to the input of a block. This was intended to address the vanishing gradient problem, which occurs when the feedback signal, used for updating parameters, passes through the model and becomes increasingly smaller due to the large number of layers. Further details on the training algorithm can be found in Section 2.1.4. An interesting consequence between the VGG and ResNet architectures is how the latter improved upon the number of parameters and operations per accuracy [29].

The ResNet architecture consists of three main parts. The first part is the embedding block that can be used to reduce the input resolution. The second part is composed of a series of residual blocks, as seen with the convolution blocks in the VGG family. The final part is an average pooling layer and a classification head consisting of a single linear layer. In Figure 8, we represent one of the two classical residual blocks, the "basic

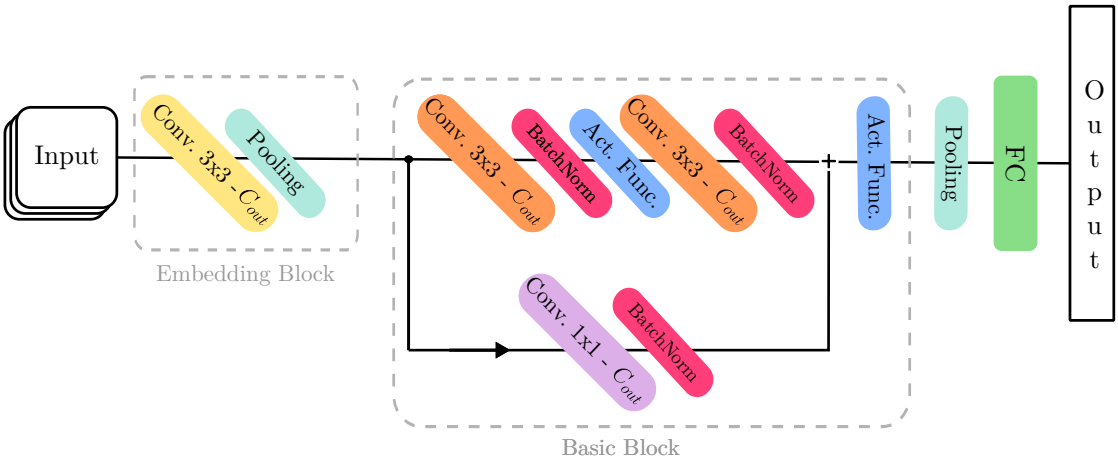


Figure 8 – A simplified ResNet architecture, with a basic block.

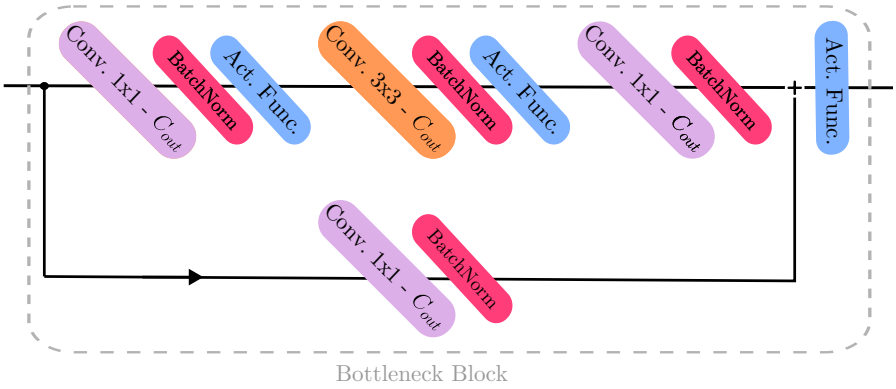


Figure 9 – A bottleneck block for the ResNet architecture.

block", the other one being the "bottleneck block" represented in Figure 9. The difference between them is the use of convolution layers with kernels of 1×1 to reduce the resolution.

Normalization Layers - One important addition, compared to the original VGG proposal, was the inclusion of a normalization layer. The most popular type is Batch Normalization [20] (BatchNorm), which normalizes inputs across a batch. This helps to mitigate issues related to internal covariate shifts, accelerating and facilitating the training of deeper models. For example, batch normalization consists of centralizing X_l by its mean value, $E[X_l^{(i)}]$, and normalizing by its standard deviation, as per Equation 4. In addition to the normalization operation, there are two learnable parameters, γ and β. During training, the layer learns these variables to scale and shift the feature maps between layers, which are used during the inference phase. Finally, a small value ϵ is added to the denominator in Equation 4 to provide some computing stabilization for cases where the standard deviation value could be too small. Typical values for ϵ are around 10^{-7} [93].

$$X_{l+1}^{(i)} = \frac{X_l^{(i)} - E[X_l^{(i)}]}{\sqrt{Var[X_l^{(i)}] + \epsilon}} * \gamma^{(i)} + \beta^{(i)} \tag{4}$$

Effectively, ResNets were not the first to adopt normalization layers, and later, VGG versions started using them between convolution layers, too. ResNet also adopted the

scaling law of doubling the amount of output channel for each basic/bottleneck block. In this chapter, we will refer to a ResNet-20 model, illustrated in Figure 10. This model consists of multiple basic blocks, each containing two convolution layers, along with an initial convolution layer in the embedding block and, finally, a linear layer. This totals 20 layers, which is where the model gets its name. The example is given for the CIFAR-10 dataset, where the output size of each part is indicated below.

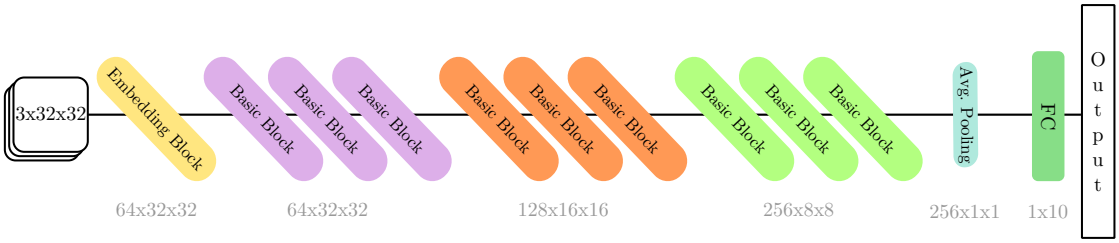


Figure 10 – Block views of a ResNet-20 model for the CIFAR-10 dataset.

2.1.4 Deep Learning Training: Supervised Learning

Having a model's architecture and the dataset for the target task, one can combine these elements with an optimization algorithm to train the model to solve the task. Taking the image classification as an example: from a dataset D, we can sample a pair $(e, y) \in D$, representing one sample of the data, $e \in \mathbb{R}^{C \times H \times S}$, and its corresponding class vector, $y \in \mathbb{R}^n$, for a task with n classes. The class vector is encoded as a one-hot vector, with a value of one for the index representing the class and zeros for all other indices. As the input X is fed to the model, it produces an output, $\hat{y} = \mathcal{F}(X)$, representing the probabilistic vector characterizing X among the possible classes. The model's parameters are initialized following some uniform distribution, which means that the initial values of $\hat{y}$ are far from representing the correct class of X. In order to do so, it is necessary to optimize the parameters of $\mathcal{F}(.)$ to fit the task at hand. In this process, a loss, also called cost, function is first calculated to measure the estimation error between $\hat{y}$ and y. The loss function is denoted by $\mathcal{L}(\hat{y}, y)$, which maps the difference between the two vectors to a single scalar value. The expression can then be expanded to incorporate the weights (parameters) of the model and represent the learning objective, which is to determine the optimal set of parameters $\mathcal{W}$ that can minimize the loss function $\mathcal{L}$. Equation 5 represents the Cross-Entropy (CE) function commonly used as a loss function in conjunction with a softmax function to measure the difference between the two distributions $\hat{y}$ and y. The learning objective can be expressed as $\text{argmin}_{\mathcal{W}} \mathcal{L}(\mathcal{F}_{\mathcal{W}}(X), y)$.

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(\mathcal{F}_{\mathcal{W}}(X, y)) = -\sum_{i=1}^{n} y^i log(\mathcal{F}^i_{\mathcal{W}}(e)) \tag{5}$$

Given the error signal, we can search to optimize the parameters in the model by applying the backpropagation. The idea is to apply the chain-rule derivation to be able to find a P that minimizes $\mathcal{L}$. The model $\mathcal{F}(.)$ can be seen as a composition of the functions, f_l, representing each one of the layers, $\mathcal{F}(.) = f_L \circ f_{L-1} \circ ... \circ f_1(X)$, for a model with L layers.

So the chain-rule can be applied to $\frac{\partial \mathcal{L}}{\partial \mathcal{W}}$ as $\frac{\partial \mathcal{L}}{\partial \mathcal{W}} = \frac{\partial \mathcal{L}}{\partial X_L} \cdot \frac{\partial X_L}{\partial X_{L-1}} \cdot ... \cdot \frac{\partial X_l}{\partial W_l}$, until a certain layer l. So, the gradient of the loss function can be interpreted as:

- $\frac{\partial \mathcal{L}}{\partial X_L}$: As $\mathcal{L}$ is measuring the prediction error, this expression measures the influence of the last layer output on the error, creating the first feedback error signal to

be propagated to the other layers.

- $\frac{\partial X_L}{\partial X_{L-1}}$: The effect of layer $L-1$ on the last layer, which depends on the feedback signal from the last layer and the weights of the current layer $L-1$.
- $\frac{\partial X_l}{\partial W_l}$: For a certain layer l, its activation depends on the values of the weights or learnable parameters associated with it.

This way, we build a derivation graph per learnable parameter with its associate gradient. Finally, the optimization step consists of updating the weights in the opposite direction of its gradient in order to minimize the loss, $\mathcal{W}^* = \mathcal{W} - \nabla(\mathcal{L}(\mathcal{F}_{\mathcal{W}}(X), y))$. This optimization algorithm is also famously known as gradient descent [143], for which the weights ideally move to a minimum in the loss landscape. In reality, the weights update expression has a tuning parameter η, $\mathcal{W}^* = \mathcal{W} - \eta\nabla(\mathcal{L}(\mathcal{F}_{\mathcal{W}}(X), y))$, to coordinate smaller updates and avoid overfitting the model for new data, while forgetting the old ones. The gradient descent algorithm applies the optimization process after the model has seen the entire dataset. This is repeated multiple times, known as epochs, to let $\mathcal{F}_W$ better fit the task distribution.

As a common practice, the dataset is separated into batches instead of using the whole dataset for one update. This allows faster convergence as the weights are updated more frequently and with smaller data portions. To which the sampled data becomes $(X_B, y_B) \in D$, carry B examples of (X_B, y_B). After each batch, the optimization process is applied, composing multiple iteration steps inside one epoch. One epoch represents then the point where the model has seen all the batches. The cross-entropy function is reformulated as seen in Equation 6. Another significant impact is that reducing the amount of data that the model needs to see at once reduces the amount of memory associated with the data, improving efficiency and memory management. In this case, the gradient descent algorithm takes the form of the Stochastic Gradient Descent (SGD) algorithm [144], where for every batch, the model's parameters are updated to minimize the associate error for the batch.

$$\mathcal{L}(\hat{y_B}, y_B) = \mathcal{L}(\mathcal{F}_{\mathcal{W}}(X_B), y_B) = -\frac{1}{B}\sum_{j=1}^{B}\sum_{i=1}^{n} y_B^{(i,j)} log(\mathcal{F}_{\mathcal{W}}^{(i,j)}(X_B)) \tag{6}$$

This describes the basic mechanisms behind Supervised Learning, where one uses the ground truth as feedback for the training process. Other training methods [130], such as Unsupervised, Self-Supervised, Semi-Supervised, and Reinforcement Learning, also exist but are not described or used in this work.

2.1.4.1 Running on Hardware

The execution of a model is separated into inference and training, where inference refers to the use of the execution of the model without the construction of the error propagation graph. Regarding training, the amount of memory necessary to store the samples fed to the model and its activations scale directly with the dataset and the model size. An important mark in the history of deep learning was the moment when GPUs started being used to accelerate the training phase. This is thanks to the fact that graphic algorithms are heavily based on matrix operations, as we find in the inference/training processes of deep learning models.

2.2 Distributed Learning

So far, we have introduced the basic elements that compose a neural network and briefly explained how one training procedure is structured. In this section, we take a step forward and introduce the efforts made to distribute the learning process. This section introduces different paradigms of distributed learning frameworks, to serve as a context for problems that are close to or came before federated learning.

2.2.1 Ensemble Learning

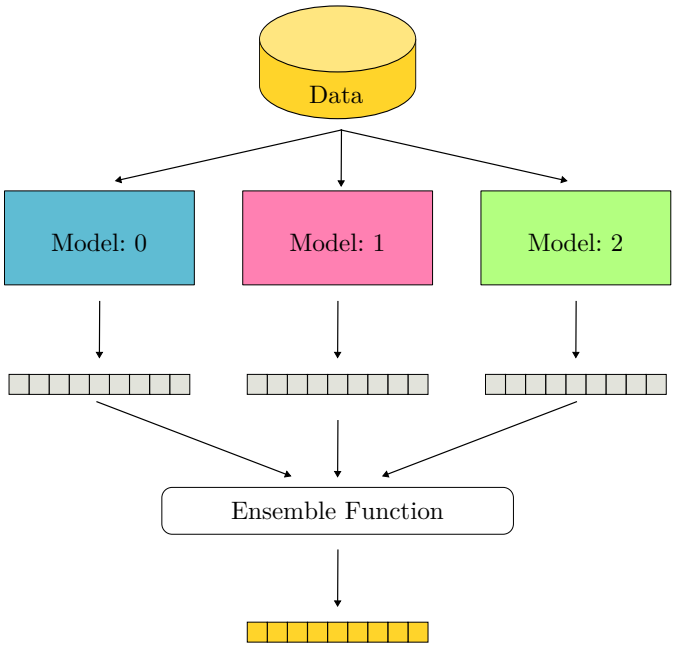


Figure 11 – An ensemble learning paradigm. Three models are trained on the same data, and their outputs are combined through an aggregation function. Adapted from: [4].

In ensemble learning [48], multiple models are trained independently to solve a certain task, being combined to have an improved performance over having one model. In Figure 11, we demonstrated an example of stacking [51]: three different models receive the same samples from a certain dataset, and their outputs are combined with the "ensemble function" to produce the final prediction. Other variations of ensemble learning exist, with the most notable being Bagging and Boosting [56].

2.2.2 Model and Data Parallelism

Distributed learning [164] is often used for two main reasons: to accelerate the training process and address memory limitations. In either case, there are two possible cases; the first one, shown on the left of Figure 12, is to partition the model computing graph into different parts and to attribute them to different GPU/accelerator nodes, also called model parallelism. Node 0 receives all the data, while the results of this node are passed to the other nodes. Another possibility, data parallelism, is to partition the dataset to different computing nodes, replicating the same model to each node but with different data partitions, as shown to the right of Figure 12. Distributed learning algorithms have gained more attention in the era of Foundation Models scale of models, where multiple GPU nodes are necessary to be able to train and execute one model.

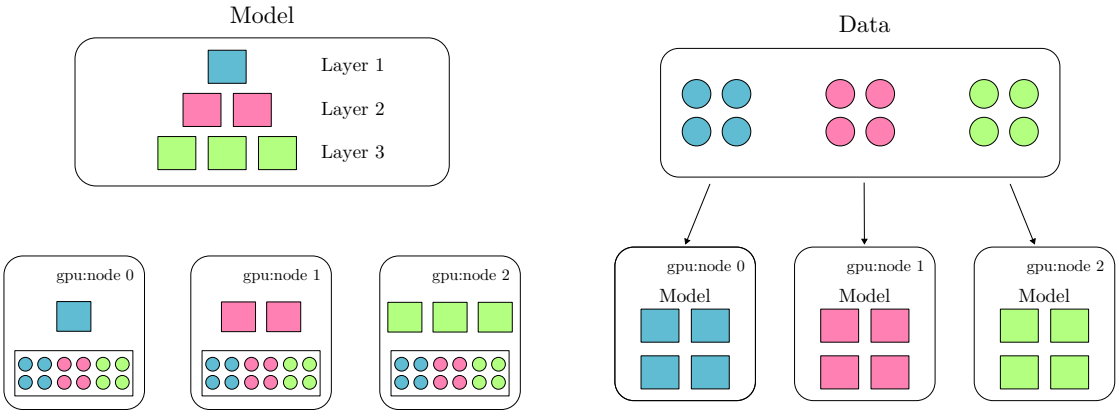


Figure 12 – On the left, the model parallelism paradigm. On the right is the data parallelism paradigm. Adapted from: [115].

2.2.3 Split Learning

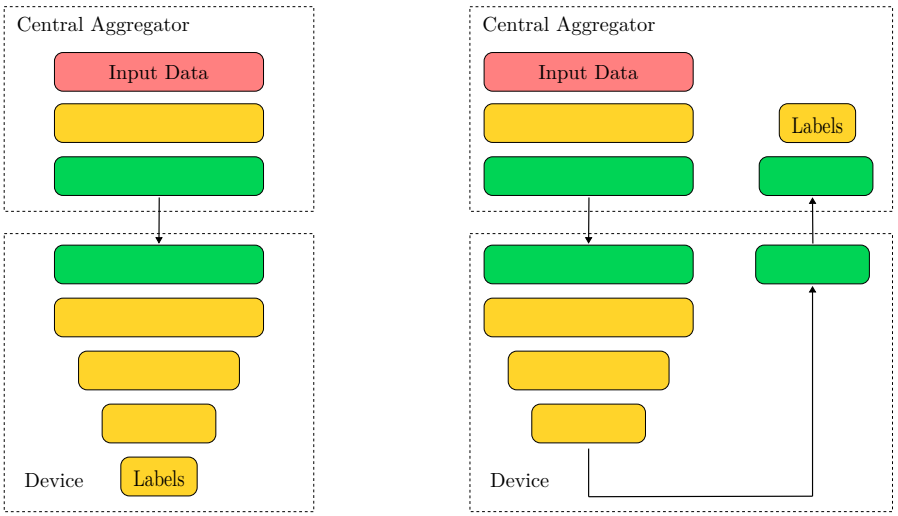


Figure 13 – Two types of split learning. On the left, data is kept local, but the labels are not; this is known as vanilla split learning. On the right, by partitioning more, the model, data, and labels are kept at the device level, known as u-shaped split learning. Adapted from: [89].

The idea of split learning [163] is to share the computational burden with different training entities while keeping their data local. Instead of having a central data and model aggregator, split learning considers that multiple devices/entities would participate in the training of a deep learning model. So, taking inspiration from model parallelism, the model is partitioned among each participant in training. The vanilla split learning algorithm, shown to the left of Figure 13, is done so that a partition containing the model's initial layers is executed on the training device. Meanwhile, the rest of the model is executed in a central aggregator orchestrating the training loop. Data are kept at the device level during training, while the central aggregator possesses the labels. To perform the optimization step, the central aggregator needs to calculate the loss with the results of the devices and propagate the gradients back to the devices to update the model. Information is exchanged at least twice between the central aggregator and the devices, resulting in some training overhead. Moreover, the labels on the central aggregator indicate some violation of the participants' privacy. Some variations of the framework try to solve these problems by proposing unsupervised and self-supervised

learning regimes or by also having the final classification layer on the training participants, as shown to the right of Figure 13.

2.3 Federated Learning Problem Definition

Data are becoming increasingly important worldwide in high-frequency trading, product placement (marketing), or machine learning applications. Data privacy can be critical in these domains, as data can be acquired and monopolized without their owners' direct consent or knowledge. The situation contributed to the increase in awareness and concern about data privacy, particularly with data-hunger algorithms, such as deep learning, where the quantity and quality can significantly improve a model, and data acquisition for such models has become a race. Moreover, centralized data collection can be subjected to interception during communication and unauthorized access by the owner of the computing node or a third party. In this context, Federated Learning positions itself as a training alternative to the centralized view, having redesigned the learning framework to increase data protection.

2.3.1 An Attempt For a More Private Machine Learning Framework

Federated learning [94, 23, 117], much like split learning, is a collaborative and distributed machine learning approach. This framework was designed to enable machine learning models to be trained on decentralized data stored across multiple participant devices, referred to as **clients**. The primary advantage of this approach is that it is designed to keep the data on local devices. The primary goal of federated learning is to mitigate data privacy violations by preventing data sharing.

The federated learning process typically begins with an orchestrating entity, commonly known as the **server**. The server first selects a sub-sample of the pool of clients to participate in the training loop. The server then sends each selected client a copy of the most recent model checkpoint. Once the clients receive the model, they train it using their local data. Clients only share the training results back to the server, which may include updated model weights, gradients, or other forms of proxy information, representing the effect of fitting the previous checkpoint on their data. Upon receiving these updates, the server performs an optimization step in which it aggregates the various training results from all participating clients. This aggregation is crucial, as it seeks to fuse the diverse knowledge learned by the model across different clients into a single, improved model. This client selection process, local training, and result aggregation is known as a communication **round** and is repeated for multiple rounds until the model achieves a predefined performance.

As the framework evolved from the initial proposition, other elements were added to its formulation, creating different aspects of the field. Here, we define the main elements commonly used to characterize a federated learning algorithm.

The different elements presented in federated learning [89] can be categorized based on how the system is orchestrated and synchronized, the types of clients involved, the statefulness of each client, and how data are distributed and partitioned.

1. **Orchestration**: Refers to how the overall training loop is managed and who coordinates clients' updates.

    - Centralized: In this setup, the one responsible for coordinating clients' participation and updates is a central server. The server is responsible for se-

lecting who integrates the current training round, it manages all the communication protocols and related steps, aggregates the training results and it propagates back the final, global, aggregated model/update. The original work on the domain considered this orchestration.
- Decentralized: Here, there is no central server. Clients communicate directly with each other to perform updates and communications. Examples of such orchestration involve blockchain and drone swarm applications.

2. **Synchronization**: Defines how model updates are processed between each round.
   - Synchronous: The orchestration entity waits for all participating clients to submit their results to produce one global model that is synchronized between rounds.
   - Asynchronous: Clients can train and send updates at different moments. The server updates the global model as soon as it receives new information, avoiding round bottlenecks associated with clients that fail to train a model or that possess limited communication systems. The result is a training loop, where the model trained is not always the same between clients, which can lead to update divergence between rounds.

3. **Client's type**: Categorizes clients based on their scale, computing and computation power, and availability.
   - Cross-device: This case is commonly categorized as involving a large number of clients, typically mobile or edge devices, with limited computational power, communication capabilities, and data. The idea is that a large and variable pool of clients can bring variability and representativity to the model. Clients are also not considered always available or able to finish and provide their training results. Clients are not expected to be part of the entire training loop, making it possible for a certain client to participate in only one round. The number of clients is in the hundreds to millions of clients.
   - Cross-silo: In this case, clients are considered to be organizations or institutions (silos) with significant computational resources and larger datasets. They are expected to be always available and reliable during the training loop. Cross-silo scenarios are expected to have a few dozen clients, as it is expected that training institutions holding both data and computing power are scarce. The most notable example is a group of medical institutions training together a model without sharing patients' data.

4. **Client's statefulness**: In a training loop, statefulness refers to the ability of a client to keep optimizer states, training statics, or any kind of inter-round information that can be used in the next round.
   - Stateful: Clients can keep inter-round information. This is frequently associated with cross-silo scenarios, as clients are expected to stay in the training loop from beginning to end.
   - Stateless: Clients are not expected to hold any tracking information from previous rounds. This can be frequently associated with cross-device scenarios due to its characteristic of not expecting clients to participate in more than one round.

5. **Data distribution**: Specifies how the data is distributed across the clients, affecting the learning process and model performance.

- IID (Independent and Identically Distributed): The data across clients is assumed to have an independent and similar distribution. This case is closer to the training performed on classical machine learning algorithms, as described in Section 2.1.4. Datasets for the centralized machine learning case are built in a way that makes them IID, as unbalanced classes and examples tend to impact training performance negatively.
- Non-IID: The data distribution varies significantly between clients, reflecting real-world scenarios where clients may have vastly different data, making the training process more complex. As this case represents a more real approach for federated learning, it is common to artificially create non-IID datasets from classical ones, like CIFAR-10; this can be done in two main ways: First, by using the Latent Dirichlet Allocation (LDA) [80] method. The idea here is to use a Multinomial distribution, where the number of experiments corresponds to the number of clients, the number of events represents the number of classes, and the event probabilities are defined by a Dirichlet distribution controlled by a concentration parameter. This allows each client to be assigned different examples from each class. The concentration parameter can be adjusted to create a range of distributions: a complete IID distribution when the parameter approaches infinity or a fully non-IID scenario when the parameter approaches zero, where each client receives data from only one class. Second, another standard method is to partition the data based on classes, where each client receives data from only a subset of classes [148]. For example, one client might only have access to samples from classes 1 and 5, while another might have access to samples from classes 2 and 8. This method creates non-IID partitions by ensuring that different clients have access to other sets of classes. However, both methods share the disadvantage of creating non-IID scenarios with an equal number of examples per client, limiting the simulation's realism.

6. **Data partition**: Describes the relationship of data partition between clients. This classification serves as study cases for possible real-world scenarios where clients' data have some intrinsic relationship related to the task and the client's application domain. However, it's important to note that, by definition, federated learning assumes no prior knowledge of clients' data.

    - Horizontal (sample-based): Clients' data partitions represent the same feature spaces, but they potentially have different data points. This means that clients have collected different examples, but the features that describe each point are the same. This is, for example, the case for a set of clients to which a certain partition of the CIFAR-10 dataset was attributed; every data point is represented by an RGB image, but each client has different images.
    - Vertical (feature-based) [173]: Each client uses different features to describe the same data point. For example, a client possesses a subset of the CIFAR-10, while another client possesses, for the same subset, its text description. Same data points, different feature spaces.

The previous categories define aspects that are particular to the federated scenario, and on top of those, the machine learning algorithm, model definitions, training optimizations, and hyperparameters still need to be defined. This work centers its attention on deep learning problems in image classification tasks, with CNN models being trained in a supervised learning fashion, following a centralized orchestration with synchronous updates, stateless cross-device clients for both IID and non-IID cases, and horizontal

data partition.

2.3.2 The Training Loop and Federated Averaging

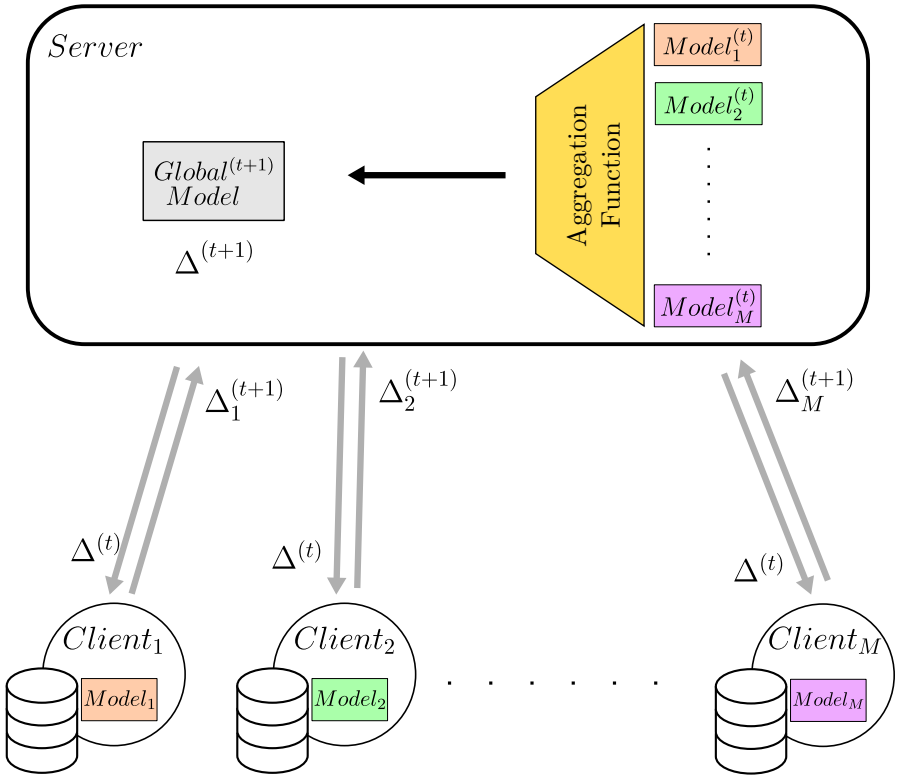


Figure 14 – Training loop for one round of federated learning.

A typical round of federated learning involves several key steps: client sampling, model download (from server to client), local training, model update upload (from client to server), and knowledge aggregation. A federated learning loop is shown in Figure 14. The server sends the current model, $\Delta^{(t)}$ to the clients, where they train their model and send back an update information. The update can be the gradients, model weights, logits, or any kind of information resultants from the training. Here, we consider the case where the client sends the model weights $\Delta^{(t+1)}$ back to the server. Finally, the server uses an aggregation function to fuse all updates into the global model for the next round.

In cross-device federated learning, as the number of users participating in a training round increases, it becomes impractical to orchestrate thousands or even millions of clients in every round. To ensure scalability in such scenarios, client sampling is essential. Another significant challenge with having a large number of clients per round is ensuring their availability and responsiveness. Clients must complete training and transmit their results to the server within a set timeframe. However, the varying computing capacities of clients can lead to a problem of straggling clients. This is less of a problem when the cross-silo scenario is considered, as the pool of clients is smaller, and clients are expected to be reliable.

Regarding knowledge aggregation, the most widely adopted baseline is Federated Averaging (FedAvg) [117]. We use it as background to define the general objective of federated learning. FedAvg operates by sampling a subset M of a pool of clients U in each round to train a model with parameters $\mathcal{W}$ for a specified number of local epochs, with a local batch size of g. Each client has its dataset, d_m, of size $|d_m|$, and the total size of the

datasets of the participating clients in a round is denoted $|D|$. Each client $m \in M$, seeks to find the parameters $\mathcal{W}_m$ that minimize its local loss $\mathcal{L}_m(\mathcal{W}_m) = E[\mathcal{L}(X_m, y_m, \mathcal{W}_m)]$, for a set of examples X and labels y, and the loss function $\mathcal{L}(.)$. The general objective of FedAvg is then to find a global $\mathcal{W}$ that minimizes the Equation 7. When this process is iterating for a certain number of rounds R, the final result is expected to increase the individual performance of each client [153], without the need for data sharing.

$$\min_{\mathcal{W}} V(\mathcal{W}) = \sum_{m=1}^{M} \frac{|d_m|}{|D|} \mathcal{L}_m(\mathcal{W}_m) \tag{7}$$

To exemplify a federated learning training using the FedAvg algorithm, next, we present an experiment with the following setting :

- **Aggregation algorithm**: Federated Averaging
- **Model**: ResNet-20
- **Number of Clients**: 20
- **Number of Rounds**: 100
- **Clients per round**: 4
- **Data partition**: IID and non-IID using the LDA partition with a concentration parameter of 0.1
- **Clients local epochs per round**: 10
- **Clients batch size per round**: 32
- **Dataset**: CIFAR-10

The simulation is performed with the Flower [17] framework and Pytorch [134]. We simulate 20 clients during 100 rounds, training a ResNet-20 model. Two types of experiments are presented, the first with an IID distribution and the second with a non-IID distribution. Figures 15 and 16 show the data distribution per client for the IID and non-IID cases, respectively.

In Figure 17, the impact of the non-IID partition on the training loop can be seen. FedAvg can attain around 92% of accuracy in the IID setting, while in the non-IID, the same algorithm, model, and regime achieve only 81%. The following Section 2.3.3 discusses further the data distribution and open problems in the federated learning framework.

2.3.3 Open Challenges

Once the different elements and characteristics of federated learning are identified, it becomes crucial to pinpoint its main open challenges [89]. This motivates the various research topics that aim to advance the field. The following list enumerates such challenges, briefly describing their context and providing some literature references that can be followed for a more complete view of the problems at hand. The challenges presented are a direct consequence of the design choice of integrating different parties into the training loop. Especially for the cross-device setting, it is not directly possible to control what clients can do with their training cycle, data, and model updates, which imposes a series of considerations for federated learning to be applicable in real-world applications.

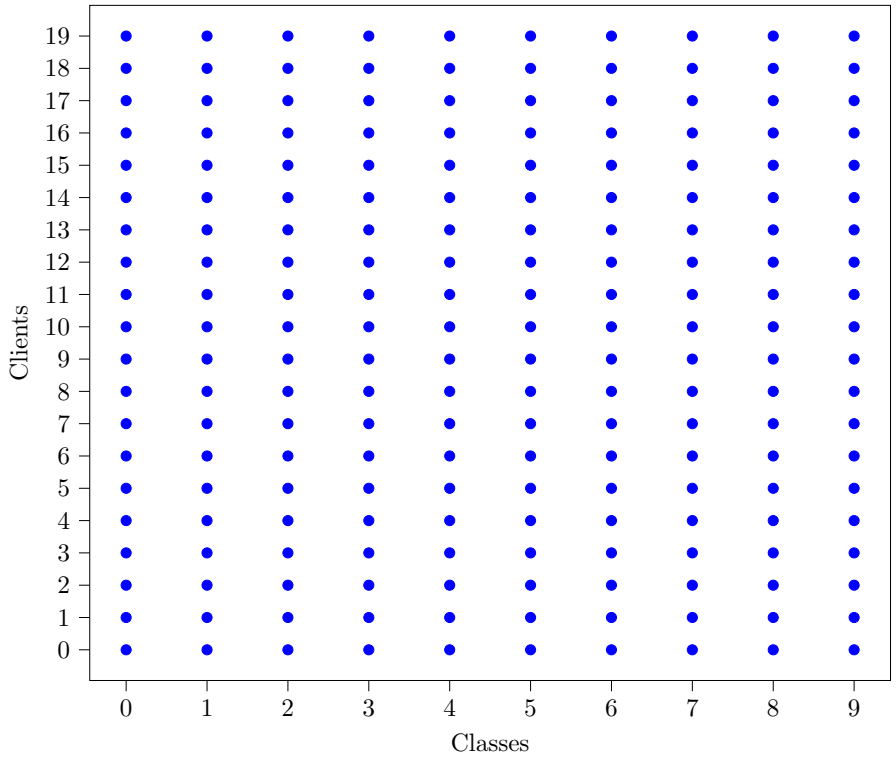


Figure 15 – IID partition of CIFAR-10, over 20 clients.

2.3.3.1 Privacy (data breach)

As exhibited in [117], although federated learning increases the attainable privacy level of a client, in its default algorithm, it is still possible to recover enough information from the model updates and/or gradients [82] to be able to reconstruct clients' local data. An example of an attack consists of tracking each client's individual model updates to reconstruct text data [66]. One research direction to prevent these attacks is the development of homomorphic encryption [179] techniques, where the entire training loop could be encrypted to ensure a higher level of privacy. Outside federated learning, one option for privacy concerns is the use of techniques that are not dependent on outside intervention. In Chapter 5, we provide an example of a solution based on few-shot learning depending only on a device's local data.

2.3.3.2 Distributed Optimization

The IID and non-IID data distributions generate the main topic of research in the field, as different optimization methods try to compensate for the non-IID effect in clients to close, first, the gap to IID distributions and, later, to the centralized training setup. One of the main effects of the non-IID is the client drift effect [155]. Due to clients' particular data distribution and the client selection/sampling, individual model updates can provoke a drift to the global model update, pushing the update direction closer to a solution that fits their particular distribution.

Client drifting is observed in FedAvg, as the method struggles to converge to the same level of performance between IID and non-IID distributions of the same dataset. This challenge, in general, has been addressed in various forms in the literature. Improving upon the FedAvg algorithm, FedAvg with Momentum (FedAvgM) [80] adds a gradient-like optimization to the server by using the mean of clients' update, controlled by a

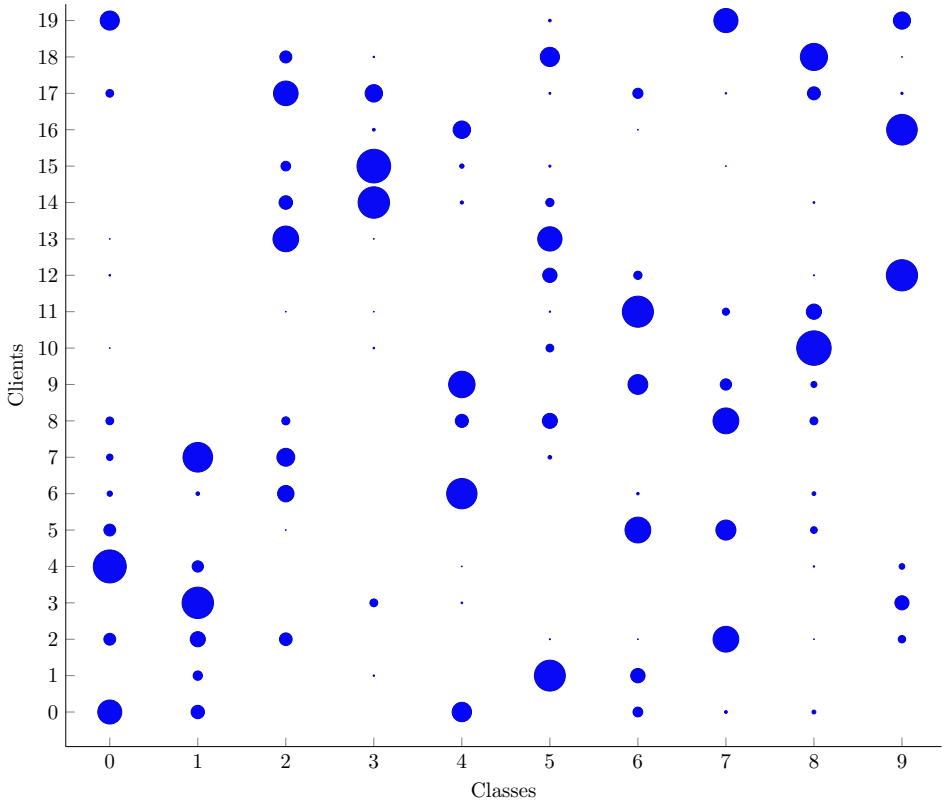


Figure 16 – Non-IID partition of CIFAR-10, over 20 clients, with a concentration of 0.1.

scaling parameter, to aggregate the models. Federated Proximal FedProx [107] adds a proximal term to clients' local loss to act as a regularization to prevent clients' updates from deviating significantly from the global model. Stochastic Controlled Averaging for Federated Learning (SCAFFOLD) [91] introduces two corrective variables, one globally updated and one locally updated, which are added to clients' local loss. These variables help reduce the variance in client updates by providing a consistent "guide" that aligns the direction of updates across all clients. Federated Optimization (FedOpt) [140] unifies the FedAvg and FedAvgM algorithms, proposing a general method that introduces an optimization step in the server, allowing the integration of classical optimizers from the centralized training setup to the knowledge aggregation step. Federated Distillation Fusion (FedDF) [111] proposes to use an additional dataset on the server to perform ensemble knowledge distillation [10] as a further optimization step to aggregate model updates. Model Contrastive Federated Learning (MOON) [105] reformulates clients' local loss as a contrastive learning [34] problem by using the representations, the output of the backbone [1] of the current global model as a guide for the local client model. The idea is that a client's model representations should not deviate from the global model in between rounds. Federated Extrapolated Averaging (FedExP) [87] improves the FedOpt method by proposing an adaptive method to determine the server step size, or learning rate, at each round.

All presented methods aim to increase performance and improve convergence, directly impacting the number of rounds a model needs to be trained. Reducing the number of communication rounds reduces the probability of finding a straggling client and the

1. The term backbones refers to the part extracting the features of an input, being the layers just before the classification head. In CNNs, they refer to the convolution blocks before the block of linear layers.

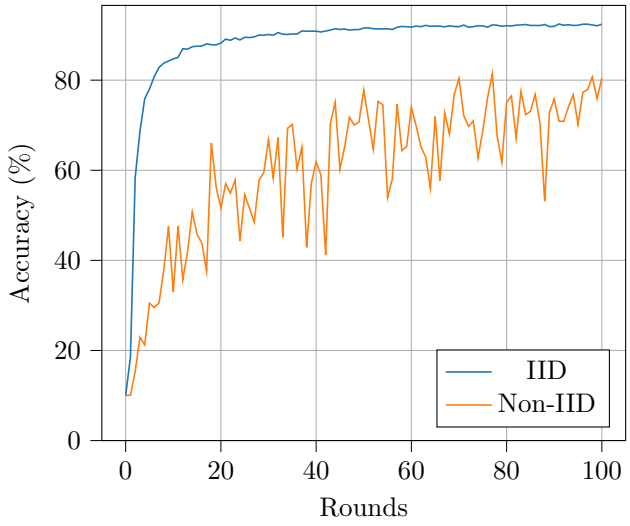


Figure 17 – The IID and Non-IID accuracy evolution along 100 rounds of communication for 20 clients.

energy consumption of both the server and the clients.

Another line of research to close the gap and improve convergence is using pre-trained models as an initial state for the training loop. Previous works have investigated the impact of pre-training in the federated learning framework [154, 32, 129], demonstrating that, similar to the centralized setting, pre-training serves as a suitable initialization method for federated learning. This is a crucial design decision in federated learning [117], which improves its robustness, generalization, and convergence. The main limitation of its general adoption is related to tasks or fields where access to data limits the availability of such pre-trained models. More about this is discussed in Section 4.5.

2.3.3.3 Security

With the same reasoning as in the privacy challenge, clients control their data and the model update. This leaves a backdoor for possible attacks to the training loop. There are mainly two forms of attack; the first is called "data poisoning" [162], in which the attacker manipulates the local data of one or more clients, manipulating the global model performance. Another form of attack called "model poisoning" [53], consists of modifying the model update sent to the server, resulting in similar types of degradation as data poisoning. As clients' data and model updates are naturally heterogeneous, detecting anomalies or tampering is challenging for secure knowledge aggregation. The literature has proposed robust aggregation techniques focused on eliminating outliers, which could possibly degrade the global model performance. Differential privacy [169] was also suggested as a method to ensure that data and models remain unaltered by introducing controlled noise, which can help detect and prevent tampering.

2.3.3.4 Fairness

Fairness seeks to have a model that performs the same across all participants and simultaneously represents the different knowledge present in each one. During training, clients can have different dataset sizes or not always be available to participate, leading to clients with larger datasets and present exerting more influence on the global model. This issue may be tackled by using fair aggregation methods [118] that try to generate a global model considering the frequency, importance, or variance of the

client's update. An additional well-adopted technique is to dissociate the global model from the local models by creating personalized models at each client level. This can be achieved through additional local fine-tuning steps [106], where the global model is further trained to fit the local data better. Another approach involves introducing personalized layers [153]. In this method, only a subset of the client's model is updated and sent to the server, while the local portion of the model remains on the client. This allows one part of the model to capture global features while the other part focuses on local features.

2.3.3.5 Computing and Communication

Moving training from the centralized setup to a cross-device setting involving heterogeneous devices introduces significant challenges for low-power devices. Devices often have limited hardware capabilities and varying energy constraints, making executing a Floating Point Operations (FLOPs) and memory-hungry algorithm such as deep learning model training challenging. Furthermore, frequent communication between devices and server consumes considerable bandwidth and energy, which can drain batteries and strain network resources, especially in environments with unreliable connectivity. Improving the convergence of federated learning, consequently reducing the number of rounds, is a way to tackle this problem. However, often, these algorithms add extra computation steps to clients' algorithms, becoming a trade-off between convergence, overhead, and energy.

In [137], authors have conducted a study on carbon emissions and energy cost on a federated learning training. The training energy consumption was calculated based on the CPU and GPU power usage, considering the total wall clock training time, the number of communication rounds, and different hardware profiles. For communication, energy consumption was modeled using factors such as the size of the model, upload/download speeds, router energy usage, and idle time between communications. To quantify the impact of federated learning, the authors compared centralized and federated training using different hardware configurations. The centralized setup used a server with a 240 W TDP CPU and a 250 W TDP GPU, while the federated setup employed two types of embedded GPU devices with power limits of 7.5 W and 10 W.

For example, their experiments on image classification tasks used a ResNet-18 for the CIFAR-10 and ImageNet datasets. For CIFAR-10, they simulated a system with 500 clients. For ImageNet, they used 100 clients. In both cases, 10 clients were sampled per round. The experimental protocol defined a target accuracy of 70% for CIFAR-10 and 50% for ImageNet to determine the end of training, enabling a consistent comparison between the centralized and federated setups. The following numbers reflect the energy consumption when using the FedAvg strategy.

- CIFAR-10: In the centralized setting, 2.7 Wh was consumed to reach the target accuracy. For federated learning, the total energy consumption across all clients was significantly higher, with 30.4 Wh. The per-client energy was 3.04 Wh on average.
- ImageNet: Centralized training consumed 971 Wh to reach the 50% accuracy. In the federated setting, the total energy was higher at 2697.5 Wh, where each client consumed 269.5 Wh on average.

Finally, their study highlights the impact of dataset size and task complexity. For the ImageNet dataset, communication costs represent 0.7% of the total energy while repre-

senting 96% for CIFAR-10. This can be explained by the difference in dataset complexity and size, where ImageNet clients must train a 24× bigger dataset.

In Section 3.2, we discuss this problem further to show how model compression techniques from the centralized setting can help straggling clients.

2.3.4 Federated Applications

As discussed so far, federated learning was designed explicitly for privacy-aware applications, where machine learning models are traditionally trained in centralized settings with sensible data. This approach has enabled the creation of collaborative real-world training efforts, enhancing both model performance and representativity and improving the overall user experience. Next, two real-world case studies demonstrate how federated learning is applied.

- Gboard: The Android operating system's first real-world application of federated learning was demonstrated with the Google keyboard [69]. This application employs a variant of the Long Short-Term Memory (LSTM) model [62] to predict the next word in a phrase based on the current context. Researchers deployed the federated learning algorithm on selected users' smartphones to enhance the virtual keyboard model. The selected clients to participate in each round needed to be connected to the Wi-Fi, be in an idle state, and be connected to a power source to avoid any impact on the phone's usage. This setup, a typical cross-device scenario, involved approximately 100-500 clients per training round. It is important to notice that, despite having a stable connection and an energy source, the training was conducted on devices with limited computing power. Also, from the server's perspective, each round represented the combined bandwidth of hundreds of clients uploading and downloading model updates. These examples highlight the need for strategies to reduce communication and computation overhead while addressing the fairness challenge in federated learning.
- Healthcare: Due to the confidential nature of medical data, collaborations between institutions have become more feasible through federated learning [158]. For example, a previous work [147] has demonstrated the feasibility of training an image segmentation model based on the UNet architecture [142] for brain tumor segmentation. In another instance, the authors of [44] developed a model called the "EMR CXR AI Model", which was trained in collaboration with 20 institutions worldwide using actual clinical data to assist with patient triage during the COVID-19 pandemic. This collaborative model outperformed locally trained models due to the increased size and quality of the dataset. The "MELLODDY" platform [131] represents a collaboration among multiple industrial actors, including pharmaceutical companies, start-ups, and academic research labs, to create a federated model for drug discovery in a privacy-preserving setting. The platform utilizes a multi-task model [31], capable of classification, regression, and a hybrid mode, based on a MLP architecture called SparseChem [8]. This represents a classical cross-silo scenario where the large size of medical datasets and the need for frequent model updates can significantly strain network bandwidth and computational resources. Institutions can collaborate more efficiently by minimizing communication overhead, allowing for faster model convergence and reducing latency, which is crucial in time-sensitive applications such as patient triage or drug discovery.

2.4 Recapitulation

This chapter briefly presented the image classification task, along with its various architectures, datasets, and training routines. The introduction to the distributed learning setting provided a motivation for the federated learning framework. We then discussed the key elements of the federated learning paradigm. Finally, in Section 2.3.3, we introduced the main research directions in the field, which motivate the next chapter. The upcoming chapter will address the communication and computation challenges by exploring compression techniques for deep learning models.

Chapter 3

Compressing the Federation

Contents

This chapter extends the discussion on the communication and computation challenges introduced in Section 2. Moreover, we focus on communication costs and how they can be minimized. Solutions to these problems can be approached at two levels. The first is at the system level, where data compression techniques are applied. These methods are typically in place and depend heavily on the communication protocol or system used. The second level involves the algorithm itself; compression can be effectively applied with a sufficient understanding of the information being transmitted. This work explores how compression techniques commonly used in centralized deep learning can also impact federated learning. The chapter begins with a background review of classic compression techniques, such as quantization and pruning, and extends the discussion to other methods. Following this, we introduce how these techniques have previously been applied in federated learning. This section motivates our main contributions, which build upon and extend the existing literature in federated learning.

3.1 Squeezing Every Bit

Deep learning algorithms are inherently designed with overparameterized models. As discussed in Section 2.1.3, the universal approximation theorem does not specify the required size of a network for it to be a good approximation. So empirically, scaling up the network's size has enabled models to handle more complex problems. Therefore, compression of deep learning models has become a long-term research goal. Model compression goes beyond reducing overparameterization; it also addresses physical and computational constraints. As models grow, their storage, computing power, and memory requirements also increase. Pruning and quantization directly address this growth by reducing excess elements in the network and simplifying the model. In this section, we introduce these techniques, along with more recent methods.

3.1.1 Quantization Methods

In general, quantization refers to constraining the possible representation states of information to a more limited subset. In the case of signal processing algorithms, it relates to taking the data format used to encode information and "quantize" it to a data format with fewer possible representation states. From an algorithm point of view, restricting the representation space will incur a direct loss of information, directly impacting the algorithm's performance. For example, passing to a smaller representation format frequently results in an accuracy drop in a deep learning model.

Before describing data formats, typical quantization schemes, and related methods, it is vital to understand the motivation behind quantization and its impact on hardware. Deep learning layers, such as convolution and linear layers, are the most resource-intensive in deep learning models, in terms of the number of parameters and operations to be executed. These operations can be further described as a series of Multiply-Accumulate (MAC) operations and described as grouped matrix-multiplications. From a system perspective, these operations are executed within an arithmetic computing unit, while the inputs and weights are stored in a nearby memory unit. Figure 18 illustrates a simplified view of this system. During a typical execution loop, the parameters are first transferred to the computing unit, followed by the necessary inputs. Once the execution is complete, the results are returned to memory.

The data format has three implications for this loop. First, the larger the data format, the more memory is required to store all the weights and inputs. Second, the size of the data format affects memory transfer, as larger formats require more effort in terms

of latency and energy. Finally, the physical size of the MAC operators, as well as the execution time and energy consumption, are directly influenced by the complexity of the data format. Quantization comes as a way to address these issues by enabling more efficient execution in terms of latency, energy, and memory space.

In this discussion, we are adopting a simplified view of the computing paradigm in deep learning. The problem has been reduced to the key steps of transferring data from memory to computing units, processing it, and then storing the results. In reality, this process heavily depends on the specific platform in use. For systems that use Application-Specific Integrated Circuits (ASICs) or FPGAs, data can be stored near the computing units, either in registers or on-chip memories or in off-chip memories, with the latter incurring higher access costs. In more traditional setups involving CPUs and GPUs, the GPU acts as the computing unit with its own memory space, where both the model and dataset are stored. Meanwhile, the main application runs on the CPU, and multiple levels of memory transfers are needed—from disk, to CPU memory, to GPU memory, and ultimately to the computing core.

Regardless of the platform, the memory hierarchy, or the organization of computing units, data formats have a significant impact on data transfers and computational complexity to different degrees.

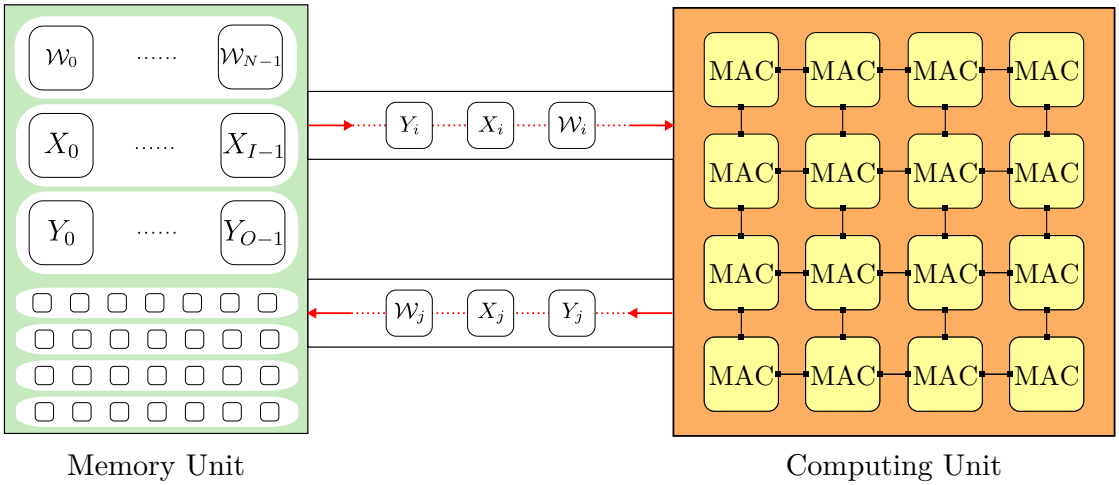


Figure 18 – A simplistic system view of a computing model for a deep learning layer. $\mathcal{W}$ are the model parameters, X the inputs and Y the outputs. Here, all the parameters and data are stored in the memory unit, and a data bus symbolizes the exchanges with the computing unit. The represented computing unit is a simple representation, where, in practice, the unit would possess other computing cores besides MAC units and more elaborated memory hierarchies.

In real-world applications, the most commonly adopted formats are smaller, more simplified versions of Floating-Point (FP) and integer formats. In the following parts, we present the common formats accepted by modern frameworks and hardware. Other formats are discussed in a later part.

3.1.1.1 Floating Points

Traditionally, deep learning models have adopted single-precision floating-point numbers containing 32 bits, FP32. Table 1 shows the range and precision of the FP32 format [1]. With this starting point, many propositions were made to improve the hardware

1. An extended review of the number of floating-points can be found in [123]

efficiency, taking FP32 as the baseline.

In the case of floating-points, smaller word sizes can be adopted to increase hardware efficiency in deep learning applications. This is achieved with a compromise between the dynamic range (min. and max. values) and the precision through resizing the exponent and mantissa fields. Additionally, the standard FP32 format, as defined by [85], includes features such as subnormals, rounding modes, and NaN, which can be simplified. These insights have driven the development of various trade-offs within the floating-point family, resulting in configurations with 16- and 8-bit representations.

Table 1 – Some notable floating-point flavors used in deep learning applications. The dynamic range represents the absolute minimum and maximum values the data representation can assume. From the general formulation of floating-point format, the maximum value happens when the exponent is at its maximum, and the mantissa is full of 1s, $Value_{Max} = 2^{2^E - 2^{E-1} - 3} \times (2 - 2^{-M_{antissa}})$. While the minimum value happens when the exponent and mantissa are at their minimum values, $Value_{Min} = 2^{2-2^{E-1}}$. Precision represents the number of significant digits obtained with $log_{10}(2^{M_{antissa}})$. E and $M_{antissa}$ represent the number of exponent and mantissa bits, respectively.

Floating-point flavors	Dynamic Range	Precision	Exponent	Mantissa
IEEE 754 - FP32 [85]	1.18×10^{-38} to 3.40×10^{38}	7 to 8	8	23
IEEE 754 - FP16 [85]	6.10×10^{-5} to 6.55×10^{4}	4	5	10
Bfloat16 [90]	1.18×10^{-38} to 3.39×10^{38}	3	8	7
DLFloat16 [3]	9.31×10^{-10} to 4.29×10^{9}	3 to 4	6	9
FP8 - E4M3 [119]	1.56×10^{-2} to 2.40×10^{2}	1 to 2	4	3
FP8 - E5M2 [119]	6.10×10^{-5} to 5.73×10^{4}	1 to 2	5	2

Subnormal numbers were omitted when calculating dynamic range and precision, as is only used for the IEEE 754 defined formats. Formats dedicated to deep learning applications tend to drop the support for subnormals as their handling incurs additional computational overhead.

Table 1 exhibits some typical configurations of floating-points used in deep learning frameworks, such as Pytorch, TensorFlow or JAX. Dynamic range and precision trade-offs are in place for optimal utilization when considering whether the quantization scheme is used for inference or training [119, 152].

3.1.1.2 Integer methods

Regarding area and energy consumption, integers have the upper hand compared to floating-points [79]. The main limitation of integers is their limited fractional representation capability. This limitation can be circumvented by applying fixed point notation [67], dividing the integer word into integer and fractional parts with an implied binary point. An alternative approach relies on a combination of integer and floating-point format, called affine quantization. Here, we explore the second approach method due to its prevalent use in deep learning deployment pipelines.

As a first approach, the objective of the integer quantization is to be able to map a specific matrix W of elements $w^{(i,j)}$, where $i \in \{0, ..., I-1\}$ and $j \in \{0, ..., O-1\}$, to a matrix $W_q \in \mathbb{Z}^{I \times O}$, with elements $w_q^{(i,j)}$, and a scale factor $s_q \in \mathbb{R}^O$, through a quantization function $Q(\cdot)$.

As an example, consider a 4x4 matrix $W \in \mathbb{R}^{4\times4}$. The quantization function $Q(\cdot)$ that maps the values in W to a pair (s_q, W_q), where $s_q \in \mathbb{R}^4$ and $W_q \in \mathbb{Z}^{4\times4}$. This example is in Figure 19, where a randomly generated matrix W is quantized to a pair (s_q, W_q). This

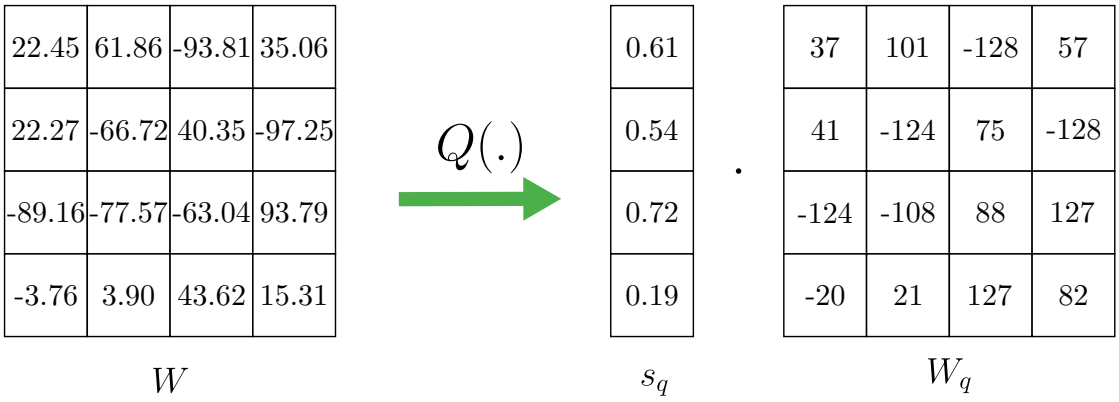


Figure 19 – A toy example of a quantization process, where the values of W_q are limited to signed int8.

approach is called symmetric quantization, as opposed to the other possible approach known as asymmetric quantization.

The asymmetric quantization expression includes a bias point, called the zero point $z_q \in \mathbb{Z}^{I \times O}$, which allows a real representation of the zero value, making the quantization scheme more flexible to different values distributions. Figure 20 recreates the previous example but includes the zero point. This time, the quantization function $Q(.)$ maps the values in W to the triplet (s_q, z_q, W_q).

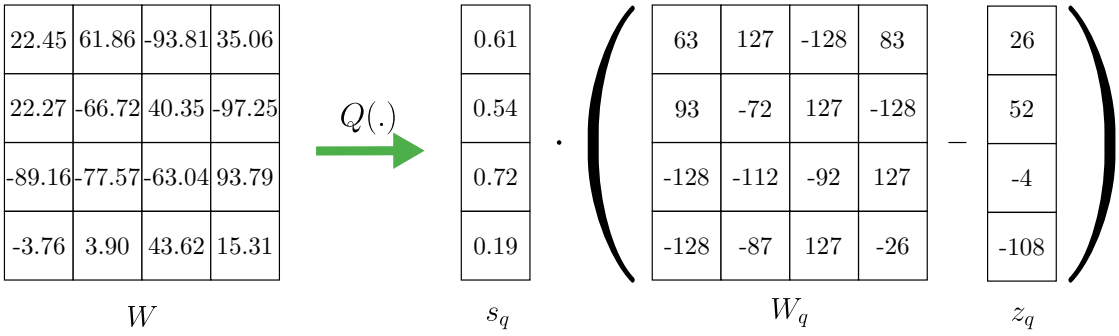


Figure 20 – A toy example of a quantization process for the asymmetric quantization scheme.

In both approaches, the values of W_q can be either unsigned or signed integers. Unsigned integers are represented by values in the range $[0, 2^b - 1]$, while signed integers are represented by $[-2^{b-1}, 2^{b-1} - 1]$. The parameter b denotes the number of bits representing the integer, defining the quantization limits. Figures19 and 20 consider a signed 8-bit quantization scheme, which means that the elements $w_q^{(i,j)}$ can take values between $[-128, 127]$.

The general expressions for $Q(\cdot)$ can be expressed as in Equations 8 and 9, which correspond to the symmetric and asymmetric cases, respectively. The terms min_q and max_q refer to the minimum and maximum values of the data format for $w_q^{(i,j)}$, and $\lfloor \cdot \rceil$ denotes the round-to-nearest operator. Given that $Q(\cdot)$ uniformly maps the possible values of $w_q^{(i,j)}$, a linear system of equations can be applied to determine the values of s_q and z_q. These values are derived from Equation 9, based on the assumption that the mapping of $(\max(W), \min(W))$ should correspond to (max_q, min_q). Additionally, this mapping is performed while ignoring the clipping effect when the representation capacity is exceeded. Consequently, the scale factor and zero point can typically be

defined as $s_q = \frac{\max(W)-\min(W)}{max_q - min_q}$ and $z_q = min_q - \lfloor s_q \cdot \min(W) \rceil$, respectively. The presented equations assume the use of the maximum and minimum values over W; however, subsets of W are often used. Quantizing weights in groups presents a performance trade-off. When the granularity of quantization decreases, more operations are performed in floating-point, due to the scale factor. As granularity increases, the distribution of values becomes more diverse, leading to outliers, which can result in higher quantization errors [170]. For instance, in linear layers, quantization is often applied per column or row to handle the weights. In convolutional layers, the most common approach is to apply quantization per-channel. Dequantizating W involves inverting the quantization process, as expressed by Equations 10 and 11 for the symmetric and asymmetric cases.

$$Q(W, s_q, min_q, max_q) = clip(\lfloor \frac{w_q}{s_q} \rceil, min_q, max_q) \tag{8}$$

$$Q(W, s_q, z_q, min_q, max_q) = clip(\lfloor \frac{w_q}{s_q} \rceil + z_q, min_q, max_q) \tag{9}$$

$$DQ(W_q, s_q) = s_q \cdot (W_q) \tag{10}$$

$$DQ(W_q, s_q, z_q) = s_q \cdot (W_q - z_q) \tag{11}$$

The quantization/dequantization process is inherently lossy, as it involves mapping from a much larger representation space to a smaller one, resulting in quantization errors. For the examples shown in Figures 19 and 20, the mean errors are 4.26 and 0.13, respectively. Much of the quantization literature for deep learning models focuses on finding better methods to determine the values of s_q and z_q [52, 49]. To mitigate the quantization error, various optimization techniques and subsets of W_l, to which quantization is applied, have been proposed.

As the quantization process is applied to data, it can be used for the activations, represented by the model input and the output of each layer in the model. Regardless of the type of layer, weight-only quantization methods are more accessible to optimize than weight-activation schemes [126]. That's because model parameters, once trained, are fixed, and any outliers or particularities can be addressed and studied. The activation distributions are highly dependent on the input and their interaction with the model parameters, leading to more unstable quantization schemes [9].

3.1.2 When to quantize ?

When quantizing a model, in addition to the different parameters of quantization, such as the scale factor, zero point, and data size, one needs to define when to quantize. Commonly, there are two moments when quantization is applied: after the model training, called Post-Training Quantization (PTQ), and during the training loop, called Quantization-Aware Training (QAT). Both choices have their trade-offs and particularities.

PTQ is typically more straightforward and faster to obtain a quantized model, as it involves applying quantization to a pre-trained model without requiring additional full training. However, the resulting quantization error is bigger than in QAT, leading to a worse accuracy degradation in low-bit precision than in QAT. The reason why is

that QAT incorporates quantization during the training process, allowing the model to compensate for the quantization errors and perform better at low bit precision [109]. Although QAT is more computationally demanding and time-consuming, it typically results in less accuracy degradation, as the model can fine-tune its weights and activations to mitigate the effects of quantization better.

Taking the weights-only quantization scheme for CNNs as an example, we next describe the common PTQ and QAT pipelines.

3.1.2.1 After training

We can describe a common PTQ pipeline in three steps,

(1) Calibration Dataset: PTQ methods usually use a subset of the dataset as a calibration dataset to interactively compensate for the quantization error. Data-free methods are also possible, as in ZeroQ [27], which uses the running statics of batch-norm layers in the model to generate a synthetic dataset for fine-tuning.

(2) Quantization: During this step, an optimization process is applied to different network elements to quantize the weights. Commonly, the quantization is done per layer, and inside each layer, sub-groups of weights are used to avoid large groups of widely different weights and magnitudes. For example, PACT [38] adds a learnable parameter to the clipping function in Equation 9, quantizing layer-wise and per channel. AdaRound [125] replaces the deterministic round-to-nearest operator with a stochastic one based on the distance between the value to be rounded and the neighboring quantized levels, quantizing in a layer-wise and per-channel manner.

(3) Fine-Tuning: Finally, the quantization is done interactively with the calibration set to reduce the performance difference between the floating-point and the quantized model.

3.1.2.2 During training

To integrate the quantization error into the learning process, QAT can be seen as a two-step process,

(1) Introduce quantization operators: As illustrated in Figure 21, in the QAT training loop, quantizer nodes are added to convert the weights of each layer into fake quantized weights. These quantizers produce weights still represented as floating-point values but are constrained to the possible quantized states, simulating quantization.

(2) Quantize weights: Various techniques and strategies can be used for quantizing the weights in the model. As seen in PTQ, it is expected to quantize different elements within each layer separately. Another common approach is to assign different precision levels to other layers, as they can exhibit varying behaviors, allowing one layer to compensate for the errors introduced by another. For example, HAWQ [49] applies different quantization precisions to each layer by using the Hessian matrix to assess the sensitivity of each layer to a specific precision level. Based on this information, HAWQ selects the optimal quantization precision for each layer in a per-channel, layer-wise manner. LSQ [52], on the other hand, incorporates the scale factor into the learning process, aiming to find the optimal value to compensate for quantization errors.

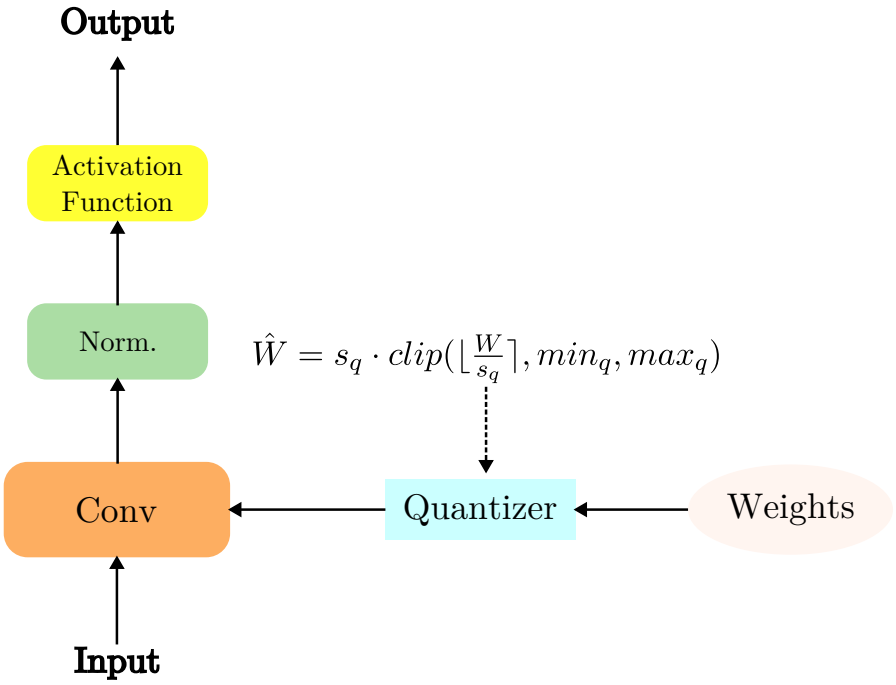


Figure 21 – An example of a QAT quantization operation on a sequence of convolution, normalization, and activation layers. Weights are quantized and dequantized to generate fake quantized weights, $\hat{W}$, with a symmetric quantization scheme. In practice, other quantization techniques can be applied.

3.1.3 Hardware implications

As a final motivation for adopting integer numbers, by having the weights in integer format and a floating-point scaling, two matrices, quantized in this way, can be multiplied with most of the operations done in integer. We demonstrate matrix multiplication of two quantized matrices, W_a and W_b.

Given the quantization Equation 9, for the asymmetric quantization scheme, we can express the quantized versions of W_a and W_b as W_{qa} and W_{qb}, respectively, as seen in Equation 12.

$$\begin{aligned} W_{qa} &= clip\left(\left\lfloor \frac{W_a}{s_{qa}} \right\rceil + z_{qa}, min_q, max_q\right) \\ W_{qb} &= clip\left(\left\lfloor \frac{W_b}{s_q} \right\rceil + z_{qb}, min_q, max_q\right) \end{aligned} \tag{12}$$

The sets (s_{qa}, z_{qa}) and (s_{qb}, z_{qb}) refer to the scale factor and the zero point of the matrices W_{qa} and W_{qb}. We assume that they are quantized to the same precision for simplicity. The dequantized form of their product, W_c, can be obtained with $W_c = W_{qa} \times W_{qb}$, which after rearranging the different terms, results in Equation 13, for which s_{qa}, s_{qb}, z_{qa} and z_{qb} are constant terms, with only the scale factor terms represented with floating-points. This reduced the amount of data that needed to be sent from memory to the computing unit and the number of operations performed in the floating-point.

$$W_c = s_{qa} \times s_{qb} \times (W_{qa} \times W_{qb} - z_{qa} \times z_{qb}) \tag{13}$$

3.1.4 Other formats

In this section, we have focused primarily on floating-point and integer-based data formats, which are the most commonly discussed in the literature due to widespread hardware support. Leading manufacturers of deep learning hardware accelerators, such as NVIDIA, AMD, ARM, Google, Microsoft, and Qualcomm, predominantly support integer and floating-point MAC units. This has naturally drove much of the research towards these formats, as they offer more practical, real-world applications on existing hardware.

However, other numerical formats also hold the potential for efficient deep-learning model execution. For example, POSITS [180], logarithmic-based formats [26], and block floating-point formats [43] have been explored as alternatives. These formats can offer advantages such as improved dynamic range or more efficient representation of certain data types, potentially leading to faster computation or lower power consumption.

They are especially interesting for reconfigurable platforms, like FPGAs, or dedicated circuits like ASICs. One can design optimized MAC operators for different data formats in such hardware. For example, a model quantized with less than 4-bit would operate inefficiently in commercial GPUs that usually has support for 8-bit operators. The design choice of quantization and data format highly depends on the deployment platform and hardware flexibility. In Chapter 5 and Appendix A, we discuss the use of dedicated frameworks and solutions to leverage the reconfigurability of such platforms for optimized deployments.

3.1.5 Pruning Methods

Pruning in neural networks aims to reduce the number of parameters or operations in a model while maintaining or minimally affecting its performance. The fundamental idea is that not all parameters within a model contribute equally, and some can be pruned to improve the model's computational efficiency. Pruning can lead to faster inference and reduced memory usage. This, in turn, lowers energy consumption, making it particularly useful for deploying models on hardware platforms that would otherwise be unable to run. This is shown in [103, 64], where pruning was applied to a CNN and an Large-Language Model (LLM) model to reduce their on-device deployment costs. The study of pruning and its application to deep learning models is important to explore the problem of model over-parameterization [77].

The following sections explore the different types of pruning, which elements are pruned, and the standard criteria behind pruning techniques.

3.1.5.1 Pruning Elements

Pruning techniques can be divided into two types: unstructured pruning and structured pruning. Each kind targets different structures that compose a model's architecture and has different final objectives.

- **Unstructured pruning** involves pruning individual weights within the model, where the values of the pruned weights are set to zero, thereby increasing the model's sparsity. By identifying which weights contribute less to the model's output, these weights can be zeroed out, leading to more sparse computations. For example, SWD [161] is a technique that adds a weight-decay regularization term to the loss function to penalize large magnitudes in weights, causing smaller weights to decay to zero naturally. However, exploiting sparse computations

is not always feasible because it requires the ability to detect zeros or specific sparse matrix algorithms, which can introduce overhead that negates the benefits of pruning. This issue can be mitigated on specialized hardware, such as FPGAs [114, 178, 150], where different hardware strategies can be employed to detect zeros or generate flexible architectures for sparse matrices. On more widely accessible hardware, such as NVIDIA GPUs, [124] has demonstrated a hardware-aware pruning method that takes into account the capabilities and architecture of the GPU, proposing to zero out parameters following the sparse pattern supported by the hardware.

- **Structured pruning** involves removing entire structures from the model, such as rows or columns in linear layers, filters in convolution layers, or even whole layers. This approach directly alters the network's architecture, leading to a more significant reduction in computational complexity and memory usage. For example, in CNNs, structured pruning might involve removing entire convolution filters, which reduces the number of feature maps and, consequently, the overall size of the features generated by a layer. In practical applications, [64] demonstrated the use of structured pruning to reduce an LLM from 6.4 billion parameters to 3 billion. During model training, learnable binary masks were added to the model architecture to simulate removing specific components based on the methods of [168, 171]. Once learned, these masks were applied to remove specific dimensions from the linear layers effectively. While structured pruning results in smaller models due to the removal of elements, it also leads to irregular computational patterns. [160] study revealed a series of dimensional problems arising from the irregular structures in Residual Neural Network (ResNet) architectures and proposed a methodology to address such issues, showing that some attention is necessary to compensate for possible architectural incoherence.

3.1.5.2 Pruning Criteria

Pruning criteria measure the weight's importance and contribution to the model output. In unstructured pruning, these criteria are commonly based on two metrics: weight and gradient magnitude.

- **Weight Magnitude:** One of the most simple and widely used criteria. The approach involves pruning weights whose magnitude/norm is the smallest. The first intuition is that small magnitude weights have a small contribution to the model output, but as studied by [159], small weights resulting from small gradients could still significantly impact the model. A more thorough investigation with the gradient magnitude could lead to false-positive identification. This criterion is directly applicable to unstructured pruning, where weights are typically ranked according to their magnitude and then pruned as proposed by [68]. Their pruning pipeline, as seen with quantization, consisted of pruning a model after training and then using a calibration dataset to recover the lost performance.
- **Gradient Magnitude (Saliency):** Another common criterion is based on the magnitude of the gradients. This method considers the sensitivity of the loss function to each parameter, calculated through gradients during backpropagation. Parameters with smaller gradients are deemed less important and are pruned. This criterion can be more sophisticated than weight magnitude, as it considers the role of each parameter in the network's learning process, potentially leading to more effective pruning strategies. For example, [70] measured the error introduced by removing a weight based on the Hessian matrix information of zeroing out a certain weight. Their method further formulated a way to compensate for the error

introduced by the pruned weight by changing the other weights in the same layer.

Structured pruning re-adapts the mentioned criteria to specific groups of elements. [103] adapts the idea of weight magnitude by computing the $\mathcal{L}_1$ norm of all the weights in a certain filter to determine if they should be pruned. Another alternative is using binary masks, as seen in Sparse Weight Activation Training (SWAT) [168]. A learnable binary mask operates by assigning a value of 0 or 1 to each component in the structure, for example, a row in a linear layer or a filter in a convolutional layer. A value of 0 "turns off" or removes the associated component, while a value of 1 keeps it. During training, masks are optimized along with the network's weights to find the group of weights/elements that can be turned off and minimize the loss term simultaneously. This allows the model to adjust its architecture based on the mask dynamically. Finally, [159] performed a thoughtful study on the impact of pruning on deep learning models, concluding that pruning is an effective tool to compress a family of models. However, it is a limited tool to obtain performant models for specific complexities. We perform a similar discussion for model compression in federated learning in Section 4.4.

3.1.6 Other Methods

In addition to the two approaches to compressing a model mentioned before, other propositions in the domain are listed here in a non-exhaustive list.

Mixing pruning and quantization: There is also the possibility of combining both approaches. In [21], authors studied the impact of quantization and pruning when deploying models on different hardware platforms such as FPGAs, GPUs, and specialized hardware for CNN models. Their study concludes that pruning and quantization are orthogonal techniques, where the order in which each technique is applied has a low impact on the final performance. As such, both techniques can be applied together to maximize deployment efficiency. As an alternative, [71] proposed a Quantization Aware Pruning (QAP) by integrating a pruning step in QAT, allowing the network also to learn to be robust to the introduced sparsity. They showed, as seen with the PTQ and QAT, that QAP had a better final accuracy dans pruning and quantizing a model after training, with an increase in training overhead.

Low-rank Decomposition: It involves decomposing the weights of certain layers in a trained model to lower-rank versions. The goal is to reduce the computational workload and memory required to store these weights by directly decreasing the number of weights. [46] applied this concept to linear and convolutional layers. Using a decomposition algorithm, they were able to produce low-rank matrices that require less computational cost than the original weights while maintaining a certain approximation error. In their implementation, they took the weights of a linear layer, denoted by $W_l \in \mathbb{R}^{I \times O}$, and applied Singular Value Decomposition (SVD) to generate the approximation $\tilde{W}_l = \tilde{U}\tilde{S}\tilde{V}^T$. Here, $\tilde{U} \in \mathbb{R}^{I \times r}$, $\tilde{V} \in \mathbb{R}^{r \times O}$, and $\tilde{S} \in \mathbb{R}^{r \times r}$ is a diagonal matrix of singular values. The parameter r controlled the rank of the approximation from the full SVD of W_l.

Knowledge distillation: Instead of proposing a method to bring changes to the model architecture, knowledge distillation [76] tries to transfer the knowledge of one model to another. In the case of model compression, the idea would be to use a larger pre-trained model, the teacher, to help train a smaller model, the student. In this way, the student, a presumably less capable model, learns how to imitate the outputs of the teacher model.

This imitation condition is modeled by adding a Kullback-Leibler (KL) divergence [97], which measures the dissimilarity between the probabilistic vector of teacher and student. The distillation loss, L_{KD}, can be formulated as in Equation 14. Where $\mathcal{L}_{CE}$ represents the cross-entropy loss, as in Equation 5, and $\mathcal{L}_{KL}$ is the divergence loss expressed in Equation 15. Terms $\hat{y}_t$, $\hat{y}_s$, and y are the teacher and student predictions and the ground truth, respectively. The tuning parameter α controls how much the student should learn from its prediction or the teacher's.

$$\mathcal{L}_{KD}(\hat{y}_t, \hat{y}_s, y) = \alpha \mathcal{L}_{CE}(\hat{y}_s, y) + (1 - \alpha)(\mathcal{L}_{KL}(\hat{y}_s, \hat{y}_t)) \tag{14}$$

$$\mathcal{L}_{KL}(\hat{y}_s, \hat{y}_t) = \sum_{i=1}^{n} \hat{y}_t^i log \frac{\hat{y}_t^i}{\hat{y}_s^i} \tag{15}$$

Neural Architecture Search (NAS): Automatizes the design of deep learning architectures by proposing a two-level optimization [183]. It explores a predefined search space of possible components, such as operators, layers, and activation functions, to find candidate models. Candidates are evaluated based on target metrics, such as latency, memory usage, or energy efficiency. Finally, the optimal candidate is evaluated on its performance for the specific task on which the model is being trained.

3.2 Communication and Computation Challenges

The algorithms detailed in Section 2.3.3.2 are interested in closing the gap between the distributed and centralized learning paradigms. In this sense, as the gap gets smaller, federated learning algorithms performance improves, so the number of rounds tends to be reduced. So, naturally, the advancement of the field is going in a direction that seeks to reduce the communication cost of its design. Focusing specifically on deep learning models, compressing models allows for a smaller exchange of messages and simpler models to be trained.

Especially in the context of cross-device federated learning, making it accessible for embedded devices is highly relevant. Embedded devices, which we define as specialized computer systems designed for a specific set of tasks with real-time constraints and limited computational power, are increasingly playing a crucial role in distributed learning. These devices often operate in environments with constrained memory, energy, and processing capabilities yet continuously generate vast amounts of valuable, real-world data. Examples include IoT sensors capturing environmental data, drones equipped with cameras for surveillance or delivery, wearable health monitoring devices, and even smartphones with various built-in sensors and AI capabilities. As such, building methods to facilitate the integration of embedded devices with deep learning is the main motivation for this thesis.

This section discusses the merging of centralized compression and federated learning. It reviews how quantization and pruning can be used to decrease communication costs in federated learning. Additionally, it addresses other methods and questions related to model heterogeneity, where each client has its own model.

3.2.1 Deep Learning Compression Techniques in Federated Learning

Federated learning's framework design requires clients to train their models for a number of local epochs and then communicate their training results to the server.

For all that is related to the computational limitation of devices, they may come from two factors. First, while training, clients executed a classical deep learning algorithm. In Section 2.1.4, we saw that during backpropagation, to find the best weights that minimize the prediction error, we calculate the gradient of the loss with Equation17. Notice that to perform this derivation, it is necessary to save the intermediate results, or activations, during inference, resulting in a memory overhead. Second, besides memory requirements, both inference and training require the execution of matrix operations with operands that scale in the function of the input; what comes in place is the implication described in Section 3.1.1 and 3.1.5.

The communication challenge comes from clients needing to communicate with an external entity due to their training. In this work, we mainly consider that clients transmit weights or information with equal dimensions as the weights. So, for a communication round, this represents a Total Communication Cost (TCC) as expressed in Equation 16. With $|\mathcal{W}|$ being the number of parameters in the model, Q_W being the number of bits adopted for each parameter, and R number of communication rounds.

$$TCC = 2R|\mathcal{W}|Q_W \tag{16}$$

The following parts of this section present works that have tackled the mentioned limitations, alleviating the federated training requirements. This is important, as discussed in Section 2.3.3, not only in terms of fairness or carbon footprint reduction [137] but also to reduce the cases of failing clients, stragglers, that cannot complete one communication round due to their limitations.

$$\frac{\partial \mathcal{L}}{\partial \mathcal{W}} = \frac{\partial \mathcal{L}}{\partial X_L} \cdot \frac{\partial X_L}{\partial X_{L-1}} \cdot ... \cdot \frac{\partial X_l}{\partial W_l} \tag{17}$$

3.2.2 Quantization in Federated Learning

Quantization was also explored in federated learning, as it was applied to centralized deep learning models. One of the initial limitations of quantization in federated learning is the immediate lack of calibration dataset. Models can be quantized in PTQ or QAT fashion, but as each client has its own dataset, applying the calibration step tends to lead to biased quantization. Otherwise, a calibration dataset would imply extra training epochs, which, from the point of view of the entire system, could imply higher energy costs.

For a client k, Federated Learning method with Periodic Averaging and Quantization (FedPAQ) [141] calculates the difference between the global model at timestep t and the client-updated model after local training. The difference, $\Delta_{Diff}^{t+1} = \Delta_k^{t+1} - \Delta^t$, is then quantized with a stochastic quantization. The quantization function, $Q_s(.)$, quantized normalized value of Δ_{Diff}^{t+1} between a series of predetermined discrete levels between 0 and 1. A stochastic rounding method randomly approximates an element Δ_{Diff}^{t+1} based on its distance to the neighboring quantization levels. The stochastic rounding guarantees an unbiased quantizer. This strategy is further combined with FedAvg. With the number of possible states reduced, the quantization states are encoded to reduce the number of bits sent.

LFL [7] used the same quantization function as FedPAQ but introduced a feedback quantization error signal. The server transmits $Q_s(\Delta_{Diff}^t)$. Clients perform the local training and quantization as seen with FedPAQ, but calculating the quantization error

$e_k^{t+1} = \Delta_{Diff}^{t+1} - Q_s(\Delta_{Diff}^{t+1})$. The error signal is added to the newly received global model in the subsequent round to compensate for the expected error it had in the previous round.

[65], first, introduced a regularization term to client loss, called Kurtosis regularization [36], to enforce a uniform weights distribution. The objective was to obtain models that were more robust to quantization on top of FedAvg strategy. Further, this solution was combined with a Multi-bit QAT algorithm for each client that selected the best bit-width to quantize clients based on hardware specifications or fixed values.

BHFL [175] starts from the principle that not all clients can train a floating-point model, proposing to let full integer weight models be trained. The pool of clients would be composed of clients in floating-point and different integer quantized modes, such as 4-, 8- and 16-bit. However, the aggregation of low-bitwidths and high-precision models results in a high degradation of the model's performance. To workaround this limitation, they proposed a neural network to act as a dequantizer on the server side to convert low-bit models to floating-point ones. The dequantizer was trained on a dataset constructed during training with the weights sent by clients. The model's loss had two terms, one based on the dequantization error and a distillation loss on a public independent dataset.

3.2.3 Pruning in Federated Learning

Most research lines in federated learning assume that all clients have the same architecture. FedAvg, for instance, considers that the server can average all the models to produce the global model. As such, the application of structured pruning is inviable, as pruning criteria are correlated to the task data. From a practical point-of-view, one could apply classic compression techniques, such as ZIP and Compressed Sparse Row (CSR), to compress models between communication rounds. However, such a scheme is inefficient as it does not leverage any particular data distribution in the model. This is the case where non-structure pruning becomes relevant to generate more sparse models. Although this alone does not reduce communication costs, sparser matrices can be explored more efficiently when in combination with classic compression algorithms. We explored this approach in Chapter 4. In terms of reducing computational cost, as stated in Section 3.1.5, it depends heavily on hardware and software support and the sparsity level.

PruneFL [88], based on weight magnitude, starts by selecting a client with more computational resources to prune the initial model. The model is sent back to the server once the selected client finds a suitable pruning rate. With the initial pruned model, other clients use it as a starting point. As the initial pruning rate may not be adequate for all clients due to data and resource differences, the pruning method is reapplied on a client basis. To reduce the communication cost, they proposed two strategies. The first was to use a binary mask to indicate which weights are not zero, transmitting only the non-zero values. The second strategy was to store only non-zero values, using 16-bit integers to store the index of the row/column of the values. During training, the best strategy is chosen based on the sparsification level.

FedDST [18] used a layer-wise sparse training to allow clients only to train a subnetwork. Clients train their models starting from a randomly generated binary mask that dictates which weights to be trained. Weight magnitude pruning is used after training to adapt the binary mask to each client better and clean smaller weights. However, weights with larger gradients that are zero in the mask are "regrown," allowing dynamic mask training. To reduce communication costs effectively, FedDST has clients

only periodically communicate their masks. On the server side, they implemented a non-zero weight aggregation method to consider clients' masks.

ZeroFL [136] proposes an efficient two-level method to improve federated learning by incorporating sparsity at both training and communication stages. In the first level, it adapts a sparse training approach based on the SWAT [168] method, where only the most significant weights (Top-K by magnitude) are used during forward and backward passes. As training progresses, ZeroFL learns binary masks that promote natural sparsity with the objective of reducing training time. In the second level, ZeroFL addresses communication overhead by transmitting only the Top-K significant weights from each client to the central server. This reduces data exchange and is further optimized by compressing the sparse models using the CSR format, which encodes non-zero values and their positions efficiently.

3.2.4 Alternative Compression Methods

Due to the particularities of the federated learning design, several papers have designed specific strategies to tackle its costs. The following works have been inspired by or have some connection to pruning, distillation, and NAS techniques.

FedDropout [28] was the first federated learning work to propose "partial training". The idea is that from the same architecture, one can sample sub-models inspired by dropout layers (Section 2.1.3.1). The server defines a dropout rate, dropping a fixed number of filters in convolution layers and rows/columns for linear layers. At each round, the layers to be dropped are randomly selected, and the submodel is sent to clients. Effectively, clients train different, smaller parts of the global model during the entire training frame. The paper also proposed that lossy compression methods be applied to the subset models to further the compression ratio.

FedRolex [5] improved on the submodel training of FedDropout. The work starts with the consideration that not all clients should be able to train the same model due to limited resources. They modified the submodel sampling to target client-specific constraints, as done by [47]. Unlike previous works on partial training, they proposed sampling each client's specific configuration in a rolling manner. For one client configuration, the server keeps track of which parts of the global model were sampled last, sampling newer parts. For instance, for a convolution layer with filters "a-b-c-d-e", a client would train filters "a-b-c-d" on round t and filters "b-c-d-e" on round $t+1$. The core idea was to have the global model be sampled and trained more evenly.

SLT [135] notice that previous works on partial training achieved great cost reductions but were sub-optimal. When comparing the submodel use with just training a smaller full model from the start, they found that submodel results were worse. They demonstrated that although reductions were achieved, the global model was sub-optimally trained. SLT introduces a method where clients train models with certain layers frozen to reduce communication overhead. A "frozen" model means that specific layer weights are not updated, so clients don't need to send these weights. In SLT, the model layers are separated into three types: fully trained, partially frozen, and fully frozen. Fully trained layers train all their weights, while partially frozen layers have only certain portions of the weights updated. Training begins with the initial layers being fully trained and the rest of the model being partially frozen layers. Over time, the fully trained layers are progressively frozen, and the frozen portions of the partially frozen layers are "unfrozen". This technique counts on the size of the activations required for gradient computations and the layer sizes to achieve an effective compression ratio.

FedDF [111], as discussed in Section 2.3.3.2, introduces a distillation signal to improve model performance. To extend on FedDF's discussion, as their solution only needs access to the model's logits, they showed it could be used for training heterogeneous clients. As different clients have different constraints, the paper presented a solution to allow each client to define its own model architecture. To prove the efficacy of their method, the distillation dataset was assumed to be unlabeled or artificially generated. They proposed an ensemble knowledge distillation to fuse the knowledge of different models. Once at a time, each model would be considered the student, and all other models would be the teachers. The distillation loss is then composed of the logits of the student and the average logits of all teachers using the KL divergence.

FedNAS [72] introduces NAS in federated learning to find the architecture per client/dataset. FedNAS allows clients to search for the best architecture locally using a gradient-based NAS algorithm, MiLeNAS [73]. Each client optimizes the architecture parameters and model weights on its local data simultaneously. After local training, clients send their updated architecture and model weights to the server, aggregating the parameters and sending them back to the clients. This is possible as all clients share the same restrained search space and similar architectures.

3.3 Recapitulation

We have introduced the use of compression techniques to centralized deep learning, motivating the application of quantization and pruning due to hardware constraints. These techniques were then extended to the federated learning setting to tackle the communication and computational challenges. The next chapters demonstrated how we, based on these techniques, have contributed to reducing the communication cost in federated learning.

Chapter 4

Cutting Communication Costs

Contents

4.1 Contributions

This chapter presents our contributions to reducing communication costs in federated learning. We have addressed this problem using two approaches.

The first approach focuses on reducing message sizes to minimize energy and bandwidth requirements. Our proposed method can be seamlessly integrated with other techniques in federated learning. Specifically, we combine weight magnitude pruning with entropy compression to achieve significant message compression. Compared to prior methods[136, 104], our approach is simpler yet more effective, reducing message sizes by 50% with less than 1% accuracy loss. The code for this work is publicly available [1].

The second approach addresses the communication bottleneck in federated learning by adopting the Low-Rank Adaptation (LoRA)[81] technique. Rather than focusing on model compression techniques such as pruning, we explored the application of LoRA to train small vision models from scratch. Unlike previous works [84, 11, 37] that employ LoRA in federated learning, our work demonstrates the potential of LoRA in this new context. We can summarize this work's contributions as follows:

- We propose Federated Learning Compression with Low-Rank Adaptation (FLoCoRA), which integrates LoRA adapters into the federated learning framework. FLoCoRA can be implemented with any federated learning optimization method, establishing a strong baseline for communication reduction in federated learning. The code is publicly available [2].
- We investigate the impact of LoRA hyperparameters in a classification task using small vision models. We demonstrate that these models can be trained from scratch while reducing message sizes by up to 4.8 times, with only a 1% accuracy loss for a ResNet-18.
- We introduce an affine quantization scheme to FLoCoRA, enabling further compression rates of 18.6 to 37.3 times with up to a 1% accuracy loss for a ResNet-18.

The results of this chapter were published in two instances. One at the IEEE 30^{th} International Conference on Electronics, Circuits, and Systems (ICECS 2023) [60] and the other at published at the 32^{nd} European Signal Processing Conference (EUSIPCO 2024) [61], respectively.

4.2 Magnitude Pruning for Double Side Compression

Building upon the standard federated learning pipeline described in Section 2.3.2, we incorporated pruning to increase message sparsity between server and clients. Inspired by [68], both the server and clients apply non-structured weight-magnitude pruning immediately before transmitting their messages. In this process, the global weights are pruned based on their absolute values, following a predefined pruning rate. As a result, the smallest $\theta\%$ of weights are set to zero, generating sparse messages. This approach ensures that both the server and clients maintain a consistent level of sparsity in their communications during each training round.

1. https://github.com/lgrativol/fl_exps
2. https://github.com/lgrativol/flocora_eusipco24

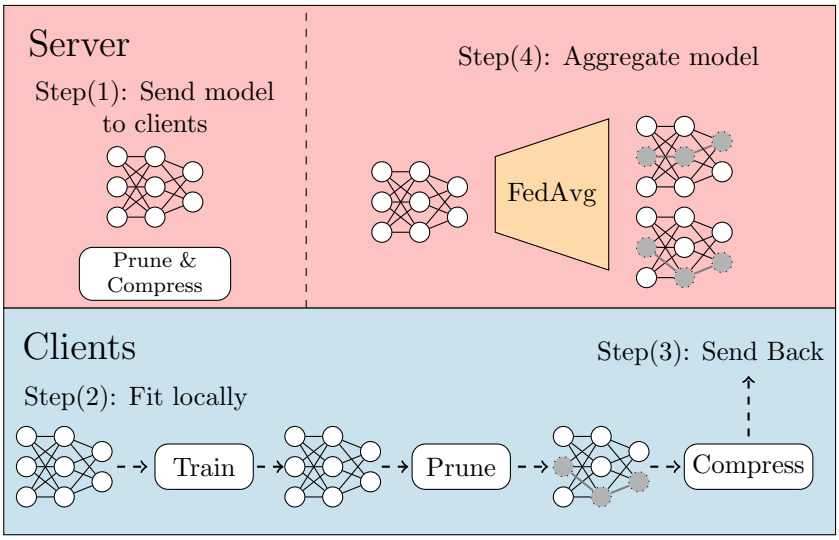


Figure 22 – A reviewed training loop, including the pruning step. Source: [60] © [2023] IEEE.

4.2.1 Adding Pruning to Federated Learning

Initially, we conducted experiments to analyze the behavior of our pruning method, particularly its impact on message compression. We observed the trade-offs between compression and convergence in a federated learning training loop by applying different pruning levels. Figure 22 shows a view of the modified training loop. Based on these findings, we expanded our study to include a comparative evaluation with the ZeroFL[136] method, summed up in section 3.2.3.

As illustrated in Figure 22, our approach is integrated at two moments within the federated learning framework. First, during *Step (1)*, the global model is pruned at the server level before being transmitted to the clients. Following this, clients perform their local training as described in *Step (2)*. Before sending the updated model back to the server, the pruning method is applied again in *Step (3)*. Once the message has been pruned, we compress the message to the ZIP format. We chose ZIP format due to its implementation of entropy encoding (Huffman Coding [122]). This encoding technique works by assigning shorter codes to more frequent elements and longer codes to less frequent ones. As a result, the more common elements require fewer bits to be encoded, leading to a reduction in message size. As we do not use a data-dependent technique to compress the message, both the client and server can apply it to reduce communication costs, avoiding overhead information or one-side compression only.

4.2.2 Pruning applied to Federated Learning

To evaluate the impact of our technique in federated learning, we simulated an image classification task using the Canadian Institute For Advanced Research (CIFAR)-10 datasets with a ResNet-12 model consisting of 780K parameters and a size of 2.97 MB. The simulation was conducted using the Flower framework [17] with 10 clients. In each training round, 40% of the clients were selected, and the process was repeated for 100 rounds. Each client employed SGD with momentum as the optimizer. For simplicity, we used the same hyperparameters as in [140] and replaced the batch normalization layer with a group normalization layer, as suggested by [80]. The server aggregated the clients' updates using the FedAvg strategy.

The training examples were distributed among clients using LDA [80]. We explored two scenarios, one with a concentration equal to infinity, an Independent and Identically Distributed (IID) scenario, and a concentration of 100 for our Non-IID scenario.

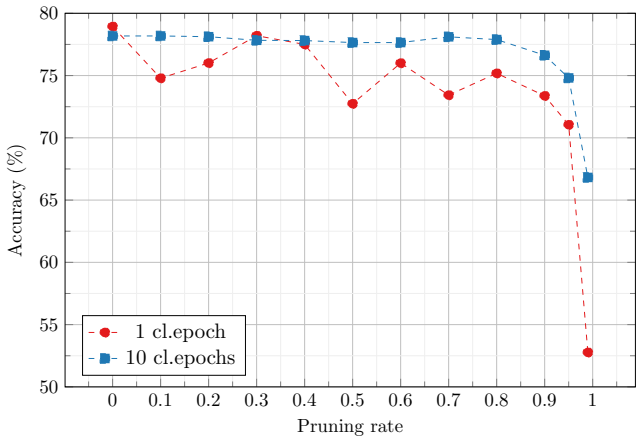


Figure 23 – Pruning effect on the accuracy in function of the pruning rate, where the rate represents the % of total parameters pruned, for 1 and 10 clients epochs. Source: [60] © [2023] IEEE.

As noted in previous works [117, 89] the number of local iterations performed by clients during training can have an important impact on model aggregation. For so, we decided to investigate this behavior in the presence of model compression. The results in Figure 23 show that spending more time on each client contributes to a more robust model, allowing sparser data communications while retaining approximately the same accuracy. However, this approach also results in a higher total number of local iterations.

To further validate the experiment, we repeated it using the CIFAR-100 dataset, with the results shown in Figure 24. As noted by [140], CIFAR-100 presents a more challenging task due to the greater number of classes and fewer examples per class. This is reflected in the results, where the model shows greater sensitivity to pruning, exhibiting more degradation compared to its performance on the CIFAR-10 dataset.

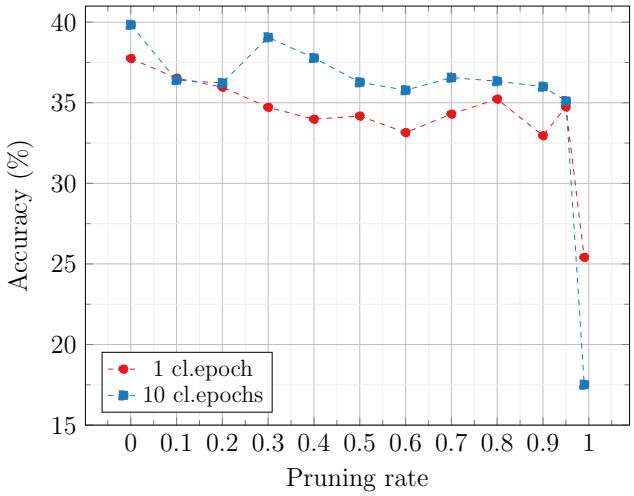


Figure 24 – Pruning effect on the CIFAR-100 dataset.

To further evaluate the feasibility of our method in a non-IID scenario, we replicated the test case used by ZeroFL. In this case, the model is a ResNet-18 with 11 million

trainable parameters, occupying 44.7 MB. The scenario simulates 100 clients with a 10% participation rate for only 1 local epoch and with LDA concentration of 1.0 (Non-IID), training for 700 communication rounds. Table 2 presents the evaluation results, which are the averages of three separate runs, each using different random seeds to create varied data distributions among clients.

Table 2 – Comparation to ZeroFL. Where SP indicates the Sparsity level from the ZeroFL method. Source: [60] © [2023] IEEE.

Method	Compression	Accuracy	Message Size (MB)
ZeroFL	**Full model**	80.62 ± 0.72	**44.7**
	90 % SP + 0.2 Mask Ratio	81.04 ± 0.28	27.3
	90 % SP + 0.0 Mask Ratio	73.87 ± 0.50	10.1
Global magnitude (Ours)	**Full model**	84.43 ± 0.36	**44.7**
	10 % pruning rate	85.96 ± 0.37	38.1
	20 % pruning rate	85.57 ± 0.19	34.8
	30 % pruning rate	85.03 ± 0.32	31.1
	40 % pruning rate	85.20 ± 0.20	27.1
	50 % pruning rate	83.85 ± 0.65	23.0
	60 % pruning rate	83.19 ± 0.44	18.9
	70 % pruning rate	82.25 ± 0.63	14.5
	80 % pruning rate	80.70 ± 0.24	9.8
	90 % pruning rate	76.77 ± 0.47	4.9
	95 % pruning rate	69.14 ± 0.85	2.5
	99 % pruning rate	0.10 ± 0.0	0.5

Table 2 compares our approach and ZeroFL. Initially, without pruning, our baseline achieves higher accuracy than ZeroFL, which can be attributed to two key differences. First, we do not employ SWAT [138] for local training. Second, we use a batch size of 8, whereas ZeroFL does not specify the batch size used. As highlighted in prior works [117], batch size is a critical hyperparameter that significantly impacts the accuracy of the model aggregation. While SWAT is effective in reducing communication costs, it also affects model accuracy, which may explain the performance gap. Our baseline, which uses pure FedAvg without any compression, achieves 4% higher accuracy compared to ZeroFL.

Moreover, when pruning is applied, our approach demonstrates less accuracy degradation than ZeroFL for equivalent levels of pruning. For example, while ZeroFL experiences an 8% accuracy loss when pruning the model to 10 MB, our method only incurs a 4.63% reduction. These results suggest that allowing clients the flexibility to perform pruning independently better compensates for the sparsity introduced. This flexibility results in more efficient message compression, leading to a more robust global model.

4.2.3 Compressing more with Quantization

Building on the message savings observed from our pruning experiments, one might ask if it is possible to achieve even smaller message sizes. As discussed in Section 3.1, another widely used compression technique is quantization. Figure 25 illustrates the impact QAT [132, 110] in the previously described IID scenario.

For this study, we experimented with 1-bit, 4-bit, and 8-bit quantization levels. Binary networks were handled using Binary Connect [40], while 4-bit and 8-bit quantizations were implemented using the Brevitas framework [132] with its default quantization scheme. In this scheme, weights are quantized to 4-bit and 8-bit integers, with the QAT scaling calculated on a per-layer basis.

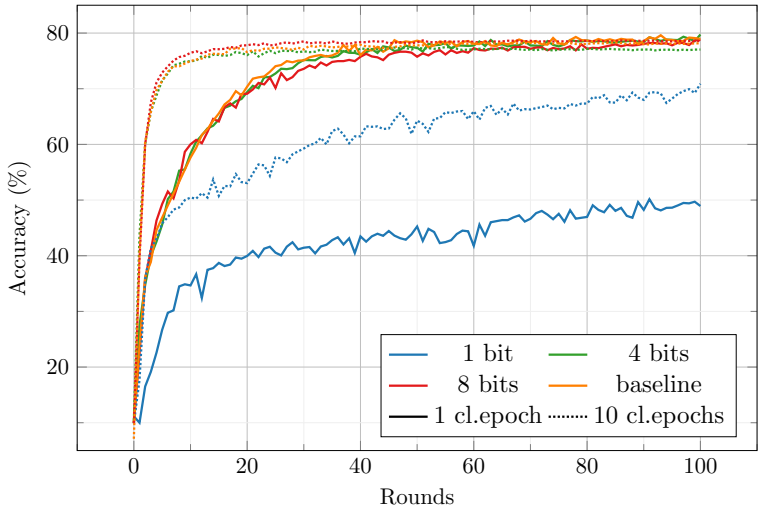


Figure 25 – Accuracy evolution comparison between baseline (FP32), 1-bit, 4-bit and 8-bit, for 1 and 10 clients epochs. Source: [60] © [2023] IEEE.

Figure 25 shows that the convergence time, or the number of rounds required to reach maximum accuracy, varies between experiments, as it depends on the level of quantization. Both 4-bit and 8-bit formats allow us to achieve an accuracy comparable to the reference model, but they also highlight a trade-off between communication and computation. For example, to reach a similar accuracy of around 75% 40 rounds of communication and a total of 40 epochs are required when using 1 local epoch per round. In contrast, when using 10 local epochs per round, achieving the same accuracy demands 100 total epochs across 10 communication rounds.

In the case of 1-bit quantization, increasing the number of local epochs per round from 1 to 10 significantly improves accuracy, from 48.8% to 70.9%. Despite this increase in total epochs from 100 to 1000, the communication cost remains the same. As seen in the IID pruning experiment, Figure 23, spending more time on each client helps build a more robust model, compensating for the perturbations introduced by quantization.

Table 3 summarizes the message sizes exchanged between clients and the server for the IID experiences. For quantization, the message size is determined solely by the quantized weights, as the server already knows the client's quantization scheme. The table also demonstrates that even simple compression methods can reduce network bandwidth usage by 2 to 4 times, with minimal impact on accuracy.

In this section, we demonstrated the promising application of traditional compression techniques in the context of federated learning. These straightforward yet effective methods reduced message sizes by up to 50% without significantly affecting model accuracy, leading to direct savings in energy and bandwidth. Additionally, our approach allows each client to tailor the pruning process to their specific dataset, providing greater flexibility. By incorporating quantization into the training process, we introduced an additional layer of compression to the framework. Although our results show the effectiveness of pruning and quantization individually, combining both techniques could further enhance message compression. In early tests of integrating the

Table 3 – Summary of message size and accuracy for the CIFAR-10 dataset for the IID case. Source: [60] © [2023] IEEE.

Compression Technique	Accuracy (%)		Message Size (MB)
	1 Local Epoch	10 Local Epochs	
Baseline	78.94	78.18	2.97
Pruning			
10 %	74.79	78.18	2.57
20 %	76.01	78.12	2.34
30 %	78.20	77.83	2.10
40 %	77.50	77.81	1.85
50 %	72.74	77.65	1.57
60 %	76.00	77.65	1.29
70 %	73.43	78.11	1.01
80 %	75.18	77.89	0.70
90 %	73.37	76.63	0.37
95 %	71.05	74.81	0.19
99 %	52.77	66.82	0.04
Quantization			
8 bits	78.80	78.58	0.75
4 bits	79.74	77.04	0.38
1 bit	48.93	70.89	0.10

described pruning approach plus the QAT framework, clients could not learn, indicating that a more thoughtful design is needed to combine both techniques. Based on these findings, we believe that integrating compression-aware training methods while ensuring smooth compatibility is a critical step toward advancing FL.

4.3 FLoCoRA

Low-Rank Adaptation (LoRA) methods have gained popularity in efficient parameter fine-tuning of models containing hundreds of billions of parameters. In this work, instead, we re-adapted it to reduce communication costs in federated learning. In this section, we introduce the concept of model fine-tuning and the LoRA method. Then, we demonstrate how this can be integrated into the federated learning framework.

4.3.1 Fine-Tuning Models

Deep learning models are typically trained on well-known datasets that serve as standard benchmarks, allowing for consistent comparison between different machine learning models. However, retraining a state-of-the-art model from scratch on large-scale datasets can be computationally intensive; it often yields diminishing returns, particularly when pre-trained weights are publicly available and becoming more common. A notable example is the HuggingFace [3] that can serve as a hosting site for the model's weights. These pre-trained models offer generalization capabilities that can be leveraged for new tasks or datasets, significantly reducing the cost of training by adapting the existing weights [176].

3. https://huggingface.co/

This process, known as fine-tuning, traditionally involved retraining some or all layers of a pre-trained model to specialize it for a new task in the pre-LLM era [182]. However, the rise of LLMs, with their novel architectures and massive scale, often comprising billions or even trillions of parameters, has required new strategies to adapt such models efficiently. This has given rise to Parameter-Efficient Fine-Tuning (PEFT) [74], which focuses on optimizing fine-tuning methods to minimize computational resources such as memory and time. PEFT techniques are designed to selectively update only a small subset of parameters or introduce lightweight parameter modules, enabling the efficient fine-tuning of models at extreme scales, from billions to trillions of parameters[181]. This shift represents a significant evolution in the fine-tuning paradigm, especially as models continue to grow in size and complexity.

4.3.1.1 Low-Rank Adaptation

Among PEFT methods, LoRA has recently gained attention for task adaptation in models with hundreds of billions of parameters [81, 16, 84]. LoRA addresses the challenges of fine-tuning large-scale pre-trained models for specific tasks while significantly reducing computation time and memory overhead. Instead of fine-tuning all model parameters, LoRA introduces a lightweight parallel adaptation layer that focuses on updating a small subset of parameters.

In LoRA, the parameters of a pre-trained linear layer, $W_l \in \mathbb{R}^{I \times O}$, are modified by adding a low-rank adaptation. Specifically, a parallel layer is introduced, resulting in the new set of parameters $W_l^* = W_l + \frac{\alpha}{r} BA$, where $A \in \mathbb{R}^{I \times r}$ and $B \in \mathbb{R}^{r \times O}$ are two matrices with a maximum rank of r. The term $\frac{\alpha}{r}$ serves as a scaling factor, with α being a hyperparameter and r representing the rank. The core idea is to freeze the original weights, W_l, and train only the low-rank matrices, A and B, which provide a more compact and efficient update to W_l.

One of the key advantages of this approach is that the adaptation matrix, BA, can be merged back into the original pre-trained weights, resulting in the adapted weights W_l^* without incurring additional latency during inference. Figure 26 shows how the input is processed by both the pre-trained weights and the LoRA-adapted weights, with the outputs from both paths merged to produce a single final output.

[81] has shown that for $r << min(I, O)$, W_l can be adapted to a new task, greatly reducing the memory associated with updating all the weights in a model, once LoRA spawns fewer parameters. As an example, the original LoRA paper illustrates that a GPT-3 [2] model with 175 billion parameters was fine-tuned for various text-related tasks using between 4.7 million and 37.7 million parameters (0.003% - 0.021% of the total), with an accuracy drop of less than 1%.

4.3.2 LoRA in the Context of Federated Learning

In Federated Learning, when LoRA is applied, the original model remains frozen, and only the adapters, B and A, are trained. These adapters have significantly fewer parameters than the original model, resulting in lighter message exchanges between the server and clients. Since the original model stays frozen, there is no need to transmit it between the server and clients in every communication round, which reduces the overall communication cost.

Previous studies, such as SLoRA [11], have explored the potential impact of LoRA techniques in federated learning. They proposed combining standard federated learning

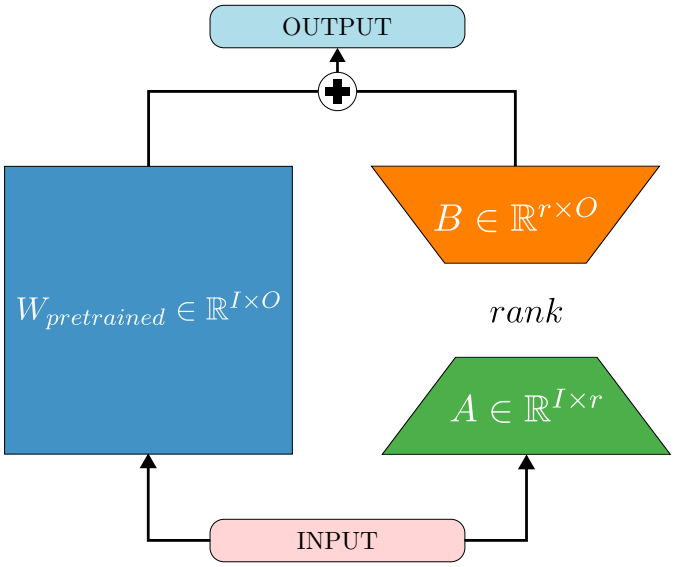


Figure 26 – Low-Rank Adaptation of a pre-trained layer $W_{pretrained}$ with adapters B and A, controlled by the rank hyperparameter "r".

training with matrix decomposition to achieve optimal initialization for both the original model and the matrices B and A. Similarly, HLoRA [37] allowed clients to select different LoRA ranks based on their individual resource constraints, enabling them to adapt a pre-trained base model accordingly. On the other hand, FedPara [84] introduced a low-rank adapter, formulated as $W_l^* = W_l + (A_1 \cdot B_1^T) \odot (A_2 \cdot B_2^T)$, where $\odot$ denotes the Hadamard product. This method achieves an update of higher rank compared to traditional low-rank decompositions.

However, both SLoRA and HLoRA have focused on applying LoRA techniques to larger models, such as LLMs and foundation models, without investigating their use in training small CNNs from scratch. FedPara [84], which is more closely aligned with our work, has tested its low-rank approach on small CNNs. However, its method, including quantization to FP16, and the adaptation of FedPAQ [141], was evaluated on relatively simple test scenarios.

4.3.3 FLoCoRA Framework

In this work, we propose keeping the parameters of the randomly initialized model, shared between clients at the start of training, completely unchanged throughout the training. Instead, only the adapter parameters, B and A, will be trained, exchanged, and updated. By keeping the original model parameters frozen, the exchange of LoRA parameters should be sufficient to capture the updates from each client. The server continues to receive the updated LoRA parameters from the clients, allowing this method to be seamlessly integrated with other federated learning techniques without requiring additional modifications.

Figure 27 shows our proposed method, **F**ederated **L**earning **Co**mpression with **Lo**w-**R**ank **A**daptation (FLoCoRA), within a single communication round. All clients begin with the same initial weights, denoted as $\mathcal{W}_{initial}$, which remain unchanged throughout the training process. In the first step (1), the server sends the global LoRA adapter parameters, $\bar{\Delta}_m^t$, to a selected subset of clients, denoted as M. In step (2), each client locally trains its LoRA adapter and uploads the resulting parameters, Δ_m^{t+1}, back to the server in step (3). Finally, in step (4), the server applies a weighted averaging mecha-

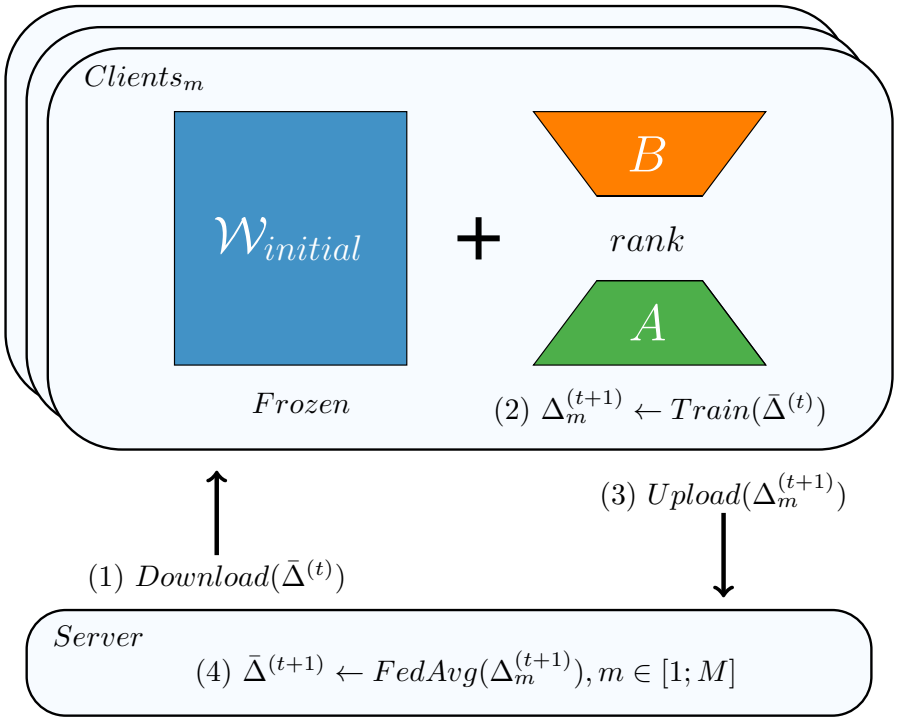


Figure 27 – FLoCoRA training loop, where Δ represents the matrice AB. Adapted from: [61] © [2024] IEEE.

Table 4 – Number of parameters for different sizes of r. For each value, we have the total number of parameters to be trained/sent and the total number of the parameters with the original model plus the LoRA adapter. Source: [61] © [2024] IEEE.

Method	Total Params	Trained Params	% of Trained Params
FedAvg	**1.23M**	**1.23M**	**100**
FLoCoRA ($r = 8$)	1.30M	69.45K	5.35
FLoCoRA ($r = 16$)	1.36M	131.92K	9.70
FLoCoRA ($r = 32$)	1.48M	256.84K	17.30
FLoCoRA ($r = 64$)	1.73M	506.70K	29.22
FLoCoRA ($r = 128$)	2.23M	1.00 M	45.05

nism, similar to FedAvg, to compute the updated global LoRA adapter parameters for the next round, denoted $\bar{\Delta}^{t+1}$.

Table 4 presents the size of the trainable parameters for a ResNet-8 model with different values of r. It's important to note that while LoRA is applied to adapt the convolutional layers, the normalization and FC layers are trained conventionally. These layers are also exchanged between the server and clients.

For convolution layers, as depicted in Figure 28, we follow the decomposition proposed in [83]. Let $W_l \in \mathbb{R}^{C_{out} \times C_{in} \times k \times k}$ be a convolution layer; then we define its LoRA adapter matrices as $A \in \mathbb{R}^{r \times C_{in} \times k \times k}$ and $B \in \mathbb{R}^{C_{out} \times r \times 1 \times 1}$. Output channels are denoted as C_{out}, the input channels as C_{in}, and the kernel size as k. An important difference from the original LoRA, the hyperparameter r does not represent a reduction in terms of matrix rank, but rather a reduction in the number of channels, which represents a bottleneck effect in the decomposition.

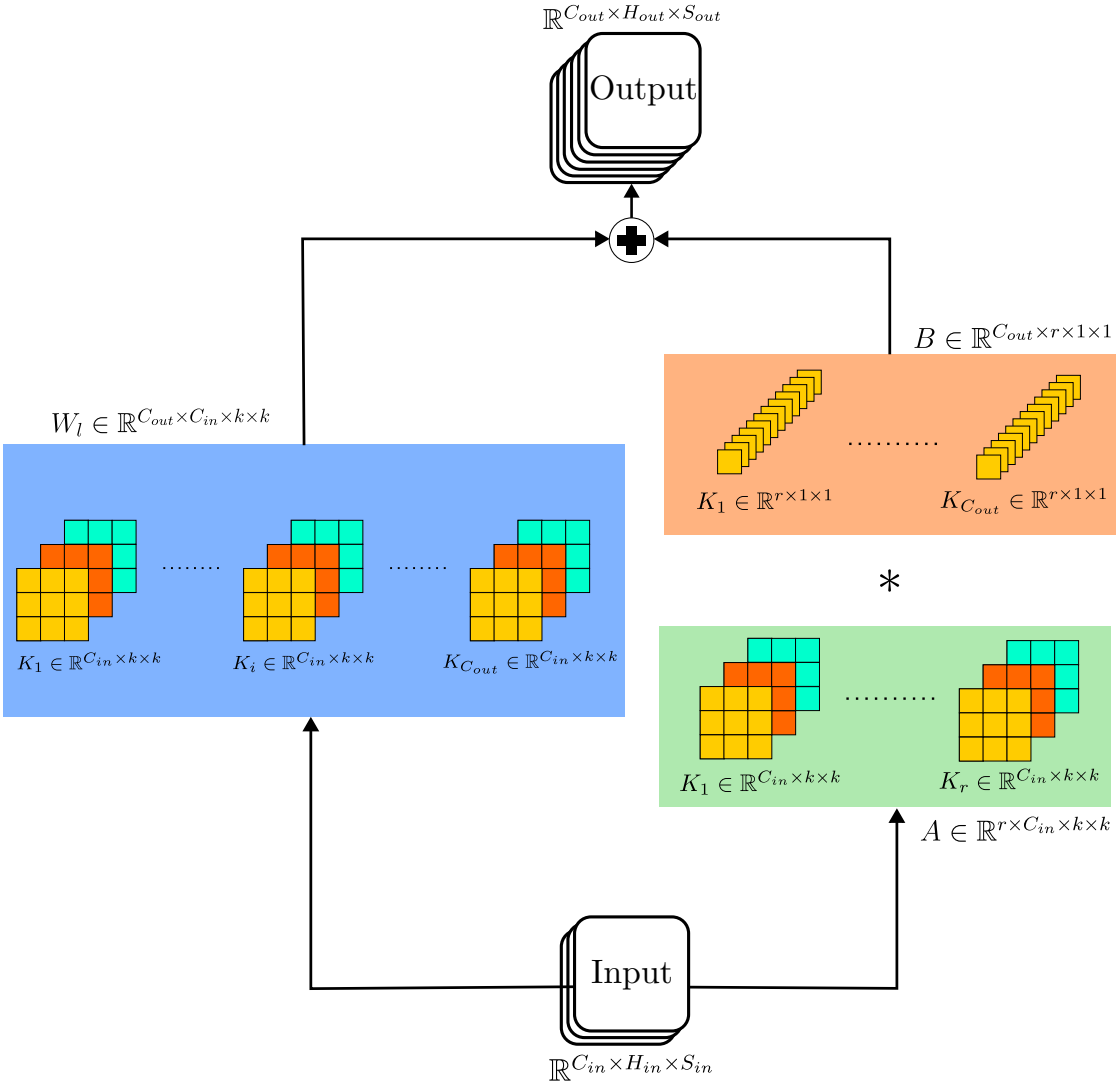


Figure 28 – A convolution layer adapted with LoRA.

To check the impact of using LoRA in federated learning, we identify which layers of the original model need standard training and which can be adapted with LoRA. Next, we compare our approach with recent studies on compression methods in federated learning, demonstrating that LoRA provides a strong baseline for future work in the field. At last, we show that quantizing LoRA parameters can further enhance message compression, even when dealing with highly diverse data distributions across clients.

4.3.4 FLoCoRA Results

Our federated learning setup involves 100 clients, with 10% of clients randomly selected in each round. The training runs for a total of 100 rounds. Following the approach in [84], we keep the batch size, learning rate, number of local epochs, and momentum constant across all clients, set to 32, 0.01, 5, and 0.9, respectively. Clients use SGD with momentum [145], and FedAvg serves as the aggregation algorithm. Each experiment was run three times using different random seeds.

To evaluate how FLoCoRA training affects convergence in federated learning, we initially trained a ResNet-8 model on the CIFAR-10 dataset. We used a concentration of 0.5 for the LDA distribution. As pointed out by [80], we replaced the batch normalization

layer with a group normalization layer. Table 5 provides an ablation study using LoRA hyperparameters $r = 32$ and $\alpha = 512$. The value $r = 32$ was chosen as the minimum rank at which accuracy degradation remained below 1%.

We began with a randomly initialized ResNet-8 model and froze the entire model. LoRA adapters were then introduced to all convolutional layers and the final FC layer. This configuration is labeled "FLoCoRA Vanilla" in Table 5. Next, we progressively adjusted the model setup. First, we unfroze the normalization layers to allow them to learn the running statistics, labeled "+ Norm. Layers". Then, we removed the LoRA adapter from the final FC layer and unfroze it, as represented by "+ Final FC".

Table 5 – The effect of training different layers with or without LoRA adapters on selected layers. Source: [61] © [2024] IEEE.

Method	Nb. of Params. to update	Accuracy
FedAvg	**1.23 M**	**76.14 ± 0.74**
FLoCoRA Vanilla	0.26 M	22.14 ± 3.99
+ Norm. layers	0.26 M	39.80 ± 12.05
+ Final FC	0.26 M	**75.51 ± 1.34**

The normalization layers must be trained because they cannot be adapted using LoRA methods, as they must capture running statistics. We also hypothesize that the final FC layer requires full training because the lower rank may not suffice for this highly specialized and sensitive layer [128]. Therefore, this FLoCoRA configuration is used in all experiments.

To highlight the importance of selecting the right layers, we explore the trade-off between the rank r and the scaling parameter α. We compare the FedAvg baseline with two setups where α is set to $2r$ and $16r$. The results, shown in Figure 29, indicate that while LoRA's original paper[81] scales α to twice the rank for LLMs, we found that further increasing this factor when training small CNNs with federated learning can improve accuracy by up to 4.4%. Increasing the scaling factor of the LoRA-adapted layers effectively raises their learning rate while keeping more sensitive layers, such as normalization and FC layers, at a lower learning rate, thus enhancing training stability and performance.

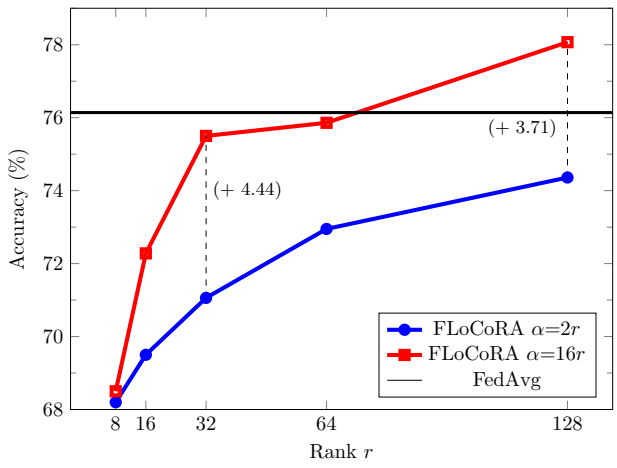


Figure 29 – The relationship between the r hyperparameter in FLoCoRA and the scaling factor α, in $\frac{\alpha}{r}$. Two scenarios are evaluated, $\alpha = 2r$ and $\alpha = 16r$, against FedAvg.

As shown in Figure 29, FLoCoRA achieves an accuracy drop of less than 1%, while sharing only 0.26M parameters, which is a 4.8× reduction compared to sharing the entire model. For a rank of 128, the number of shared parameters is closer to that of the original model (see Table4), but accuracy improves by 2%. In the ResNet-8 architecture, the convolution layers have output channel sizes of 64, 128, or 256. By selecting a rank of 128, we effectively increase the rank for certain layers while maintaining it for others. This results in larger rank updates for the shallower convolution layers, while the deeper layers receive a low-rank update. Moreover, because the larger convolution layers with 256 output channels are adapted with a rank of 128, the total number of parameters is slightly reduced.

Table 6 – Additional results for the CIFAR-100 and CINIC-10 datasets with FLoCoRA expressed in Total Communication Cost (TCC)

CIFAR-100

Method	Rank	Nb. Parameters	TCC	Accuracy
FedAvg	-	1.25 M	1.00 GB	46.24 ± 0.15
FloCoRA	r=16	1.40 M	124.04 MB ($\div 8.06$)	40.67 ± 0.14
	r=32	1.53 M	224.00 MB ($\div 4.46$)	45.93 ± 0.25

CINIC-10

Method	Rank	Nb. Parameters	TCC	Accuracy
FedAvg	-	1.23 M	982.07 MB	67.07 ± 0.54
FloCoRA	r=16	1.36 M	105.53 MB ($\div 9.31$)	62.51 ± 0.43
	r=32	1.48 M	205.5 MB ($\div 4.78$)	65.91 ± 1.05

Additionally, we experimented with FLoCoRA on the CIFAR-100 and CINIC-10 datasets. Table 6 shows that we achieved communication reductions of 4.46× and 4.78×, with an accuracy drop of 0.31% and 1.16% for CIFAR-100 and CINIC-10, respectively. The table displays the total number of parameters, including those not exchanged between the client and server, but represents the TCC (Equation 16) based on trainable parameters. We used here the same hyperparameters as the CIFAR-10 experiment.

Thus far, FLoCoRA has successfully reduced the number of trainable parameters, thereby decreasing the amount of data that needs to be communicated in each round. Next, we explore the impact of applying an affine quantization to both the client and server messages. For the convolution layers, we compute the scale factor and zero point values on a per-channel basis, while for the FC layer, these values are calculated per column. The normalization layers remain unquantized. We use 2-, 4-, and 8-bit formats to quantize the trainable layers. For a model like ResNet-8, this results in reductions in the

Table 7 – TCC for different quantization levels with FLoCoRA, for a rank of 32 and alpha of 512, during 100 rounds of FL. Source: [61] © [2024] IEEE.

Method	Quantization	TCC	Accuracy
FedAvg	FP32	982.07 MB	76.14 ± 0.74
FLoCoRA	FP32	205.47 MB ($\div 4.8$)	75.51 ± 1.34
	int8	55.56 MB ($\div 17.7$)	74.21 ± 1.05
	int4	30.15 MB ($\div 32.6$)	73.15 ± 0.18
	int2	17.44 MB ($\div 56.3$)	55.03 ± 1.90

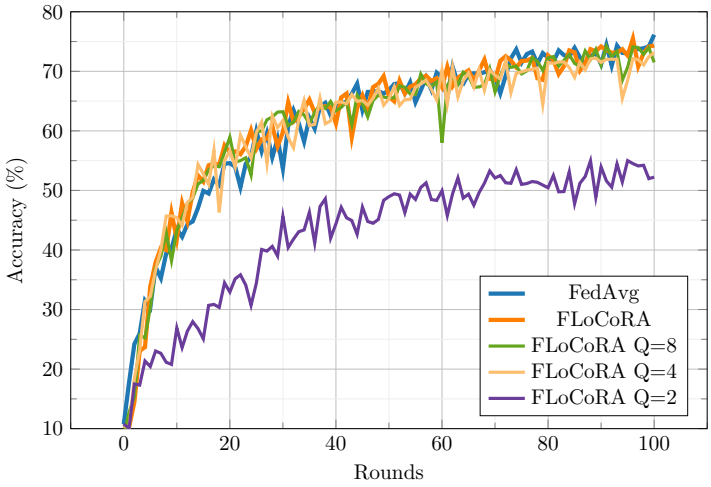


Figure 30 – Convergence behavior between FedAvg, FLoCoRA with rank of 32 and its quantized versions of 2/4/8-bits. Source: [61] © [2024] IEEE.

total communication cost (Equation 16) by 56.3, 32.6, and 17.7 times for the 2-, 4-, and 8-bit formats, respectively. We also account for the overhead required to transmit the quantization scale factors and zero points with 32 bits. Table 7 summarizes the TCC, as expressed in Equation 16, when training a model with FLoCoRA and quantization.

Figure 30 illustrates the accuracy progression for federated learning FedAvg, FLoCoRA, and the quantized version of FLoCoRA. Notably, the convergence rate for both FP32 FLoCoRA and its int8 quantized counterpart remains unaffected, indicating that this approach effectively reduces communication costs without harming performance. However, as shown in Table 7, the quantized versions do experience a slight accuracy degradation, with a 2% drop in the int8 case. Despite similar convergence rates, the quantized versions introduce more instability in training, suggesting a promising direction for future research: improving the quantization scheme for FLoCoRA. Revisiting model compression techniques [127, 52] and combining them with FLoCoRA could yield a more refined quantization method. Nevertheless, we demonstrate that even a simple round-to-nearest quantization technique achieves competitive results.

The FedPara[84] method was applied to a VGG-16 model and compared to FedAvg and a low-rank tensor parameterization using Tucker decomposition, achieving communication reductions between 2.8 and 10.1 times. FLoCoRA, in its base form, delivers comparable compression ratios, proving that the low-rank adapter for convolution proposed in [83] can efficiently train a small CNN from scratch.

Finally, we compared FLoCoRA, combined with the proposed quantization approach, against techniques from ZeroFL [136] and Magnitude Pruning [60]. Table 8 demonstrates that despite being primarily a fine-tuning method, LoRA serves as a strong baseline for communication savings in federated learning. For this experiment, we replicated the setup used by [136, 60], which involved 100 clients training a ResNet-18 model for 1 local epoch, with an LDA parameter of 1.0, over 700 communication rounds.

Table 8 extends the results from Table 2, exploring the impact of different rank values and quantization levels when using FLoCoRA. A rank of 64 achieves a compression ratio similar to 40% pruning without any accuracy loss. As the rank decreases, message sizes are reduced, but accuracy remains higher than with 80% pruning. Notably, a rank of 16 offers a compression rate four times greater than either pruning or ZeroFL, with

Table 8 – Comparing LoRA and quantization to ZeroFL and Magnitude Pruning methods. SP indicates the sparsity level used during the SWAT step, and MR the mask ratio indicating a percentage of "extra" weights clients can send besides the Top-K weights, as per the ZeroFL method. Source: [61] © [2024] IEEE.

Method	Config.	Message Size (MB)	TCC (GB)	Accuracy
FedAvg	**Full Model**	44.7(÷1.0)	62.6	84.43 ± 0.36
ZeroFL [136]	90% SP+ 0.2 MR	27.3(÷1.6)	38.2	81.04 ± 0.28
	90% SP+ 0.0 MR	10.1(÷4.4)	14.1	73.87 ± 0.50
Magnitude Pruning [60]	40% prune	27.1 (÷1.6)	38.0	85.20 ± 0.20
	80% prune	9.8 (÷4.6)	13.7	80.70 ± 0.24
FLoCoRA	r=64	9.2 (÷4.9)	12.9	85.17 ± 0.44
	r=32	4.6 (÷9.7)	6.5	83.90 ± 0.20
	r=16	2.4 (÷18.6)	3.3	82.33 ± 0.35
	r=64, Q=8	2.4 (÷18.6)	3.3	85.24 ± 0.23
	r=32, Q=8	1.2 (÷37.3)	1.7	83.95 ± 0.32
	r=16, Q=8	0.7 (÷63.9)	1.0	81.89 ± 1.01

less accuracy degradation. There is a clear trade-off between reducing the rank and applying quantization. For example, applying 8-bit quantization to a rank of 64 maintains accuracy, while achieving the same compression rate as a rank of 16 without quantization. This trend continues with lower ranks: combining FLoCoRA with quantization reduces communication costs by a factor of 63.9, resulting in only a 4% accuracy loss. In this case, each client sends only 0.7 MB to train a 44.7 MB model.

Interestingly, when compared to our earlier results in Table 7, the quantized versions of ResNet-18 show less accuracy degradation. We attribute this to the fact that ResNet-18 is 9× as large and is trained 7× as long in a relatively easier training scenario. The higher training time, for a scenario that samples 10 clients each round, allows for the training to access all client's datasets multiple times. The earlier experiments used an LDA of 0.5, while ResNet-18 uses an LDA of 1.0, which results in more similar data distributions across clients as the LDA parameter increases. We hypothesize that LoRA-based methods could scale more effectively with larger models and more challenging training setups, suggesting an interesting direction for future research. One point of particular importance is the relationship between the number of output channels and the rank used to adapt each convolution layer. A better understanding of how the per-layer rank influences the layer capacity to learn could lead to an optimization of the rank used based on a per-layer sensitivity.

4.4 Compression or Just Smaller Models ?

So far, we have compressed models using techniques such as pruning, quantization, and adapting LoRA for federated learning. Upon re-examining the impact of these methods, beyond training and accuracy considerations, the key benefit in terms of communication efficiency is the reduction in the size of messages exchanged between clients and the server, measured in bytes. This is accomplished by introducing sparsity, compressing the messages, simplifying the data format, or reducing the number of parameters through low-rank approximations.

Next, we explored what happens when using a model specifically designed to meet the communication constraints. The idea is to compare a network that produces the same message size as the final compression results achieved through pruning, quantization, and/or LoRA, with what we call a "smaller model."

To explore the impact of using a compressed model versus a smaller model, we use a scenario with 100 clients, where 10 clients are sampled per round for a total of 100 rounds. Each client trains its model for 5 local epochs and a batch size of 32. This is the same scenario used in the initial experiments of FLoCoRA and the ResNet-8. Finally, the CIFAR-10 was split using LDA with a concentration of 0.5. This represents a more challenging scenario than the one used in Table 8. We trained three models — ResNet-20, ResNet-18, and ResNet-8 — using the FedAvg algorithm. The goal is to compare similar models in terms of parameters, specifically ResNet-20 and ResNet-8, and to contrast them with ResNet-18.

ResNet-18 is an adaptation of a model typically used for ImageNet, but here it was re-adapted for CIFAR-10. This adaptation involved modifying the final classification layer to output 10 classes, instead of the 1000 classes required for ImageNet, while keeping the rest of the model unchanged [4]. We took a model designed for a more complex task, ImageNet, and applied it to a simpler dataset, CIFAR-10, for our experiment. While the ResNet-8 and ResNet-20 are models more adapted to a dataset with the CIFAR-10 complexity.

Table 9 – Comparing Compressed Models to Just Smaller Models for the CIFAR-10 Dataset. Three different architectures are tested, with different sizes, and FLoCoRA is applied to produce equivalent message size models. The feature maps (FM) hyperparameter corresponds to the width of the first convolution layer in a ResNet architecture; subsequent layers are scaled accordingly. All results in FP32.

Model	Strategy	FM	LoRA Rank	Nb. Params (Trainable)	Accuracy
ResNet-18	FedAvg	64	-	11.17 M	83.34 ± 0.38
		32	-	2.80 M	82.22 ± 0.52
		16	-	0.70 M	77.67 ± 0.86
ResNet-20		64	-	4.33 M	84.07 ± 0.41
		32	-	1.08 M	81.58 ± 0.46
		16	-	0.27 M	77.82 ± 0.59
ResNet-8		64	-	1.23 M	75.99 ± 0.99
		32	-	0.31 M	74.59 ± 0.59
ResNet-18	FLoCoRA	64	16	0.59 M	80.92 ± 0.23
ResNet-20			22	0.58 M	81.46 ± 0.9

Table 9 results allow us to look at the use of "just smaller" models in two ways. First, one clear difference is that a more optimized network is more efficient in learning the task and that the size can be misleading. In this case, ResNet-20 presented itself as a more efficient model, reaching 84.07% of accuracy with 4.33 M of parameters, against ResNet-18 with 83.34% and 11.17 M. The point here is to highlight that compression of highly cumbersome networks can result in sub-optimal performances, where for the same amount of parameters, one could train a ResNet-20 instead of a compressed version of a ResNet-18 and obtain better results. This possibly indicates that verifying that just a smaller model performs better than a compressed method should be a "sanity-check" test. The same observation can be made to the FLoCoRA case, where the models have a degradation of 2.42% and 2.61% for the ResNet-18 and ResNet-20, respectively.

4. This is the same ResNet-18 used for all the experiments in this chapter

A second point of view is the potential of compression techniques to allow clients to compensate for different constraints. So far in our analyses, we have considered that all clients are compressing their models at the same rate. In practice, while training with federated learning, clients can have different immediate constraints, such as bandwidth allocation of energy use. In such cases, compression methods, such as the ones presented in the chapter, can be used to allow each client to have different compression rates. The proposed pruning framework could be modified to allow each client to have its own pruning rate without affecting the global model aggregation. Equally, FLoCoRA could be adapted to allow a different rank per client, with an aggregation method based on matrix decomposition; a similar proposition is found in [37]. Another possibility is to allow clients to reduce costs when necessary resources are needed to be allocated to other processes, reducing the immediate cost. In either case, the federated learning would be able to integrate possibly straggling clients and, at the same time, leverage clients that can train with full models. As future perspectives, it would be interesting to test different model architectures and scenarios to verify the correct trade-offs between the different hyperparameters and possible configurations of federated learning.

4.5 Pre-trained Models for Federated Learning

Works such as [32] and [129] have demonstrated that federated learning can greatly benefit from leveraging public datasets or pre-trained models. These studies highlight that starting federated learning with pre-trained models accelerates convergence and significantly enhances model accuracy. This approach directly mitigates non-IID data effects. Moreover, using pre-trained models in federated learning can be seen as a form of fine-tuning, particularly in the context of personalized federated learning, where each client benefits from a more tailored model. By leveraging pre-trained models as fixed backbones, as suggested in recent works, computation and communication costs can be substantially reduced since clients no longer need to train large-scale models from scratch.

Minimizing the learning objective of federated learning in Equation 7 thus not only involves finding a model that performs well across all clients but also provides clients with a stronger starting point. Using pre-trained models has the potential to reduce the gap in federated learning performance, particularly in resource-constrained environments.

Table 10 – Comparing pre-training models to starting from scratch for FedAvg and FLoCoRA training. (P) indicates that a pre-trained model was used. For the experiments with FLoCoRA, we used a rank of 32 for the ResNet-18 model.

Model	Strategy	Nb. Rounds @ 75%	Accuracy (%)
ResNet-18	FedAvg	44 ± 1.41	83.34 ± 0.38
ResNet-18 (P)	FedAvg	24 ± 5.10	80.59 ± 0.41
ResNet-18	FLoCoRA	49.67 ± 2.50	82.05 ± 0.50
ResNet-18 (P)	FLoCoRA	34.67 ± 5.31	80.46 ± 0.46

In Table 10, we experimented with a ResNet-18 pre-trained on ImageNet and publicly

available at the Pytorch model zoo [5]. The experiment configuration is the same as used in Section 9. To adapt the ResNet-18 model to the CIFAR-10 task, the final classification layer was replaced with a new one with 10 classes instead of 1000. Unlike the results in [32], we do not find an increasing performance using a pre-trained model to adapt the CIFAR-10 task. The performance difference can be connected to an optimization step necessary for the fine-tuning of the pre-trained models. The experimental setting and hyperparameters were the same between starting from scratch and the pre-trained ones. However, we find, even without trying to find the correct fine-tuning hyperparameters, that the pre-trained models had a faster convergence rate. In both cases, with and without FLoCoRA, the pre-trained settings reduced the number of rounds to reach 75% by 45% and 30%, respectively.

Pre-trained models offer a solution for reducing communication costs by minimizing the number of rounds needed to achieve a specific accuracy or metric. The growing availability of publicly accessible models with various architectures presents an exciting research direction in federated learning. As mentioned in Section 4.3.2, prior works have demonstrated the potential of PEFT techniques, such as LoRA, in adapting LLM models. In Chapter 5, we demonstrated the use of a pre-trained model, leveraging few-shot learning, for an embedded system.

An unexplored area is the role of pre-trained models in federated learning, specifically the difference between the tasks they were originally trained on and the federated learning tasks. It's plausible that devices may have highly specific tasks based on their applications, often in fields where public datasets are scarce. For instance, medical applications for rare diseases or rescue drones operating in remote areas. In such cases, improving the selection of pre-trained models based on the federated task could enhance their performance within the framework. One possible approach is to adapt methods like LogMe [177], which attempts to quantify the best pre-trained model for a new task to the federated learning context.

4.6 Recapitulation

This chapter summarizes our contributions to reducing the communication costs in federated learning. Our approach has focused on developing stateless, straightforward techniques that can easily integrate with other strategies while addressing other challenges in federated learning. This design philosophy is seen in both our pruning and LoRA techniques.

The final two sections of this chapter highlight the influence of how the framework is operated and demonstrate how improved formulations and practices can help bridge the performance gap in federated learning. The last section, in particular, has led us to question how an embedded system could leverage a pre-trained model in scenarios with limited or no communication capability, a topic further discussed in Chapter 5.

5. https://pytorch.org/vision/stable/models.html

Chapter 5

Embedded Few-Shot Learning

Contents

5.1 Contributions

In this chapter, we detail how we tackled the challenges of implementing few-shot learning on embedded systems. We consider it as an alternative for co-designing an embedded image classification system for cases where embedded systems could be isolated without communication systems. In such cases, the use of a pre-trained model to compensate for the lack of data is a common approach, frequently used with fine-tuning techniques. The designed implementation pipeline seeks to allow one to explore the trade-offs in model adaptation and hardware performance. As the main focus is on the use of pre-trained models, one can also use it to deploy a model trained with federated learning, where the final global model represents a generalist model on the group of clients. We can summarize this work's contributions as follows:

- One of the first few-shot learning platforms for real-time object classification on an FPGA SoC in the literature,
- A full open-source implementation pipeline [1], based on the Tensil framework [2], for designing, training, evaluating and deploying DNN backbones using few-shot learning on FPGA SoCs,
- A fully-functional demonstrator: the presented methodology was deployed to a low-power, low-latency demonstrator trained on the MiniImageNet dataset with a systolic-array-based architecture. The proposed system has a latency of 30 ms while consuming 6.2 W on the PYNQ-Z1 board.

The work developed here was published at the 57^{th} IEEE International Symposium on Circuits and Systems (ISCAS 2024) [59].

5.2 Embedded Image Classification with Few Data

So far, we have demonstrated that cross-device systems often face hardware constraints and limited data availability. These limitations encourage their participation in federated training loops and the use of compression techniques. In Section 4.5, we explored the use of pre-trained models to bridge gaps in federated learning, drawing inspiration from transfer learning scenarios. Another possible situation is when a device cannot participate in federated learning. Be it in the form of limited communication capabilities or high privacy concerns, where the device cannot expose its data. In such cases, pre-trained models can help close the performance gap. However, federated learning allows models to train on data classes that are inaccessible to individual clients, which boosts overall performance. While fine-tuning (discussed in Section 4.3.1) a pre-trained model can be a solution, it does not enable the model to perform well on classes that have not been directly collected. To avoid repeated fine-tuning and address the limited data availability, Few-Shot Learning (FSL) [15] is a promising alternative. FSL leverages pre-trained models to perform effectively on new tasks or unseen data with only a few samples.

One of the advantages of FSL is its ability to learn new classes without relying on SGD-based techniques, reducing both memory and computational demands. In the context of embedded systems, several options are available for deploying deep learning models. Two prominent choices are embedded GPUs, like the NVIDIA Jetsons, and dedicated architectures using FPGAs. NVIDIA Jetsons are ideal for solutions that require

1. https://github.com/brain-bzh/PEFSL
2. https://www.tensil.ai/

easy deployment and low power consumption, as they utilize the same ecosystem as traditional GPUs. However, FPGAs can offer lower latencies and greater flexibility in handling different data formats, as discussed in Section 3.1.4. These are crucial elements for real-time systems, such as those we can find in a cross-device setting. Moreover, FPGAs' reconfigurable hardware allows for the implementation of specialized functionalities, enhancing the device's capabilities. For example, in the context of federated learning, the authors in [174] implemented a hardware-based homomorphic encryption function to enhance security.

This chapter proposes a pipeline for deploying an FSL model on an SoC platform that incorporates an FPGA. Building on the concepts of heterogeneity discussed in Section 4.4, we aim to find the best trade-off between model architecture and available resources. Although our methodology is designed for FSL deployment, it can also be adapted for models trained using federated learning.

The chapter is organized as follows: First, a detailed explanation of the FSL method we used, and then an introduction to our chosen platform and deployment framework. Finally, we demonstrate how the pipeline was constructed to co-design a ResNet architecture for implementation on a PYNQ-Z1 board.

5.3 EASY Few-shot Learning

Few-shot learning is a compelling solution for scenarios where only a small number of labeled examples are available. It allows to profit on public and general datasets to build powerful feature extractors. In this section, we use our proposed framework as a general view of the few-shot learning algorithm and as a way to introduce it. The specific training routine, Ensemble Augmented-Shot Y-shaped Learning (EASY), is discussed at last.

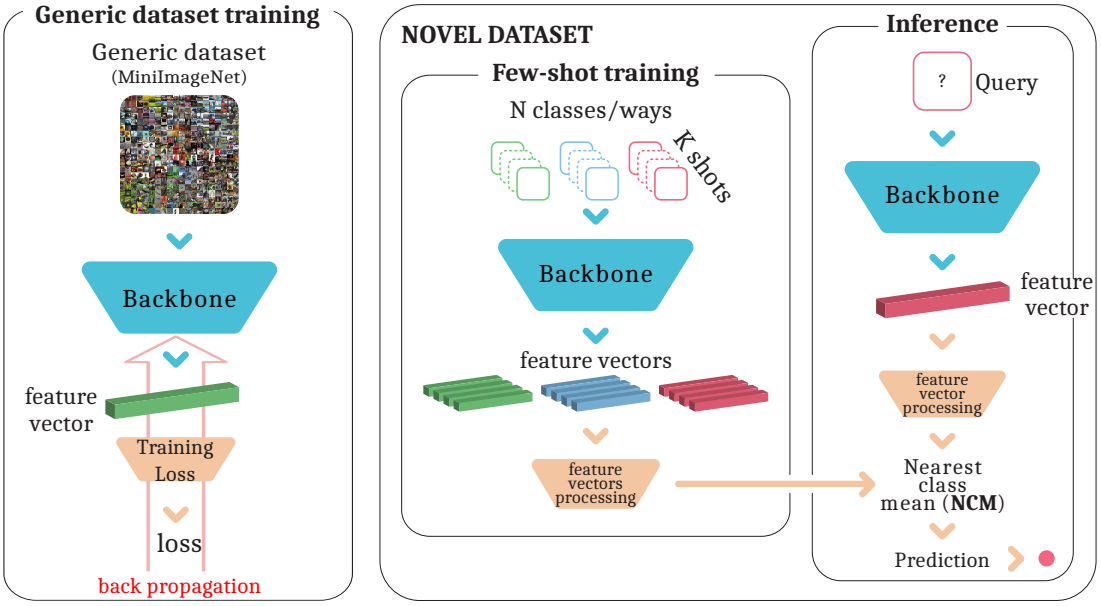


Figure 31 – PEFSL few-shot learning method. Source: [59] © [2024] IEEE.

5.3.1 Few-Shot Learning

FSL involves classifying examples from unseen classes with a small number of annotated examples. Initially, this may seem counterintuitive, as deep learning is known for

excelling when trained on large datasets where it generalizes effectively. The power of FSL comes from using specific training routines to build backbones to be universal feature extractors. With features of even unseen classes being well separated in the feature space, FSL relies on techniques to compare these features to classify the input. As such, it is fundamental to have a training algorithm that focuses on generalization rather than finding a better fit for one series of examples.

The FSL pipeline used in this work is illustrated in Figure 31. The first step, called **generic dataset training**, involves training a deep learning backbone, following EASY [15]. Few-shot datasets are divided into *base* and *validation* sets, with the latter used to assess generalization performance. For FSL, the classes in the validation set are distinct from those in the base set [116], so the method can evaluate the model's generalization capacity to unseen classes. After training, the backbone is kept frozen, serving as a universal feature extractor.

Next, FSL performance is evaluated on a third dataset, called the *novel* dataset, containing thousands of few-shot episodes [99]. Each episode includes several classes, called *ways*, each with a small number of labeled examples, or *shots*, and some unlabeled ones, or *queries*. This process is expressed in the "Few-shot training" and "Inference" parts of the diagrams in Figure 31. The model's performance is determined by its classification accuracy of queries with few available shots averaged over thousands of episodes. The number of shots and ways is benchmark-dependent. In practical terms, during the "Few-shot training" step, the system receives unseen shots of different classes. These shots are averaged to compose the class "anchors" in the feature space. In the "Inference", the same backbone extracts the features of a query, and a Nearest Class Mean (NCM) classifier [63] compares the extracted feature to the class anchors to classify the query.

For our work, we solve what is called an inductive [146] problem, where queries are not available beforehand, contrary to transductive [112] problems, where all queries are known in advance.

5.3.2 EASY training routine

In the EASY method, the training routine for backbones involves several techniques designed to enhance robustness and generalization performance. The backbone models follow a Y-shaped architecture, where they are trained using two parallel parts: one with a standard classification loss and another with a self-supervised learning loss. The self-supervised loss induces the model to recognize rotations applied to input samples, which improves the backbone's ability to extract meaningful features from the input. Additionally, the backbone training includes the use of manifold mixup [116], a regularization technique that performs linear interpolations between feature vectors in the latent space. This encourages the model to learn more generalized and robust feature representations.

The training process adjusts the learning rate during training, where the learning rate is gradually reduced during each iteration. After each iteration, a warm restart occurs with a slightly reduced maximum learning rate to help with a smoother convergence and to avoid overfitting. Another important element in this training routine is the use of an ensemble of backbones; each is initialized differently with random seeds. These backbones are trained independently, and their outputs are concatenated to form a "richer" and more diverse feature representation, which boosts performance without significantly increasing the number of parameters or complexity.

5.4 A Reconfigurable Platform

Embedded systems for deep learning applications require efficient, low-power solutions capable of executing deep learning models in real-time. As reconfigurable accelerators, FPGAs offer a flexible and highly parallelizable processing design at low power. Their reconfigurability offers space for tailored solutions to meet the different constraints of embedded systems. This section briefly introduces a System-On-Chip (SoC) platform based on a CPU and an FPGA. We then discuss how deep learning models can be implemented on FPGAs, specifically the Tensil framework used for the proposed solution.

5.4.1 What is an SoC?

Integrating System-On-Chip (SoC) and Field Programmable Gate Arrays (FPGAs) enables highly flexible and efficient embedded systems. An SoC with an FPGA combines traditional processing elements, such as a general-purpose CPU, with programmable logic in the FPGA, all within the same silicon die. This setup allows, for example, the CPU to manage software-defined tasks while the FPGA fabric can be programmed for hardware acceleration, making it ideal for computationally intensive functions like real-time data processing or implementing custom digital architectures. An introduction to common FPGAs elements is given in Appendix A.1.

In this context, one notable example of an SoC is the AMD-Xilinx Zynq family, where the SoC includes an ARM-based CPU, DSP cores, and FPGA fabric on a single chip. The CPU handles control and application-level logic, while the FPGA fabric can be customized to accelerate specific tasks, such as deep learning inference or signal processing.

In the following work, we have specifically used the PYNQ-Z1 [3] development board from the manufacturer Digilent. This entry-level board has a dual-core Cortex-A9 CPU, 512 MB of memory, and an Artix-7 equivalent FPGA.

5.4.2 Deploying Models on an FPGA

Converting a deep learning model, typically described in high-level languages like Python or as a representation graph like the Open Neural Network Exchange (ONNX) [13], into an efficient hardware implementation is challenging. This process involves mapping the diverse layers and operations of the model, each with varying dimensions, data dependencies, and computational flows, onto sequential and parallel hardware architectures [100]. As model complexity and size increase, so does the deployment difficulty. The most common approach is to design a general deployment framework supporting a group of layers and architecture families that can be used to automatize the process. These frameworks usually differ in terms of hardware architectures due to different design trade-offs. We provide in Appendix A.2 a description of different frameworks to deploy a deep learning model to FPGAs.

For this work, we chose to use the Tensil framework due to its easy workflow and adaptability to the solution we proposed. Another crucial point was the direct support for our target FPGA board.

Tensil is composed mainly of a systolic-array [98] core for matrix-multiplication operations, with side functional blocks for other operations like activation and pooling functions and a memory block working as a scratchpad. In order to use Tensil, one must

3. The board contains the ZYNQ XC7Z020-1CLG400C chip.

use its own compiler that parses an ONNX description to Tensil's own representation, which is used to program the hardware IP of Tensil.

5.5 PEFSL: An open-source Pipeline for Embedded Few-Shot Learning

This section describes the implementation pipeline developed to design a low-complexity model architecture and train it using the EASY routine.

5.5.1 Design Space Exploration for FPGA Implementation

The complexity of backbones in terms of memory and computational cost makes it challenging to find the correct trade-off between accuracy and performance. Rapid adaptation to new tasks with minimal resources and real-time constraints for image classification is crucial. Therefore, careful attention must be given to the design of efficient backbones.

From the description of a ResNet model in Section 2.1.3.2, we studied its architecture to propose a low-complexity combination of the Basic Blocks. This type of network is particularly efficient when deployed into an FPGA, as it has been vastly studied in the literature. In FSL, the ResNet-12 model version has allowed for close to the State-of-The-Art (SoTA) [15] performance, although its size. So, we part from this model to explore the hyperparameters that can impact its performance and deployment.

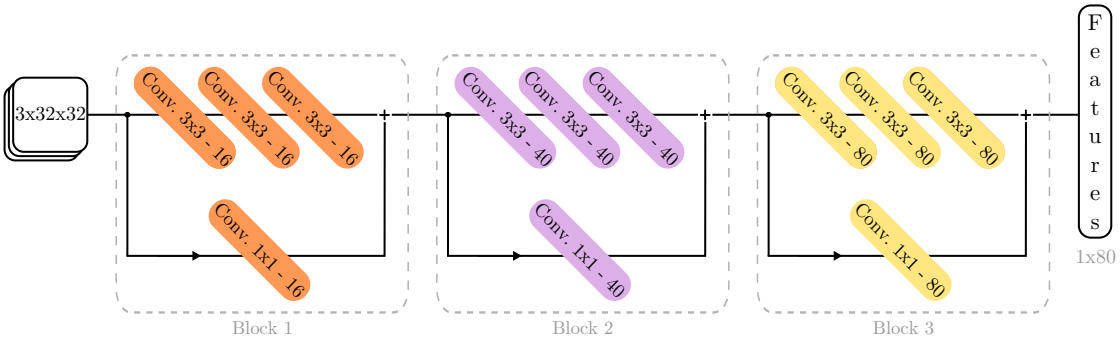


Figure 32 – Structure of a ResNet-9. Convolution layers on the main branch have 3×3 kernels with 16, 40, and 80 output channels, respectively. The convolution layers on residual paths have 1×1 kernels. Batch-normalization and activation function layers are omitted for simplicity. The actual layer configuration follows the Basic Block described in Section 2.1.3.2.

5.5.1.1 Hyperparameters

We list the main hyperparameters that influence the final system performance and complexity:

Network Depth - We opt for two shallow ResNet architectures, the ResNet-9 and ResNet-12. The ResNet-9 is essentially a ResNet-12 with the final residual block removed. Since ResNet-9 is less computationally demanding and shallower, it is expected to have lower accuracy than ResNet-12 for more complex tasks. Figure 32 presents the proposed structure of a ResNet-9.

Training and test image size - The size of training images significantly influences computational load and accuracy. Smaller images, such as 32×32, contain less information compared to larger 100×100 images, but they require fewer computational operations. The choice of both training and testing image sizes plays a crucial role in determining the model's accuracy.

Downsampling - The intra-block downsampling effect can be achieved in two ways: either by changing the stride of the final convolution in each block from 1 to 2 or by using max-pooling. A stride of 2 and a 2×2 pooling size achieve the same dimension reduction.

Number of feature maps - For convolution layer architecture, we use the number of filters (output channels) in the first convolution layer as a hyperparameter and scale the subsequent layers accordingly.

5.5.2 Training

For training, we use the MiniImageNet dataset proposed with the EASY routine. It contains 64 base classes, 16 validation classes, and 20 novel classes. Each class has 600 examples with 84×84 resolution. We focused on the 5-ways, 1-shot setup. The MiniImageNet dataset is used due to its highly diverse classes, allowing for excellent generalization to new classes.

5.5.3 PEFSL pipeline

To explore the search space of the previously defined hyperparameters for training and network architectures, we created the PEFSL, a modular pipeline for training, compiling, synthesizing hardware, and deploying a few-shot learning application on an FPGA SoC. This pipeline integrates various tools, which are described in Figure 33.

Part A - It handles the training routine of the backbone, as described in Section 5.3.2. The EASY training routine incorporates the ResNet-9 and ResNet-12 architectures and their variants for training CNNs on FSL tasks. The PyTorch model is converted into ONNX format, and we simplify the model using the ONNX simplifier tool for efficient optimization. Finally, the ONNX model is compiled via the Tensil framework. With a description of the underlying FPGA resource (`.tarch` file), which defines features like the number of Processing Elements, data format, and memory size, these three initial scripts automatically generate the latency of the neural network on the specified architecture. This enables design space exploration of neural network architectures and training methods, as shown in Figure 34.

Part B - It focuses on the architecture's compilation, which produces Register Transfer Level (RTL) files for the Tensil accelerator Intellectual Propriety (IP). These RTL files are used in part **C**, which generates project files for the AMD-Xilinx Vivado tool to create the bitstream for the FPGAin the SoC. The resulting intermediary files (bitstream and Tensil model) are utilized in the main script, which employs the PYNQ driver for data transfer between the CPU and FPGA.

The final result is a platform that executes the chosen CNN model in the FPGA in 16-bit fixed-point, thanks to the Tensil framework, while the CPU side executes the model in 32-bit floating point.

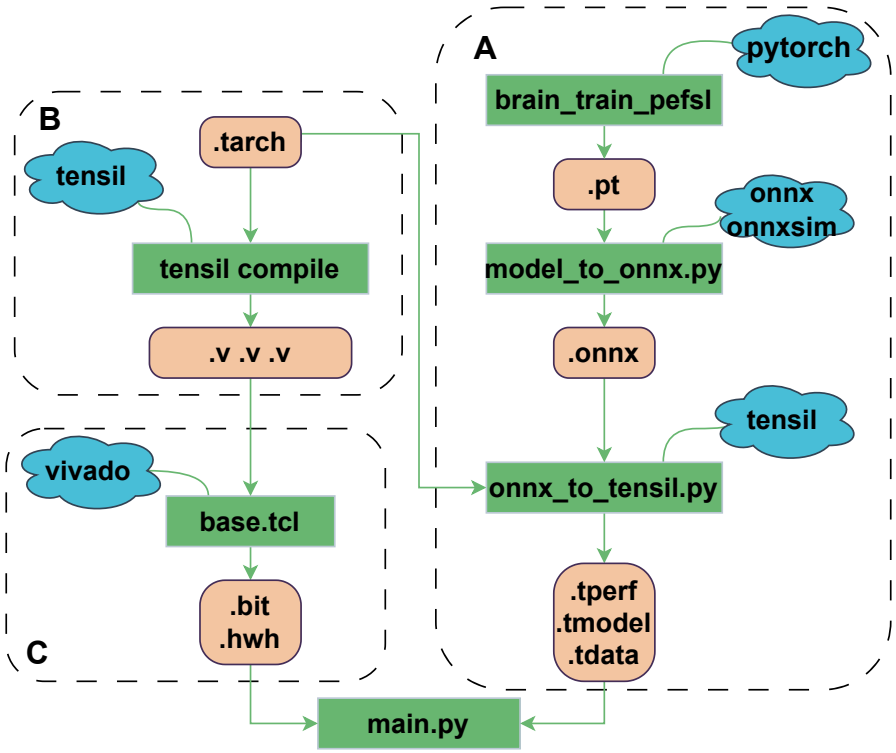


Figure 33 – Modular pipeline for the deployment of a few-shot learning system on an FPGA SoC. Source: [59] © [2024] IEEE.

5.5.4 Exploration Results

The training results are illustrated in Figure 34, where we thoroughly explored the hyperparameter search space. Each network was compiled using Tensil to determine the number of cycles required for inference. To maintain a smooth video stream (above 10 FPS), 32×32 images were necessary. Therefore, we present results for this resolution alongside the MiniImageNet standard 84×84 resolution in a 5-way, 1-shot setup. A key observation is that, for the 32×32 resolution, ResNet-9 models (empty marks) outperform ResNet-12 models (filled marks) in accuracy despite having fewer layers and parameters. We hypothesize that with 32×32 images, the final feature map dimensions in ResNet-12 are too small to utilize by the downstream NCM classifier effectively.

Another important finding is that, for a target resolution of 32×32, the training images should also be of size 32×32 (circles). Models trained on larger images, such as 84×84 and 100×100, exhibit significantly lower accuracy, even though resizing to 32×32 during training results in some loss of information. This might suggest that more advanced data augmentation techniques or better generalization metrics could further improve accuracy [14].

We also found that using convolutions with a stride of 2, as opposed to max pooling layers for reducing intermediate representation dimensions, reduces the number of operations required. This is indicated as *strided* in Figure 34. This change not only reduces latency but may also enhance accuracy. Finally, adjusting the number of feature maps in the first layer, which scales the network's width, offers a trade-off between latency and accuracy.

It is important to note that the primary objective of this project is real-time few-shot classification, where minimizing latency takes precedence over achieving the highest

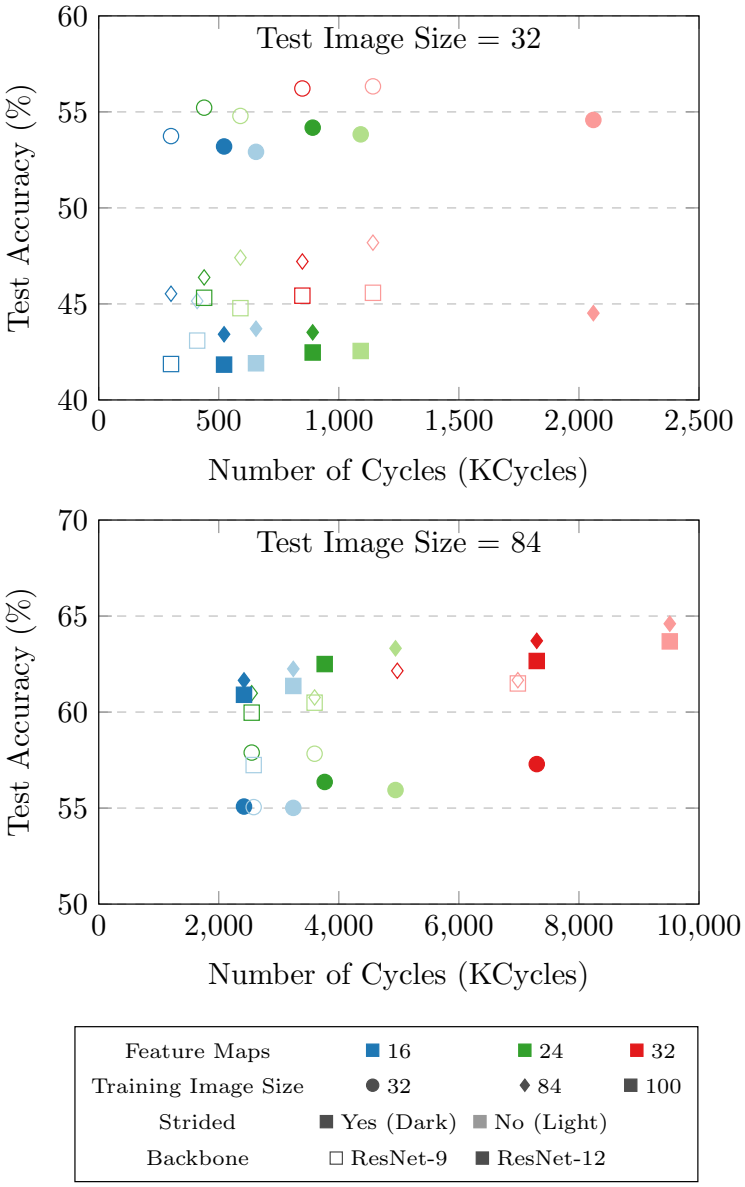


Figure 34 – Accuracy and Latency Trade-off: Graphs depict tests on 32×32 (top) and 84×84 (bottom) images. Different feature map configurations are denoted by unique colors, while distinct training image sizes are represented by different shapes. We also investigate the impact of strided architectures, differentiated by dark and light colors. Additionally, we vary the backbone architecture from ResNet-9, with empty forms, and ResNet-12, filled forms. Source: [59] © [2024] IEEE.

accuracy. As a result, using 84×84 images for inference on the PYNQ-Z1 board is not feasible, as all networks at this resolution exceed 2000 KCycles. Where 1 KCycles indicates that the Tensil accelerator needs 1000 iterations to execute the backbone, following their computing model. We focus on networks with fewer than 1000 KCycles. There is a 2% difference in accuracy between the best-performing network in terms of accuracy and the best-performing one in terms of latency.

In conclusion, the ideal trade-off for our application lies in the top-left corner of the graph, where configurations with acceptable accuracy and the lowest latency are found. We have selected the strided ResNet-9, trained on 32×32 images with 16 feature maps, and using 32×32 images during inference, represented by the empty blue circle on

the first graph in Figure 34. This work achieved 54% accuracy on the MiniImageNet dataset for the 32×32 resolution in the 1-shot, 5-ways scenario.

In [167], authors proposed to apply a FSL algorithm to federated learning. At the end of their training algorithm, each client has a personalized backbone, which was trained to generalize all clients' data and fine-tuned to its own dataset. PEFSL could be integrated into this method to provide a deployment platform.

5.5.5 Improvement of the Hardware Implementation

Now that we have chosen the model configuration for our demonstrator, we wanted to see if the Vivado and Tensil tools could minimize the latency of this same architecture.

By increasing the FPGA clock frequency, we increase the number of operations per second, thus reducing latency, although not proportionally. This is due to Tensil's computational model and memory transfer architecture. We notice that at 125 MHz, our ResNet-9 is the most performant in terms of latency, reaching 30 ms. However, it consumes more power than the same network at a clock rate of 50 MHz. Nevertheless, its consumption remains reasonable for an embedded application at 3.9 W.

We also tried to adjust Tensil's architecture parameters. We maximized the size of the systolic array according to the available resources and attempted to use 8-bit fixed point data. However, we could not configure Vivado to route 8-bit words on DSPs, and we exceeded the available LUT count. This limitation is connected to our limited knowledge of the hardware description language Chisel [12], which Tensil is made of. Based on these facts, it is clear that the best network choice is the ResNet-9 at 125 MHz, reducing latency by 8 ms compared to 50 MHz, even if it sacrifices a few tenths of a watt at 3.9 W.

An important future direction outside the Tensil's limitation would be to explore more efficient quantization schemes. For example, [100] uses 8-bit for the first and last layers, with 4-bit for the intermediate layers. This allowed for better resource utilization while conserving accuracy.

5.5.6 Comparison with other hardware implementations

There is limited literature specifically addressing FSL on FPGAs or in embedded systems. For example, in a recent study [108], researchers proposed a method for recognizing pests using FSL on an FPGA. They achieved a processing speed of 2 frames per second on a PYNQ-Z1.

To demonstrate that the hardware resources and processing time achieved with Tensil's framework align with industry standards, we conducted a benchmark and presented the results in Table 11. We have configured the array size of our systolic array to 12, which is the maximum size for our setup. The FPGA frequency is set to 125 MHz. With this configuration, the latency of the inference process is 30 ms. We compared our approach with deep learning models implementations designed to classify images on the CIFAR-10 dataset. With images of 32×32 pixels, it can be effectively processed by the ResNet-9 backbone we selected. To be fair, we implemented the additional downstream linear layer, to be comparable with other works. We specifically focused on implementations for the Zynq-7020 (z7020) chip like the one in the PYNQ-Z1. The results in Table 11 show that Tensil's implementation provides similar processing time and accuracy for comparable resources, confirming the effectiveness of our backbone's implementation. It's worth noting that each study used a different model.

Table 11 – CIFAR-10 inference on z7020 FPGA. Source: [59] © [2024] IEEE.

Work	Prec. [bits]	LUT	BRAM [36 kb]	FF	DSP	Latency [ms]	Acc. [%]
[24] hls4ml	8-12	28544	42	49215	4	27.3	87
[24] FINN	1	24502	100	34354	0	1.5	87
[172]	1-2	23436	135	-	53	1.1	86
[92]	16	15200	523	41	167	109	-
Ours	16	15667	59	9819	159	35.9	92

5.5.7 Demonstrator

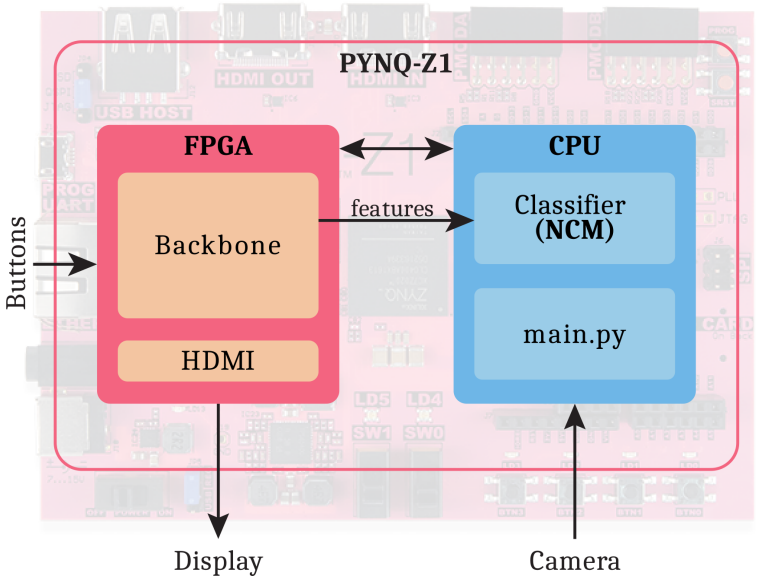


Figure 35 – Schematic of the proposed system. Source: [59] © [2024] IEEE.

We developed a standalone demonstrator housed in a compact box to demonstrate how easily this work can be applied in an industrial setting. Figure 35 presents a schematic of the demonstrator, which includes a PYNQ-Z1 board, an 800×540p HDMI screen, a 160×120 pixels camera, and a 10,000 mAh battery, which provides a battery life of 5.75 hours during inference. The demonstration features on-screen indicators to enhance user experience, and the system achieves an average of 16 FPS during inference. The network used is a ResNet-9 with 16 feature maps, and inference is performed on the FPGAat 125 MHz, implemented in a 16-bit fixed-point format with 8 bits for the integer part. The entire system, including the SoC, camera, and screen, consumes 6.2 W of power. Most of the FPGAresources are utilized by the Tensil hardware accelerator, and the HDMI AMD-Xilinx IP is implemented on the FPGA side. Meanwhile, all software tasks, such as pre-processing, post-processing, and image classification (NCM), are handled by the CPU. The demonstrator also features interfaces for the camera and buttons for live control. Currently, the NCM classifier runs on the CPU, but in a future version, it could be moved to the FPGA. A picture of the actual demonstrator can be seen in Figure 36.

Figure 36 – A picture of the demonstrator.

5.6 Recapitulation

This section showed our proposal for the first implementation of an inductive FSL system on an FPGA SoC, allowing for fast inference and low power consumption. PEFSL is a fully open-source implementation pipeline that allows for the design and deployment of a deep learning model, as well as training and deploying it on an embedded system.

Chapter 6

Conclusions and Perspectives

Contents

6.1 Conclusions

In this thesis, we have delved into the development and challenges surrounding model compression in deep learning, with a particular focus on federated learning and embedded systems. We explored how compression can be used to reduce the communication costs of federated learning, which arise once participants need to train their own model and continuously exchange model weights with an orchestration server. Additionally, we demonstrated a co-design strategy to leverage a pre-trained model for an efficient inference system.

In Chapter 2, we briefly introduce deep learning for image classification, along with details on the various architectures and datasets used in this work. Building on this foundation, we explored the different characteristics of federated learning and the challenges within the field. Many of the issues come from the privacy mechanism of keeping data locally. To address this, several techniques have been developed to compensate for the unknown distribution of client data, with most efforts aimed at reducing convergence time and bridging the performance gap with centralized systems. However, clients with heterogeneous hardware capabilities pose a significant challenge, as resource differences limit their fair integration into the framework. While new optimization strategies are crucial to narrow the performance gap, system-level approaches are also necessary to include a broader range of clients. This is particularly important because federated learning is designed as a privacy-aware, collaborative framework that allows multiple private datasets to train a single model collectively. Because of this, this thesis focused on addressing the hardware limitations of different clients by leaning on deep learning model compression techniques.

Chapter 3 begins with a review of quantization and pruning, two classic model compression techniques. For each approach, we highlighted the most commonly used methods and the hardware execution impacts of running a quantized or pruned model. Other promising compression techniques are also briefly discussed, as they are also used to address communication and computational challenges in federated learning. The federated learning literature has extensively explored the use of compression techniques both post-training and during training to tackle hardware heterogeneity. However, many proposed methods are difficult to integrate with other techniques. As a result, while these works often reduce communication and/or computation costs, they tend to overlook the potential benefits of combining them with optimization methods to improve overall performance. In federated learning, a model's accuracy is closely tied to the strategy used for a given scenario. Therefore, any compression technique should minimize its interference with optimization strategies to allow seamless integration without further modifications.

In Chapter 4, we addressed the challenge of communication costs in federated learning through two distinct solutions. First, we focused on magnitude pruning, implementing a double-sided compression technique where both clients and servers apply weight-magnitude pruning before transmitting model updates. Pruning increases message sparsity, and the resulting sparse models were then compressed using the ZIP algorithm, which is based on entropy encoding, leveraging models' sparsity. The method is easily implementable and provides a robust framework for federated learning tasks, reducing message sizes by 50% with less than 1% accuracy loss. We also proposed an implementation of Low-Rank Adaptation (LoRA) to federated learning. In this approach, convolutional layers are instantiated with low-rank adapters controlled by a hyperparameter called the rank, which allows a balance between compression rate and accuracy. This flexibility allows us to take into account different communication cost

constraints. We reached a 4.8 times reduction of message sizes while having a 1% accuracy loss.

Additionally, we extended the proposed technique by applying asymmetric quantization to LoRA layers, using 8-bit integer words to reduce communication costs further. The quantization scheme allowed us to reduce the message sizes further, reaching up to 18.6 times of compression with up to 1% accuracy loss. Notably, this technique had low degradation of both convergence rates and model accuracy. Another point in this work was considering smaller models as a viable alternative to compressing larger, more cumbersome models. We also reviewed the impact of model size versus compression in federated learning. We concluded that a critical step in applying compression techniques should involve a "sanity check" with smaller models designed to meet specific computational or communication constraints. Also, personalizing compression rates per client is fundamental in allowing constrained devices to participate in federated learning. Furthermore, using pre-trained models or public datasets can effectively close the performance gap in federated learning, particularly in scenarios with non-IID data distributions. Pre-training offers a strong initialization point for federated models, accelerating convergence and improving accuracy, all while reducing the overall training burden on clients.

In Chapter 5, we presented the development of an implementation and deployment pipeline for embedding a few-shot learning algorithm. In systems with limited communication capabilities, one potential solution to leverage on-device data is through the use of pre-trained models and adaptation techniques. We built this pipeline around a few-shot learning algorithm, envisioning a device with limited data that cannot integrate federated learning. Using the EASY [15] method as a foundation, the pipeline trains a backbone model to generalize over the MiniImageNet dataset. The goal of the training routine is to create a robust general feature extractor whose features can be utilized for downstream tasks. We conducted a design-space exploration to optimize the balance between latency and accuracy for deploying the backbone on a PYNQ-Z1 board. This platform enabled us to exploit the FPGA for model inference using the Tensil inference framework while the embedded CPU handled an image classification application. The Tensil's IP, based on a systolic-array architecture, executed the inference phase of an adapted ResNet-9 model. The extracted features were then passed to the CPU, where a Python-based application implemented a Nearest Class Mean classifier. To show the potential of our open-source Pipeline for Embedded Few-Shot Learning (PEFSL), we developed a working demonstrator featuring a camera, an HDMI screen, and a battery to create a fully portable, low-power (6.2 W), low-latency (30 ms) device. This demonstrator runs a few-shot learning image classification task with a graphic user interface and buttons for user interaction. We trained and deployed a compact model tailored to the device's hardware capabilities. While our approach was effective, further improvements could be made by using more powerful devices or incorporating a more advanced hardware accelerator framework.

6.2 Perspectives

6.2.1 Compressed Partial Training

This work has focused on applying compression techniques to reduce communication costs with minimal impact on computational expenses. A first interesting extension of these works could be its integration into the concept of "partial training," as seen in FedDropout [28] and FedRolex [5], discussed in Chapter 3. In this approach, clients train

on subsets of the global model, tailoring the submodel to their available resources. This strategy addresses both communication and computational challenges and is still compatible with compression methods. However, as shown by SLT [135], these techniques still result in models that underperform compared to "simply a smaller model." SLT introduced an adaptation of partial training, where the initial communication cost is low but increases with each round. As an alternative, recent studies such as ScaleFL [86] have integrated the "early-exit" concept [156] into client models. Early-exit models incorporate several classification heads at different points, enabling premature classification without executing the entire model. Still, both methods involve variable dynamic costs, and eventually, the full global model must be exchanged. A potential future direction could be first to minimize communication costs using compression techniques, followed by exploring iterative early-exit strategies, as outlined by SLT. This would allow for a new trade-off between partial training and message compression message, adding relaxations in previous methods and enabling partial training to close the gap to "just smaller" models. It is expected that such an approach would need to quantify the impact of compression on partial training first.

6.2.2 Efficient Inference Platform

A second perspective would be to use the federated learning training to adapt each client's model to its target hardware, obtaining a personalized compressed model per client. Works, such as [167], train federated learning clients to use both global (shared) and local (not shared) parameters. Other studies, like [39], have explored model personalization by employing a global backbone while using the classification head as local parameters, allowing the personalization of each client's model. In both cases, after training, each client possesses a backbone that generalizes across all clients' data and personalized classification heads. For more efficient deployment, a quantization objective, as seen in FedMPQ [33], could be incorporated, enabling clients to obtain mixed-precision models tailored to their own constraints. The resulting mixed-precision personalized models could then be implemented in a pipeline such as PEFSL. Additionally, Tensil, in its current state, is a limited deployment framework for deep learning models. Redesigning Tensil with a focus on mixed-precision backbones could create a deployment framework optimized for accelerating a general feature extractor, with the features processed by the device's CPU. An example of this is the NeuroCorgi application by CEA-LIST, as seen in [121]. As most works in federated learning are focused on training performance, a post-training objective would make the framework more practical for use in real-world scenarios.

6.2.3 Model Heterogeneous Federated Learning

A final, more far-fetched idea is the exploration of self-defined clients' models. This would require aggregation methods adapted to account for architectural differences. This approach would directly tackle many open challenges in federated learning, as discussed in Chapter 2, offering a promising path for future research in heterogeneous distributed frameworks. The main issue to be studied would be how different architectures can learn together and benefit each other. One possible direction would be using knowledge distillation, as seen in FedDF [111], discussed in Chapter 3. However, their method relies on the server having a proxy dataset whose task is close to the client's data. Although possible, a more exciting approach could be the use of large language or multimodal models to guide clients' knowledge fusion. The necessary steps would require the study of combining ensemble knowledge distillation [54] with federated learning strategies while not having access to clients' data distribution. The idea could

allow for a complete heterogeneous framework capable of aggregating knowledge from all types of clients.

Appendix A

FPGA and Deep Learning Models

A.1 What is an FPGA ?

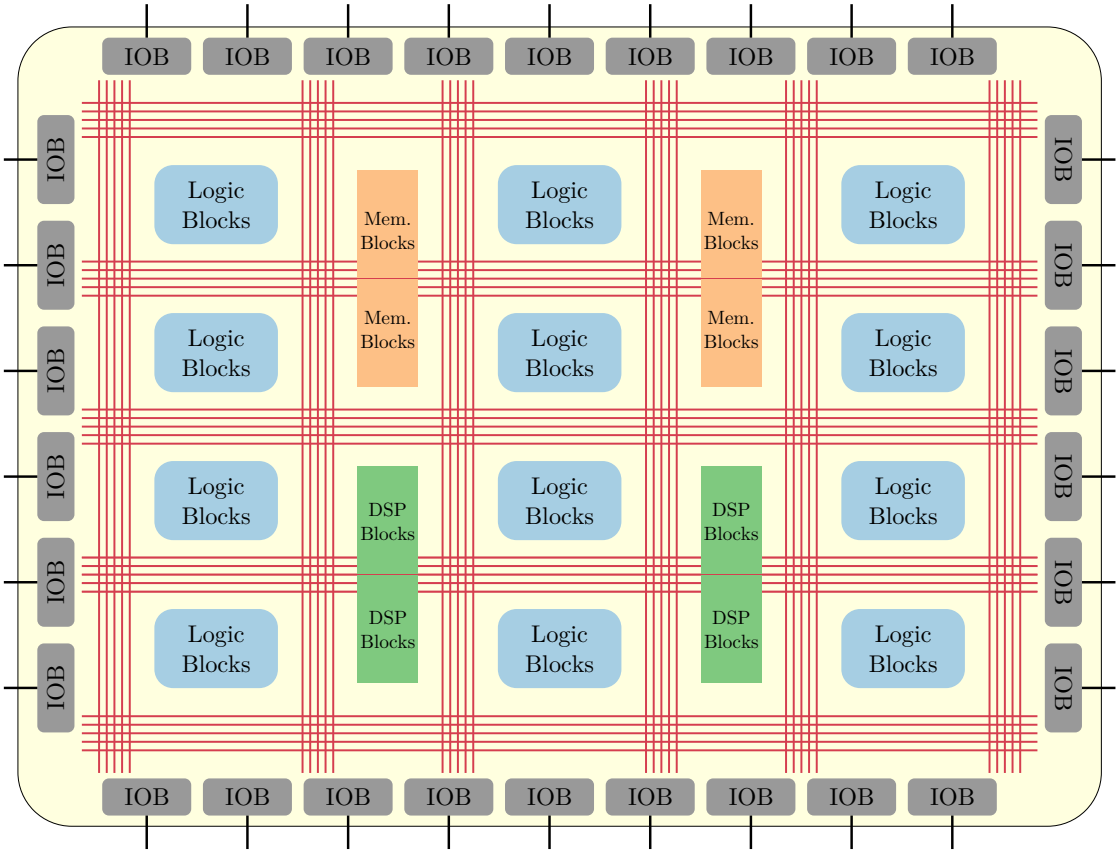


Figure 37 – A simplified block diagram of an FPGA. IOBs stands for Input/Output Blocks, Mem. Blocks for Memory Blocks and DSP Blocks for Digital Signal Processing Blocks.

A Field-Programmable Gate Array (FPGA) is a type of integrated circuit specifically designed to be reconfigurable by the user, enabling customized hardware functionality after manufacturing. Unlike ASICs, which have a fixed function, FPGAs offer a flexible hardware platform to perform different tasks. Although the physical layout of an FPGA is fixed, its reconfigurability comes from its architecture, which is composed of configurable logic blocks, interconnects, and I/O pins, as well as other specialized resources

such as Digital Signal Processing (DSP) and memory blocks. Figure 37 illustrates a simplified block diagram of an FPGA, with the key components: logic blocks, switch matrix (interconnect), DSP units, memory blocks, and Input/Output Blocks (IOBs). The switch matrix, composed of configurable routing paths (wires) and switching elements, allows signals to flow between different components of the FPGA. This matrix enables dynamic reconfiguration by directing signals across various blocks in a design-specific manner. The different functional blocks can be described as follows.

Logic Blocks are the fundamental units in an FPGA for implementing generic logic functions. These blocks typically consist of three main components: Look-Up Tables (LUTs), Multiplexers (MUXs), and Flip-FLop/Registers (FFs). LUTs are used to store precomputed truth table values, allowing them to quickly output results for small combinational logic functions by addressing those values. Registers are memory elements that hold values between clock cycles, enabling the implementation of sequential logic. MUXs are used for routing and selecting between different inputs, controlling the data flow. Figure 38 illustrates a possible configuration for a cell inside a logic block. In practice, a logic block consists of several interconnected cells connected using dedicated routing resources such as carry chains. The programmable nature of LUTs allows logic blocks to be highly flexible and capable of implementing both combinational and sequential functions, making them key components in FPGA architectures.

Memory Blocks are specialized storage units in FPGAs designed to hold larger amounts of data than individual registers. Unlike FFs, memory blocks provide a relatively large addressable space optimized for high-speed data storage and retrieval. These blocks can be configured as single-port or dual-port memories, supporting parallel data access for increased throughput. They are typically used to store large data structures such as buffers, lookup tables, and state information. Their capacity and reconfigurability make memory blocks essential for low-latency data-intense applications, such as image processing, deep learning, and communication systems.

DSP Blocks are another category of specialized hardware blocks in FPGAs. These blocks are optimized for high-performance arithmetic operations, such as addition, subtraction, and multiplication. These operators are critical for signal processing, control systems, and machine learning applications. DSP blocks often contain dedicated hardware for fast MAC operations, allowing them to perform complex mathematical computations with low latency. The optimized architecture of DSP blocks makes them highly efficient for handling computationally intensive tasks, freeing up general-purpose logic resources for other functions.

Input/Output Blocks interface the internal FPGA fabric and the external world. These blocks are responsible for driving signals in and out of the FPGAand supporting various communication protocols and standards. IOBs vary in functionality depending on the type of connection required, ranging from simple General-Purpose Input/Output (GPIO) pins to high-speed transceivers for interfaces such as PCIe, Ethernet, or DDR memory.

Thus, through programmable interconnects and functional blocks, an FPGA can be tailored to implement a wide range of digital systems, from simple logic circuits to complex processing architectures, all without requiring changes to the physical hardware itself. Its reprogrammable capabilities allow for high parallel processing, low latency, and deterministic architectures that are essential for real-time applications.

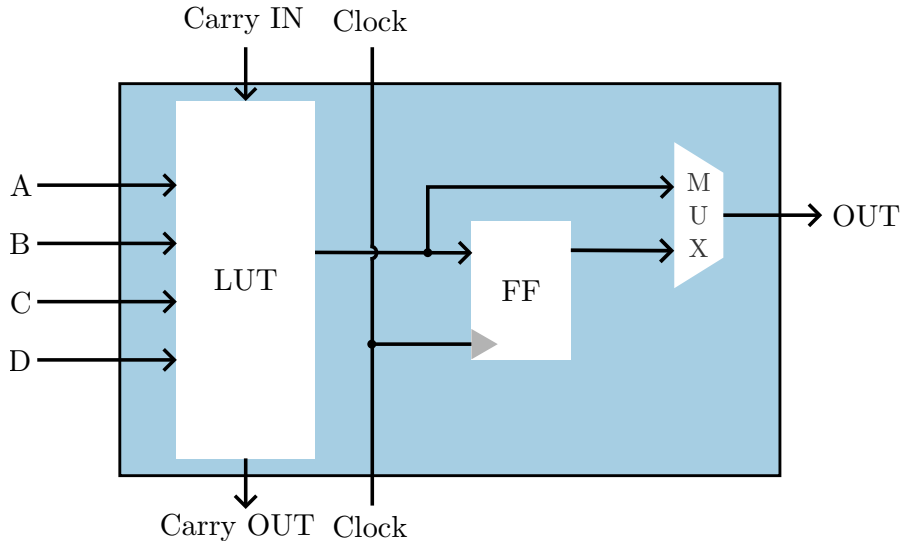


Figure 38 – A generic cell in a logic block diagram. With a 4-input Look-Up-Table (LUT), a Flip-Flop (FF), a Multiplexer (MUX), and a Clock signal.

A.2 Deployement Frameworks

The two more common hardware architectures are Deep Processing Units (DPUs) and dataflow architectures [35].

A DPU architecture revolves around specialized cores designed to accelerate matrix operations, such as convolutions and linear layers. These cores are often built using systolic arrays [98], a type of parallel computing architecture that is highly efficient in performing matrix multiplications. In addition to matrix operations, DPUs typically include dedicated blocks for activation functions, like ReLU and softmax, normalization layers, and memory space, to handle intermediate results. This architecture closely resembles GPUs, where layers of the neural network are compiled/mapped onto the DPU and executed sequentially, often layer by layer, making them flexible to support different sizes of models but still limited in terms of layer parallelism.

In contrast, dataflow architectures adopt a different approach by focusing on optimizing the flow and processing of data. These architectures focus on fine-grained operation parallelism and data flow, allowing the design to handle multiple layers or operations simultaneously. Frequently, these architectures are implemented in a pipeline fashion to maximize throughput by reducing memory movements between layers. Dataflow architectures allow for greater flexibility in deciding how computation and resources are allocated, contrary to DPU architectures. The cost of this flexibility comes in terms of resources, where the model to be deployed is more constrained by resources than DPUs.

The choice of framework depends on the deployment objectives—whether the priority is low latency, high throughput, or minimal power consumption. We present in a non-exhaustive list a few deployment frameworks:

- Vitis-AI [6]: A free closed-source software by AMD-Xilinx, based on a DPU family of architectures. Vitis-AI is a suite of solutions that one can use to deploy to AMD-Xilinx FPGAs. The software complies and optimizes deep learning models for its architecture, with dedicated tools for quantization and pruning.
- N2D2 [19]: It is a deployment framework adapted for multiple hardware targets

like FPGA, ASICs, GPUs, microprocessors, and others, from the CEA-LIST. The framework also proposes efficient training, quantization, and inference routines, making it an all-in-one solution that does not depend on classic tools like Pytorch. The tool has two closed-source IPs, DNeuro for FPGAs and PNeuro [30] for both FPGAand ASICs. Dneuro is a DPU-like architecture, while PNeuro is a dataflow architecture.

- FINN [22]: An open-source pipelined dataflow architecture by a research group of AMD-Xilinx. FINN allows for a fine-grain implementation of each layer and operator in the model. FINN supports mixed-precision deployment by combining its solution with the Brevitas [132] quantization framework.
- Tensil [157]: A systolic-array-based DPU open-source solution by TensilAI. It allows for implementing CNNs on FPGAs with a dedicated compiler and quantizer in 16- or 8-bit fixed point format. The framework focuses on being a lightweight solution for implementing models in low-end targets.

Bibliography

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. « Deep learning with differential privacy ». In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 308–318.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. « Gpt-4 technical report ». In: *arXiv preprint arXiv:2303.08774* (2023).

[3] Ankur Agrawal, Silvia M Mueller, Bruce M Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. « DLFloat: A 16-b floating point format designed for deep learning training and inference ». In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2019, pp. 92–95.

[4] Muhammad Pervez Akhter, Jiangbin Zheng, Farkhanda Afzal, Hui Lin, Saleem Riaz, and Atif Mehmood. « Supervised ensemble learning methods towards automatically filtering Urdu fake news within social media ». In: *PeerJ Computer Science* 7 (2021), e425.

[5] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. « FedRolex: Model-Heterogeneous Federated Learning with Rolling Sub-Model Extraction ». In: *arXiv preprint arXiv:2212.01548* (2022).

[6] AMD-Xilinx. *Vitis-AI*. `https://github.com/Xilinx/Vitis-AI`. 2024.

[7] Mohammad Mohammadi Amiri, Deniz Gunduz, Sanjeev R Kulkarni, and H Vincent Poor. « Federated learning with quantized global model updates ». In: *arXiv preprint arXiv:2006.10672* (2020).

[8] Adam Arany, Jaak Simm, Martijn Oldenhof, and Yves Moreau. « SparseChem: Fast and accurate machine learning model for small molecules ». In: *arXiv preprint arXiv:2203.04676* (2022).

[9] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. « Quarot: Outlier-free 4-bit inference in rotated llms ». In: *arXiv preprint arXiv:2404.00456* (2024).

[10] Umar Asif, Jianbin Tang, and Stefan Harrer. « Ensemble knowledge distillation for learning improved and efficient networks ». In: *ECAI 2020*. IOS Press, 2020, pp. 953–960.

[11] Sara Babakniya, Ahmed Roushdy Elkordy, Yahya H Ezzeldin, Qingfeng Liu, Kee-Bong Song, Mostafa El-Khamy, and Salman Avestimehr. « SLoRA: Federated parameter efficient fine-tuning of language models ». In: *arXiv preprint arXiv:2308.06522* (2023).

[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. « Chisel: constructing hardware in a scala embedded language ». In: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 1216–1225.

[13] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. `https://github.com/onnx/onnx`. 2019.

[14] Yassir Bendou, Vincent Gripon, Bastien Pasdeloup, Giulia Lioi, Lukas Mauch, Stefan Uhlich, Fabien Cardinaux, Ghouthi Boukli Hacene, and Javier Alonso Garcia. « A statistical model for predicting generalization in few-shot classification ». In: *2023 31st European Signal Processing Conference (EUSIPCO)*. IEEE. 2023, pp. 1260–1264.

[15] Yassir Bendou, Yuqing Hu, Raphael Lafargue, Giulia Lioi, Bastien Pasdeloup, Stéphane Pateux, and Vincent Gripon. « Easy—ensemble augmented-shot-y-shaped learning: State-of-the-art few-shot classification with simple components ». In: *Journal of Imaging* 8.7 (2022), p. 179.

[16] Reda Bensaid, Vincent Gripon, François Leduc-Primeau, Lukas Mauch, Ghouthi Boukli Hacene, and Fabien Cardinaux. « A Novel Benchmark for Few-Shot Semantic Segmentation in the Era of Foundation Models ». In: *arXiv preprint arXiv:2401.11311* (2024).

[17] Daniel J Beutel and et al. « Flower: A friendly federated learning research framework ». In: *arXiv:2007.14390* (2020).

[18] Sameer Bibikar, Haris Vikalo, Zhangyang Wang, and Xiaohan Chen. « Federated dynamic sparse training: Computing less, communicating less, yet learning better ». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 6. 2022, pp. 6080–6088.

[19] O. Bichler, D. Briand, V. Gacoin, B. Bertelone, T. Allenet, and J. Thiele. *N2D2-Neural Network Design & Deployment*. `https://github.com/CEA-LIST/N2D2`. 2024.

[20] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. « Understanding batch normalization ». In: *Advances in neural information processing systems* 31 (2018).

[21] Michaela Blott, Nicholas J Fraser, Giulio Gambardella, Lisa Halder, Johannes Kath, Zachary Neveu, Yaman Umuroglu, Alina Vasilciuc, Miriam Leeser, and Linda Doyle. « Evaluation of optimized cnns on heterogeneous accelerators using a novel benchmarking approach ». In: *IEEE Transactions on Computers* 70.10 (2020), pp. 1654–1669.

[22] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. « FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks ». In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23.

[23] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. « Practical secure aggregation for federated learning on user-held data ». In: *arXiv preprint arXiv:1611.04482* (2016).

[24] Hendrik Borras, Giuseppe Di Guglielmo, Javier Duarte, Nicolò Ghielmetti, Ben Hawks, Scott Hauck, Shih-Chieh Hsu, Ryan Kastner, Jason Liang, Andres Meza, et al. « Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark ». In: *arXiv preprint arXiv:2206.11791* (2022).

[25] John Bridle. « Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters ». In: *Advances in neural information processing systems* 2 (1989).

[26] Jingyong Cai, Masashi Takemoto, and Hironori Nakajo. « A deep look into logarithmic quantization of model parameters in neural networks ». In: *Proceedings of the 10th International Conference on Advances in Information Technology*. 2018, pp. 1–8.

[27] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. « Zeroq: A novel zero shot quantization framework ». In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 13169–13178.

[28] Sebastian Caldas, Jakub Konečny, H Brendan McMahan, and Ameet Talwalkar. « Expanding the reach of federated learning by reducing client resource requirements ». In: *arXiv preprint arXiv:1812.07210* (2018).

[29] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. « An analysis of deep neural network models for practical applications ». In: *arXiv preprint arXiv:1605.07678* (2016).

[30] Alexandre Carbon, J-M Philippe, Olivier Bichler, Renaud Schmit, Benoît Tain, David Briand, Nicolas Ventroux, Michel Paindavoine, and Olivier Brousse. « PNeuro: A scalable energy-efficient programmable hardware accelerator for neural networks ». In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1039–1044.

[31] Rich Caruana. « Multitask learning ». In: *Machine learning* 28 (1997), pp. 41–75.

[32] Hong-You Chen, Cheng-Hao Tu, Ziwei Li, Han-Wei Shen, and Wei-Lun Chao. « On the importance and applicability of pre-training for federated learning ». In: *arXiv preprint arXiv:2206.11488* (2022).

[33] Huancheng Chen and Haris Vikalo. « Mixed-precision quantization for federated learning on resource-constrained heterogeneous devices ». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 6138–6148.

[34] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. « A simple framework for contrastive learning of visual representations ». In: *International conference on machine learning*. PMLR. 2020, pp. 1597–1607.

[35] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. « Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks ». In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138.

[36] Brian Chmiel, Ron Banner, Gil Shomron, Yury Nahshan, Alex Bronstein, Uri Weiser, et al. « Robust quantization: One model to rule them all ». In: *Advances in neural information processing systems* 33 (2020), pp. 5308–5317.

[37] Yae Jee Cho, Luyang Liu, Zheng Xu, Aldi Fahrezi, Matt Barnes, and Gauri Joshi. « Heterogeneous LoRA for Federated Fine-tuning of On-device Foundation Models ». In: *International Workshop on Federated Learning in the Age of Foundation Models in Conjunction with NeurIPS 2023*. 2023.

[38] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. « Pact: Parameterized clipping activation for quantized neural networks ». In: *arXiv preprint arXiv:1805.06085* (2018).

[39] Liam Collins, Hamed Hassani, Aryan Mokhtari, and Sanjay Shakkottai. « Exploiting shared representations for personalized federated learning ». In: *International conference on machine learning*. PMLR. 2021, pp. 2089–2099.

[40] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. « Binaryconnect: Training deep neural networks with binary weights during propagations ». In: *Advances in neural information processing systems* 28 (2015).

[41] George Cybenko. « Approximation by superpositions of a sigmoidal function ». In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[42] Luke N Darlow, Elliot J Crowley, Antreas Antoniou, and Amos J Storkey. « Cinic-10 is not imagenet or cifar-10 ». In: *arXiv preprint arXiv:1810.03505* (2018).

[43] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. « Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point ». In: *Advances in neural information processing systems* 33 (2020), pp. 10271–10281.

[44] Ittai Dayan, Holger R Roth, Aoxiao Zhong, Ahmed Harouni, Amilcare Gentili, Anas Z Abidin, Andrew Liu, Anthony Beardsworth Costa, Bradford J Wood, Chien-Sung Tsai, et al. « Federated learning for predicting clinical outcomes in patients with COVID-19 ». In: *Nature medicine* 27.10 (2021), pp. 1735–1743.

[45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. « Imagenet: A large-scale hierarchical image database ». In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[46] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. « Exploiting linear structure within convolutional networks for efficient evaluation ». In: *Advances in neural information processing systems* 27 (2014).

[47] Enmao Diao, Jie Ding, and Vahid Tarokh. « Heterofl: Computation and communication efficient federated learning for heterogeneous clients ». In: *arXiv preprint arXiv:2010.01264* (2020).

[48] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. « A survey on ensemble learning ». In: *Frontiers of Computer Science* 14 (2020), pp. 241–258.

[49] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. « HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision ». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019.

[50] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. « Activation functions in deep learning: A comprehensive survey and benchmark ». In: *Neurocomputing* 503 (2022), pp. 92–108.

[51] Saso Džeroski and Bernard Ženko. « Is combining classifiers with stacking better than selecting the best one? » In: *Machine learning* 54 (2004), pp. 255–273.

[52] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. « LEARNED STEP SIZE QUANTIZATION ». In: *International Conference on Learning Representations*. 2019.

[53] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. « Local model poisoning attacks to {Byzantine-Robust} federated learning ». In: *29th USENIX security symposium (USENIX Security 20)*. 2020, pp. 1605–1622.

[54] Takashi Fukuda, Masayuki Suzuki, Gakuto Kurata, Samuel Thomas, Jia Cui, and Bhuvana Ramabhadran. « Efficient Knowledge Distillation from an Ensemble of Teachers. » In: *Interspeech*. 2017, pp. 3697–3701.

[55] Kunihiko Fukushima and Sei Miyake. « Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position ». In: *Pattern recognition* 15.6 (1982), pp. 455–469.

[56] Mudasir A Ganaie, Minghui Hu, Ashwani Kumar Malik, Muhammad Tanveer, and Ponnuthurai N Suganthan. « Ensemble deep learning: A review ». In: *Engineering Applications of Artificial Intelligence* 115 (2022), p. 105151.

[57] General Data Protection Regulation GDPR. « General data protection regulation ». In: *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC* (2016).

[58] Hossein Gholamalinezhad and Hossein Khosravi. « Pooling methods in deep neural networks, a review ». In: *arXiv preprint arXiv:2009.07485* (2020).

[59] Lucas Grativol, Lubin Gauthier, Mathieu Léonardon, Jérémy Morlier, Antoine Lavrard-Meyer, Guillaume Muller, Virginie Fresse, and Matthieu Arzel. « PEFSL: A deployment Pipeline for Embedded Few-Shot Learning on a FPGA SoC ». In: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2024, pp. 1–5. DOI: 10.1109/ISCAS58744.2024.10557995.

[60] Lucas Grativol, Mathieu Léonardon, Guillaume Muller, Virginie Fresse, and Matthieu Arzel. « Federated learning compression designed for lightweight communications ». In: *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2023, pp. 1–4.

[61] Lucas Grativol, Mathieu Leonardon, Guillaume Muller, Virginie Fresse, and Matthieu Arzel. « FLoCoRA: Federated Learning Compression with Low-Rank Adaptation ». In: *2024 32nd European Signal Processing Conference (EUSIPCO)*. IEEE. 2024, pp. 1786–1790.

[62] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. « LSTM: A search space odyssey ». In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[63] Samantha Guerriero, Barbara Caputo, and Thomas Mensink. « Deepncm: Deep nearest class mean classifiers ». In: (2018).

[64] Tom Gunter, Zirui Wang, Chong Wang, Ruoming Pang, Andy Narayanan, Aonan Zhang, Bowen Zhang, Chen Chen, Chung-Cheng Chiu, David Qiu, et al. « Apple intelligence foundation language models ». In: *arXiv preprint arXiv:2407.21075* (2024).

[65] Kartik Gupta, Marios Fournarakis, Matthias Reisser, Christos Louizos, and Markus Nagel. « Quantization robust federated learning for efficient inference on heterogeneous devices ». In: *arXiv preprint arXiv:2206.10844* (2022).

[66] Samyak Gupta, Yangsibo Huang, Zexuan Zhong, Tianyu Gao, Kai Li, and Danqi Chen. « Recovering private text in federated learning of language models ». In: *Advances in neural information processing systems* 35 (2022), pp. 8130–8143.

[67] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. « Deep learning with limited numerical precision ». In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746.

[68] Song Han, Huizi Mao, and William J Dally. « Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding ». In: *arXiv preprint arXiv:1510.00149* (2015).

[69] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. « Federated learning for mobile keyboard prediction ». In: *arXiv preprint arXiv:1811.03604* (2018).

[70] Babak Hassibi, David G Stork, and Gregory J Wolff. « Optimal brain surgeon and general network pruning ». In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 293–299.

[71] Benjamin Hawks, Javier Duarte, Nicholas J Fraser, Alessandro Pappalardo, Nhan Tran, and Yaman Umuroglu. « Ps and qs: Quantization-aware pruning for efficient low latency neural network inference ». In: *Frontiers in Artificial Intelligence* 4 (2021), p. 676564.

[72] Chaoyang He, Murali Annavaram, and Salman Avestimehr. « Towards non-IID and invisible data with FedNAS: Federated deep learning via neural architecture search ». In: *arXiv preprint arXiv:2004.08546* (2020).

[73] Chaoyang He, Haishan Ye, Li Shen, and Tong Zhang. « Milenas: Efficient neural architecture search via mixed-level reformulation ». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 11993–12002.

[74] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. « Towards a Unified View of Parameter-Efficient Transfer Learning ». In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=0RDcd5Axok.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. « Deep residual learning for image recognition ». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[76] Geoffrey Hinton. « Distilling the Knowledge in a Neural Network ». In: *arXiv preprint arXiv:1503.02531* (2015).

[77] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. « Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. » In: *J. Mach. Learn. Res.* 22.241 (2021), pp. 1–124.

[78] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. « Multilayer feedforward networks are universal approximators ». In: *Neural networks* 2.5 (1989), pp. 359–366.

[79] Mark Horowitz. « 1.1 computing's energy problem (and what we can do about it) ». In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

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

[81] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. « LoRA: Low-Rank Adaptation of Large Language Models ». In: *International Conference on Learning Representations*. 2021.

[82] Yangsibo Huang, Samyak Gupta, Zhao Song, Kai Li, and Sanjeev Arora. « Evaluating gradient inversion attacks and defenses in federated learning ». In: *Advances in neural information processing systems* 34 (2021), pp. 7232–7241.

[83] Minyoung Huh, Hossein Mobahi, Richard Zhang, Brian Cheung, Pulkit Agrawal, and Phillip Isola. « The Low-Rank Simplicity Bias in Deep Networks ». In: *Transactions on Machine Learning Research* (2022).

[84] Nam Hyeon-Woo, Moon Ye-Bin, and Tae-Hyun Oh. « FedPara: Low-rank Hadamard Product for Communication-Efficient Federated Learning ». In: *International Conference on Learning Representations*. 2021.

[85] « IEEE Standard for Floating-Point Arithmetic ». In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[86] Fatih Ilhan, Gong Su, and Ling Liu. « Scalefl: Resource-adaptive federated learning with heterogeneous clients ». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 24532–24541.

[87] Divyansh Jhunjhunwala, Shiqiang Wang, and Gauri Joshi. « Fedexp: Speeding up federated averaging via extrapolation ». In: *arXiv preprint arXiv:2301.09604* (2023).

[88] Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. « Model pruning enables efficient federated learning on edge devices ». In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[89] Peter Kairouz and et al. « Advances and open problems in federated learning ». In: *Foundations and Trends® in Machine Learning* 14.1–2 (2021), pp. 1–210.

[90] Dhiraj Kalamkar and et al. « A study of BFLOAT16 for deep learning training ». In: *arXiv:1905.12322* (2019).

[91] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. « Scaffold: Stochastic controlled averaging for federated learning ». In: *International conference on machine learning*. PMLR. 2020, pp. 5132–5143.

[92] Heekyung Kim and Kyuwon Ken Choi. « A Reconfigurable CNN-based Accelerator Design for Fast and Energy-Efficient Object Detection System on Mobile FPGA ». In: *IEEE Access* (2023).

[93] Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. « DeepStability: A study of unstable numerical methods and their solutions in deep learning ». In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 586–597.

[94] Jakub Konečnỳ, Brendan McMahan, and Daniel Ramage. « Federated optimization: Distributed optimization beyond the datacenter ». In: *arXiv preprint arXiv:1511.03575* (2015).

[95] Alex Krizhevsky, Geoffrey Hinton, et al. « Learning multiple layers of features from tiny images ». In: (2009).

[96] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. « Imagenet classification with deep convolutional neural networks ». In: *Advances in neural information processing systems* 25 (2012).

[97] Solomon Kullback and Richard A Leibler. « On information and sufficiency ». In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[98] Hsiang Tsung Kung and Charles E Leiserson. « Systolic arrays (for VLSI) ». In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA. 1979, pp. 256–282.

[99] Steinar Laenen and Luca Bertinetto. « On episodes, prototypical networks, and few-shot learning ». In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 24581–24592.

[100] Hugo Le Blevec, Mathieu Léonardon, Hugo Tessier, and Matthieu Arzel. « Pipelined Architecture for a Semantic Segmentation Neural Network on FPGA ». In: *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2023, pp. 1–4.

[101] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. « Deep learning ». In: *nature* 521.7553 (2015), pp. 436–444.

[102] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. « Privacy-preserving machine learning with fully homomorphic encryption for deep neural network ». In: *iEEE Access* 10 (2022), pp. 30039–30054.

[103] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. « Pruning filters for efficient convnets ». In: *arXiv preprint arXiv:1608.08710* (2016).

[104] Peichun Li, Guoliang Cheng, Xumin Huang, Jiawen Kang, Rong Yu, Yuan Wu, and Miao Pan. « AnycostFL: Efficient On-Demand Federated Learning over Heterogeneous Edge Devices ». In: *arXiv preprint arXiv:2301.03062* (2023).

[105] Qinbin Li, Bingsheng He, and Dawn Song. « Model-contrastive federated learning ». In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 10713–10722.

[106] Tian Li, Shengyuan Hu, Ahmad Beirami, and Virginia Smith. « Ditto: Fair and robust federated learning through personalization ». In: *International conference on machine learning*. PMLR. 2021, pp. 6357–6368.

[107] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. « Federated optimization in heterogeneous networks ». In: *Proceedings of Machine learning and systems* 2 (2020), pp. 429–450.

[108] Yang Li and Jiachen Yang. « Few-shot cotton pest recognition and terminal realization ». In: *Computers and Electronics in Agriculture* 169 (2020), p. 105240.

[109] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. « Brecq: Pushing the limit of post-training quantization by block reconstruction ». In: *arXiv preprint arXiv:2102.05426* (2021).

[110] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. « On-Device Training Under 256KB Memory ». In: *arXiv:2206.15472* (2022).

[111] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. « Ensemble distillation for robust model fusion in federated learning ». In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 2351–2363.

[112] Yanbin Liu, Juho Lee, Minseop Park, Saehoon Kim, Eunho Yang, Sung Ju Hwang, and Yi Yang. « Learning to propagate labels: Transductive propagation network for few-shot learning ». In: *arXiv preprint arXiv:1805.10002* (2018).

[113] Dengsheng Lu and Qihao Weng. « A survey of image classification methods and techniques for improving classification performance ». In: *International journal of Remote sensing* 28.5 (2007), pp. 823–870.

[114] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. « An efficient hardware accelerator for sparse convolutional neural networks on FPGAs ». In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 17–25.

[115] Srikanth Machiraju. *How to train your deep learning models in a distributed fashion*. `https://towardsdatascience.com/how-to-train-your-deep-learning-models-in-a-distributed-fashion-43a6f53f0484`. Accessed: 02/10/2024. 2021.

[116] Puneet Mangla, Nupur Kumari, Abhishek Sinha, Mayank Singh, Balaji Krishnamurthy, and Vineeth N Balasubramanian. « Charting the right manifold: Manifold mixup for few-shot learning ». In: *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 2020, pp. 2218–2227.

[117] Brendan McMahan and et al. « Communication-efficient learning of deep networks from decentralized data ». In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[118] Umberto Michieli and Mete Ozay. « Are all users treated fairly in federated learning systems? » In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 2318–2322.

[119] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. « Fp8 formats for deep learning ». In: *arXiv preprint arXiv:2209.05433* (2022).

[120] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*. Vol. 754. IEEE, 1985.

[121] Ivan Miro-Panades, Inna Kucher, Vincent Lorrain, and Alexandre Valentian. « Meeting the Latency and Energy Constraints on Timing-critical Edge-AI Systems ». In: *Embedded Artificial Intelligence*. River Publishers, 2023, pp. 61–67.

[122] Alistair Moffat. « Huffman coding ». In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–35.

[123] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Vol. 1. Springer, 2018.

[124] Saurav Muralidharan. « Uniform sparsity in deep neural networks ». In: *Proceedings of Machine Learning and Systems* 5 (2023).

[125] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. « Up or down? adaptive rounding for post-training quantization ». In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7197–7206.

[126] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. « A white paper on neural network quantization ». In: *arXiv preprint arXiv:2106.08295* (2021).

[127] Markus Nagel, Marios Fournarakis, Yelysei Bondarenko, and Tijmen Blankevoort. « Overcoming oscillations in quantization-aware training ». In: *International Conference on Machine Learning*. PMLR. 2022, pp. 16318–16330.

[128] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. « What is being transferred in transfer learning? » In: *Advances in neural information processing systems* 33 (2020), pp. 512–523.

[129] John Nguyen, Jianyu Wang, Kshitiz Malik, Maziar Sanjabi, and Michael Rabbat. « Where to begin? on the impact of pre-training and initialization in federated learning ». In: *arXiv preprint arXiv:2206.15387* (2022).

[130] Mohd Halim Mohd Noor and Ayokunle Olalekan Ige. « A Survey on Deep Learning and State-of-the-arts Applications ». In: *arXiv preprint arXiv:2403.17561* (2024).

[131] Martijn Oldenhof, Gergely Ács, Balázs Pejó, Ansgar Schuffenhauer, Nicholas Holway, Noé Sturm, Arne Dieckmann, Oliver Fortmeier, Eric Boniface, Clément Mayer, et al. « Industry-scale orchestrated federated learning for drug discovery ». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 13. 2023, pp. 15576–15584.

[132] Alessandro Pappalardo. *Xilinx/brevitas*. 2021. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

[133] Jupinder Parmar, Shrimai Prabhumoye, Joseph Jennings, Bo Liu, Aastha Jhunjhunwala, Zhilin Wang, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. « Data, Data Everywhere: A Guide for Pretraining Dataset Construction ». In: *arXiv preprint arXiv:2407.06380* (2024).

[134] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. « PyTorch: An Imperative Style, High-Performance Deep Learning Library ». In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[135] Kilian Pfeiffer, Ramin Khalili, and Jörg Henkel. « Aggregating capacity in FL through successive layer training for computationally-constrained devices ». In: *Advances in Neural Information Processing Systems* 36 (2024).

[136] Xinchi Qiu, Javier Fernandez-Marques, Pedro PB Gusmao, Yan Gao, Titouan Parcollet, and Nicholas Donald Lane. « Zerofl: Efficient on-device training for federated learning with local sparsity ». In: *arXiv preprint arXiv:2208.02507* (2022).

[137] Xinchi Qiu, Titouan Parcollet, Javier Fernandez-Marques, Pedro PB Gusmao, Yan Gao, Daniel J Beutel, Taner Topal, Akhil Mathur, and Nicholas D Lane. « A first look into the carbon footprint of federated learning ». In: *Journal of Machine Learning Research* 24.129 (2023), pp. 1–23.

[138] Md Aamir Raihan and Tor Aamodt. « Sparse weight activation training ». In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 15625–15638.

[139] Waseem Rawat and Zenghui Wang. « Deep convolutional neural networks for image classification: A comprehensive review ». In: *Neural computation* 29.9 (2017), pp. 2352–2449.

[140] Sashank Reddi and et al. « Adaptive federated optimization ». In: *arXiv:2003.00295* (2020).

[141] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. « Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization ». In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2020, pp. 2021–2031.

[142] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. « U-net: Convolutional networks for biomedical image segmentation ». In: *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer. 2015, pp. 234–241.

[143] Frank Rosenblatt. « The perceptron: a probabilistic model for information storage and organization in the brain. » In: *Psychological review* 65.6 (1958), p. 386.

[144] Sebastian Ruder. « An overview of gradient descent optimization algorithms ». In: *arXiv preprint arXiv:1609.04747* (2016).

[145] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. « Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986 ». In: *Biometrika* 71.599-607 (1986), p. 6.

[146] Tyler Scott, Karl Ridgeway, and Michael C Mozer. « Adapted deep embeddings: A synthesis of methods for k-shot inductive transfer learning ». In: *Advances in Neural Information Processing Systems* 31 (2018).

[147] Micah J Sheller, G Anthony Reina, Brandon Edwards, Jason Martin, and Spyridon Bakas. « Multi-institutional deep learning modeling without sharing patient data: A feasibility study on brain tumor segmentation ». In: *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries: 4th International Workshop, BrainLes 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 16, 2018, Revised Selected Papers, Part I 4*. Springer. 2019, pp. 92–104.

[148] Yujun Shi, Jian Liang, Wenqing Zhang, Vincent YF Tan, and Song Bai. « Towards understanding and mitigating dimensional collapse in heterogeneous federated learning ». In: *arXiv preprint arXiv:2210.00226* (2022).

[149] Karen Simonyan. « Very deep convolutional networks for large-scale image recognition ». In: *arXiv preprint arXiv:1409.1556* (2014).

[150] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. « Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication ». In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2022, pp. 65–77.

[151] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. « Dropout: a simple way to prevent neural networks from overfitting ». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[152] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. « Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks ». In: *Advances in neural information processing systems* 32 (2019).

[153] Alysa Ziying Tan, Han Yu, Lizhen Cui, and Qiang Yang. « Towards personalized federated learning ». In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[154] Yue Tan, Guodong Long, Jie Ma, Lu Liu, Tianyi Zhou, and Jing Jiang. « Federated learning from pre-trained models: A contrastive learning approach ». In: *Advances in neural information processing systems* 35 (2022), pp. 19332–19344.

[155] Zhenheng Tang, Yonggang Zhang, Shaohuai Shi, Xinmei Tian, Tongliang Liu, Bo Han, and Xiaowen Chu. « FedImpro: Measuring and Improving Client Update in Federated Learning ». In: *The Twelfth International Conference on Learning Representations*. 2024.

[156] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. « Branchynet: Fast inference via early exiting from deep neural networks ». In: *2016 23rd international conference on pattern recognition (ICPR)*. IEEE. 2016, pp. 2464–2469.

[157] Tensil. *Tensil*. `https://github.com/tensil-ai/tensil`. 2022.

[158] Zhen Ling Teo, Liyuan Jin, Siqi Li, Di Miao, Xiaoman Zhang, Wei Yan Ng, Ting Fang Tan, Deborah Meixuan Lee, Kai Jie Chua, John Heng, et al. « Federated machine learning in healthcare: A systematic review on clinical applications and technical architecture ». In: *Cell Reports Medicine* (2024).

[159] Hugo Tessier. « Convolutional neural networks pruning and its application to embedded vision systems ». PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique, 2023.

[160] Hugo Tessier, Vincent Gripon, Mathieu Léonardon, Matthieu Arzel, David Bertrand, and Thomas Hannagan. « Leveraging structured pruning of convolutional neural networks ». In: *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2022, pp. 1–6.

[161] Hugo Tessier, Vincent Gripon, Mathieu Léonardon, Matthieu Arzel, Thomas Hannagan, and David Bertrand. « Rethinking Weight Decay for Efficient Neural Network Pruning ». In: *Journal of Imaging* 8.3 (2022), p. 64.

[162] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. « Data poisoning attacks against federated learning systems ». In: *Computer security–ESORICs 2020: 25th European symposium on research in computer security, ESORICs 2020, guildford, UK, September 14–18, 2020, proceedings, part i 25*. Springer. 2020, pp. 480–501.

[163] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. « Split learning for health: Distributed deep learning without sharing raw patient data ». In: *arXiv preprint arXiv:1812.00564* (2018).

[164] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. « A survey on distributed machine learning ». In: *Acm computing surveys (csur)* 53.2 (2020), pp. 1–33.

[165] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. « Machine learning model sizes and the parameter gap ». In: *arXiv preprint arXiv:2207.02852* (2022).

[166] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. « Matching networks for one shot learning ». In: *Advances in neural information processing systems* 29 (2016).

[167] Song Wang, Xingbo Fu, Kaize Ding, Chen Chen, Huiyuan Chen, and Jundong Li. « Federated few-shot learning ». In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 2374–2385.

[168] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. « Structured pruning of large language models ». In: *arXiv preprint arXiv:1910.04732* (2019).

[169] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. « Federated learning with differential privacy: Algorithms and performance analysis ». In: *IEEE transactions on information forensics and security* 15 (2020), pp. 3454–3469.

[170] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. « Integer quantization for deep learning inference: Principles and empirical evaluation ». In: *arXiv preprint arXiv:2004.09602* (2020).

[171] Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. « Sheared llama: Accelerating language model pre-training via structured pruning ». In: *arXiv preprint arXiv:2310.06694* (2023).

[172] Li Yang, Zhezhi He, and Deliang Fan. « A fully onchip binarized convolutional neural network fpga impelmentation with accurate inference ». In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2018, pp. 1–6.

[173] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. « Vertical federated learning ». In: *Federated Learning*. Springer, 2020, pp. 69–81.

[174] Zhaoxiong Yang, Shuihai Hu, and Kai Chen. « FPGA-based hardware accelerator of homomorphic encryption for efficient federated learning ». In: *arXiv preprint arXiv:2007.10560* (2020).

[175] Jaehong Yoon, Geon Park, Wonyong Jeong, and Sung Ju Hwang. « Bitwidth heterogeneous federated learning with progressive weight dequantization ». In: *International Conference on Machine Learning*. PMLR. 2022, pp. 25552–25565.

[176] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. « How transferable are features in deep neural networks? » In: *Advances in neural information processing systems* 27 (2014).

[177] Kaichao You, Yong Liu, Jianmin Wang, and Mingsheng Long. « Logme: Practical assessment of pre-trained models for transfer learning ». In: *International Conference on Machine Learning*. PMLR. 2021, pp. 12133–12143.

[178] Weijie You and Chang Wu. « RSNN: A software/hardware co-optimized framework for sparse convolutional neural networks on FPGAs ». In: *IEEE Access* 9 (2020), pp. 949–960.

[179] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. « {BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning ». In: *2020 USENIX annual technical conference (USENIX ATC 20)*. 2020, pp. 493–506.

[180] Hao Zhang, Jiongrui He, and Seok-Bum Ko. « Efficient posit multiply-accumulate unit generator for deep learning applications ». In: *2019 IEEE international symposium on circuits and systems (ISCAS)*. IEEE. 2019, pp. 1–5.

[181] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223.

[182] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. « A comprehensive survey on transfer learning ». In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.

[183] B Zoph. « Neural Architecture Search with Reinforcement Learning ». In: *arXiv preprint arXiv:1611.01578* (2016).

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Compression des Réseaux de Neurones dans le Contexte de l'Apprentissage Fédéré et des Systèmes Embarqués

Mot clés : Apprentissage Fédéré, Élagage, Adaptation de Basse Rang, Apprentissage avec Peu d'Exemples, FPGA

Résumé : L'apprentissage fédéré est un cadre d'apprentissage automatique collaboratif et décentralisé, motivé par des préoccupations croissantes concernant la confidentialité des données. En transférant l'entraînement des modèles vers des nœuds locaux et en conservant les données sur place, il favorise une approche plus respectueuse de la vie privée. Toutefois, cette méthode impose un surcoût en termes de communication et de calcul à ceux qui l'adoptent. Dans ce manuscrit, nous examinons les principaux défis de l'apprentissage fédéré et proposons des solutions visant à augumenter l'efficacité tout en réduisant les besoins en ressources matérielles. Plus précisément, nous explorons des techniques de compression classiques, comme l'élagage, ainsi que des approximations en rang faible afin de diminuer les coûts associés à l'apprentissage fédéré. Pour les scénarios où les participants disposent de capacités de communication limitées, nous introduisons une méthodologie de co-conception pour un algorithme d'apprentissage avec peu d'examples embarqué. Notre solution intègre les contraintes matérielles au sein d'un pipeline de déploiement sur des plateformes FPGA, aboutissant à un algorithme à faible latence qui peut également être exploité pour mettre en œuvre des modèles post-apprentissage fédéré.

Title: Neural Network Compression in the Context of Federated Learning and Edge Devices

Keywords: Federated Learning, Pruning, Low-Rank Adaptation, Few-Shot Learning, FPGA

Abstract: Federated learning is a collaborative, decentralized machine learning framework driven by growing concerns about data privacy. By shifting model training to local nodes and keeping data local, it enables more privacy-conscious training. However, this approach imposes additional communication and computation overhead on those who adopt it. In this manuscript, we examine the key challenges in federated learning and propose solutions to increase efficiency and reduce hardware requirements. Specifically, we explore classic compression techniques, such as pruning, and low-rank approximations to lower the costs associated with federated learning. For scenarios where participants have limited communication capabilities, we introduce a co-design methodology for an embedded few-shot learning algorithm. Our proposed solution integrates hardware constraints into a deployment pipeline for FPGA platforms, resulting in a low-latency algorithm that can also be leveraged to implement post-federated learning models.