COLLEGE DOCTORAL BRETAGNE — MATHS, TELECOMS, INFORMATIQUE, SIGNAL, SYSTEMES, ELECTRONIQUE

INSA — INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

ÉCOLE DOCTORALE Nº 601
Mathématiques, télécommunications, informatique, signal, systémes, électronique
Spécialité : *Informatique*

Par
Thomas BOUVIER

Distributed Rehearsal Buffers for Continual Learning at Scale

Thèse présentée et soutenue à Rennes, le 04 Novembre 2024
Unité de recherche : Inria

Rapporteurs avant soutenance :

Bruno RAFFIN Directeur de recherche, HDR, Inria, Grenoble, France
Eddy CARON Professeur des Universités, Université Claude Bernard Lyon 1 (ISFA), France

Composition du Jury :

Examinateurs :	Ilkay ALTINTAS	Research Scientist, University of California San Diego, United States
	Cédric TEDESCHI	Professeur des Universités, Université de Rennes, France
Dir. de thèse :	Alexandru COSTAN	Maître de conférences, HDR, INSA, Rennes, France
Co-dir. de thèse :	Gabriel ANTONIU	Directeur de recherche, HDR, Inria, Rennes, France

À William,
Georges,
Odette “Chicago”,
à toutes celles et ceux que j’aime,
à Marie-Joe.

We wait for something.
We hope, we lose hope, we move closer to death... finally we die.

REMERCIEMENTS

Ces quatres dernières années de doctorat m'ont permis de grandir en tant qu'ingénieur, chercheur, et personne. Je veux ici remercier toutes les personnes qui m'ont accompagné dans la réalisation de ce projet. Cela n'a pas été de tout repos : bien sûr, la pandémie a causé de l'isolement et de l'incertitude pour toutes et tous, et un ralentissement des collaborations internationales. Mais aussi, conflits inter-personnels et décès de proches ont été d'autant plus d'épreuves à concilier avec la vie professionnelle.

D'abord, je veux remercier les membres du jury d'avoir accepté de revoir mon travail : Bruno Raffin et Eddy Caron pour les retours constructifs, Ilkay Altintas et Cédric Tedeschi. Merci également aux membres de mon comité de suivi, Christian Perez et Adrien Lebre.

Ensuite, je veux exprimer ma profonde reconnaissance pour mes deux directeurs de thèse, Alex et Gabriel, pour la confiance que vous m'avez accordé au début de cette aventure. Un chaleureux merci pour vos conseils, encouragements et opportunités. J'ai beaucoup apprécié votre bienveillance et attention portée à l'épanouissement personnel et professionnel de vos étudiants. Je suis heureux d'avoir pu mener ce projet dans d'excellentes conditions.

Un grand merci également à Bogdan Nicolae pour ton mentorat technique, l'opportunité de stage à ANL, et la bonne humeur lors de nos réunions parfois un peu trop longues.

Merci à tous les membres de l'équipe KerData, de laquelle je garderai d'excellents souvenirs. Je vous cite tous : Daniel pour les années passées au bureau, pour ton support, notre virée à Chicago, et les dizaines de sorties de course à pied le long du halage. Cédric et Julien pour nos innombrables repas à la Biocoop, Josh avec qui je garde d'excellents souvenirs de Saint-Louis, Hugo et Malvin, avec qui j'ai beaucoup apprécié travailler, François pour ton écoute et les astuces en tous genres, Robin (j'ignorais tes talents de graphiste !), Mathis, Guillaume pour tes réflexions et initiatives, Thomas et Théo, la relève de D174, Silvina, Jakob pour vos conseils, Luis, et bien sûr Laurence pour ton support. Merci à Luc pour tes conseils remplis de sagesse.

Merci également aux membres du groupe Sciences, Environnement et Société de Rennes qui m'ont permis de développer ma réflexion sur ces sujets. Je suis satisfait d'avoir pu déployer des Fresques du Climat à l'Inria.

J'ai le privilège d'être vraiment très bien entouré. Thib, Simon, merci d'être là depuis toujours, dans les bons moments et quand ça va pas aussi. J'ai une pensée pour Ju, qui m'encourageait à l'époque à me lancer dans ce projet, puis pour Aude, Romane, Roman, Alexis et Timot (les anciens de CS1) qui m'ont aidé à me projeter dedans. Vous m'avez donné l'impulsion qui m'a permis de dépasser l'auto-censure et de me lancer dans l'inconnu. Un grand merci à toute la

bande de Rennes, Quentin, Nolwenn, Mélanie, Clémence, Félix, Pierre, Lise et Émilie pour votre écoute : vous m'avez pour certain.es accompagné dans les moments de doute. Timot, une nouvelle fois, tes encouragements et conseils m'ont été précieux pour mener ce travail à bien (merci pour l'ordinateur aussi !). Je n'oublie évidemment pas Pierre et Thib pour votre accueil toujours chaleureux dans la coloc de Lyon, et les dizaines de cafés bus sur votre canapé (lit). Merci à Vincent et Claire pour les week-ends de repos à Paris, à Nolwenn et Yann pour les vacances bien méritées, à la Team P, Guigui, Sim, Alexandre et Alexandre, Aurel, Aude, Coco et Cam. À mes chers voisins, Benjamin, Chloë, Pierre, Anaëlle, à Julia pour les mises en perspective. À la bande d'Angers, à Léa pour les conseils, longues discussions et encouragements. À May et Pati pour votre support.

Je garde d'excellents souvenirs des quelques mois passés à Chicago en compagnie de Lucie, Amandine, François, Antoine et Sean, que je suis content d'avoir toutes et tous recroisés en France depuis. On se revoit bientôt !

On n'a pas souvent l'occasion de le faire : je profite de cet espace pour mentionner les profs qui ont marqué ma scolarité. J'ai donc une pensée pour M. Hébert, qui avait su me donner confiance dès le collège, à Mme Sauvé et M. Gachon pour l'apprentissage de la rigueur et la transmission de leur passion pour les sciences au lycée, et à M. Merlet pour sa suggestion (excellente !) de postuler à l'INSA après le bac. De là-bas, je pense à Muriel Pressigout du département EII qui a perçu ma passion pour l'informatique et m'a fait découvrir le monde de la recherche sous sa supervision. Il y a aussi ceux qui m'ont accompagné lors de mes premières expériences professionnelles : Jean-Noël Grimault et Christophe Hardouin pour leur confiance donnée sur mes premiers projets informatiques hors cadre personnel, puis Bertrand Prillard et Corentin Le Huby pour la montée en compétences techniques sur de lourds projets industriels. Enfin, merci à Michael Fester et Valentin Fage, qui auront accompagné ma transition entre le monde des startups et celui de la recherche.

Merci à toutes les personnes m'ayant apporté une assistance technique à un moment ou un autre : Matthieu Simonin de l'Inria, Phil Carns, Matthieu Dorier, et Robert Underwood d'ANL, ainsi qu'à mes co-auteurs Ian et Tekin.

Merci au cuisinier de la Biocoop, quel plaisir de manger là-bas !

Merci enfin à papa, à maman, à Alexis, qui se demandent depuis toujours ce que je peux bien faire sur cet ordinateur pendant des jours et des nuits. J'espère que la lecture de cette thèse vous éclairera :) Je suis reconnaissant pour la confiance que vous m'avez toujours accordé dans chacun de mes choix, de votre support sans faille, et de votre présence à mes côtés. Un chaleureux merci à tata Christelle, mamie, Philippe, Ana, Jean-Luc et Claude pour vos mots d'encouragement durant ces dernières années.

TABLE OF CONTENTS

0.1 Contexte 1

1 Introduction **14**

1.1 Context 14
1.2 Research Objectives 16
1.3 Contributions 18
1.4 Publications 19
1.5 Software 20
1.5.1 Main Contributions 20
1.5.2 Contribution to Existing Software (via Pull Requests) 21
1.6 Organization of this Dissertation 23

2 Background **25**

2.1 Towards Integrated HPC/ML Ecosystems 26
2.1.1 Definitions and Landscape 26
2.1.2 Patterns for Integration of Instruments and Computing 28
2.2 Continual Learning 29
2.2.1 Supervised Learning 29
2.2.2 Deep Neural Networks 30
2.2.3 Catastrophic Forgetting 37
2.2.4 Addressing Catastrophic Forgetting 39
2.3 Parallel and Distributed Deep Learning 41
2.3.1 Single/Multi Machine Parallelism 41
2.3.2 Parallel Algorithms for Deep Learning 42
2.4 Challenges Considered in this Work 47
2.4.1 Mitigating Catastrophic Forgetting through Continual Learning at Scale . 47
2.4.2 The Efficiency Tradeoff: Generalization Gap vs. Compute Efficiency 48

3 Distributed Rehearsal Buffers: a Flexible Continual Learning Abstraction **51**

3.1 Distributed Rehearsal Buffers and Data Parallelism 52
3.1.1 The choice for Data-parallel Experience Replay 52
3.1.2 Aggregated Memory Space of Rehearsal Buffers 53
3.1.3 Selection and Eviction Policies 55

3.1.4 Sampling Strategies for Continual Learning . . . 56
3.2 Transparent Global Sampling of Representatives . . . 57
3.2.1 The Need for Global Sampling of Representatives . . . 57
3.2.2 Efficient Augmentation of Minibatches . . . 60
3.2.3 Asynchronous Management of Rehearsal Buffers . . . 62

4 *Neomem*: an Efficient Implementation of Rehearsal-based Continual Learning 65
4.1 Architectural Overview . . . 66
4.1.1 Storing and Serving Data Samples . . . 66
4.1.2 Integration with the Training Procedure . . . 67
4.2 Asynchronous Techniques for Efficient Buffer Management . . . 69
4.2.1 Multi-threaded Concurrency . . . 70
4.2.2 Non-blocking RPCs . . . 71
4.2.3 Asynchronous CUDA Copies . . . 72
4.3 I/O Optimizations in Data Movements . . . 73
4.3.1 Transferring Data using RDMA-enabled RPCs . . . 74
4.3.2 RPC Consolidation . . . 75
4.3.3 Leveraging Page-locked Buffers for Efficient Data Transfers . . . 75

5 Experimental Evaluation using a Synthetic Benchmark 77
5.1 Experimental Setup and Methodology . . . 78
5.1.1 Training Dataset & Continual Learning Scenario . . . 78
5.1.2 Learning Models . . . 79
5.1.3 Training Procedure at Scale . . . 80
5.1.4 Performance Metrics . . . 81
5.1.5 Computing Environment . . . 82
5.2 Experiments and Results . . . 82
5.2.1 Impact of the Rehearsal Buffer Size on Accuracy . . . 83
5.2.2 Impact of Other Rehearsal-related Hyperparameters on Accuracy . . . 84
5.2.3 Comparison with Baseline Approaches . . . 84
5.2.4 Rehearsal Buffer Management Breakdown . . . 85
5.2.5 Scalability Study . . . 87
5.3 Discussion . . . 89
5.3.1 Experience Replay is a Strong Approach . . . 89
5.3.2 The Rehearsal Buffer Size Matters . . . 89
5.3.3 Rehearsal as a Trade-off Between Incremental and From-scratch Training . 90
5.3.4 Choice of the Performance Metrics . . . 90
5.3.5 The Need for Hyperparameter Optimization . . . 91

6 Towards a Generic Distributed Rehearsal Buffer **93**
6.1 Decoupling Buffer Management from the Learning Task 93
6.1.1 Supporting More Rehearsal Strategies and Learning Tasks 94
6.1.2 Extending Rehearsal with Additional States 94
6.2 Illustration Leveraging Knowledge Distillation 95
6.2.1 Integrating Dark Experience Replay 95
6.2.2 Integrating Dark Experience Replay++ 97

7 Application to a Real-Life Use-case: the Ptychography Application **101**
7.1 Motivating Scenario: Continual Learning in Support of Streaming Applications . 102
7.1.1 The Need for *ML out HPC* 102
7.1.2 Ptychographic Image Reconstruction 103
7.1.3 DL-enabled Real-Time Ptychographic Reconstruction 103
7.1.4 Generative Models and Rehearsal-based Continual Learning 106
7.2 Experimental Setup and Methodology 107
7.2.1 Training Dataset 108
7.2.2 Continual Learning Scenario 109
7.2.3 Learning Model 110
7.2.4 Performance Metrics 111
7.2.5 Computing Environment 112
7.3 Experiments and Results 112
7.3.1 Comparison with Baseline Approaches 112
7.3.2 Comparison of Tike vs. PtychoNN Stitched Reconstructions 114

8 Conclusion and Prospects **117**
8.1 Achievements 118
8.1.1 Achieving Much-Improved Accuracy in Class-Incremental CL Scenarios . 118
8.1.2 Enabling the Support of More Advanced CL Algorithms 119
8.1.3 Enabling the Steering of Experiments in Real-Time, by Learning from a Stream of Data 119
8.2 Prospects 120
8.2.1 More Advanced Selection and Eviction Policies 120
8.2.2 Application to Numerical Simulations 121
8.2.3 Application to Large Language Models 122

Bibliography **125**

List of Figures

2.1 The single operator, also called *fully connected layer*, is represented in yellow. The input layer is represented in blue. Figure borrowed from [8]. 31

2.2 A DNN is composed of many operators. Figure borrowed from [8]. 36

2.3 The split-MNIST dataset according to each of the three continual learning scenarios. With Task-IL, the choice for the model is between two digits from a given task (e.g., zero or one). With Domain-IL, the choice is still between two options (e.g., even or odd), but the problem is more difficult as task identity is not provided at inference time. With Class-IL, the model must choose between all ten digits, and the output space is growing as new tasks are observed. Borrowed from [67], where it was modified from [68]. 38

2.4 Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one training process is active at a time. Bottom: GPipe [113] divides the input minibatch into smaller microbatches, enabling different processes to work with separate microbatches in parallel. Figure borrowed from [113]. 45

3.1 For every process n, a rehearsal buffer $\mathcal{B}_n$ contains representatives from the classes seen so far. The distributed rehearsal buffer $\mathcal{B}$ contains representatives from the K classes. 54

3.2 For a given process n, c candidates from the incoming minibatch are sampled and used to populate $\mathcal{B}_n$. If the buffer for class i is full, representatives from R_n^i are replaced at random. The figure depicts the rehearsal buffer $\mathcal{B}_n$ state for two subsequent iterations for $c = 2$. 56

3.3 On a given process n, every incoming minibatch is augmented with r representatives sampled randomly and without replacement from the distributed rehearsal buffer $\mathcal{B}$. Here, $r = 2$ on two subsequent iterations. Sampling from $\mathcal{B}$ introduces communication between the p distributed processes. 61

3.4 Asynchronous updates of the rehearsal buffers and global augmentations: r representatives sampled globally beginning with the previous iteration are used by the training loop to assemble an augmented minibatch on each process n. Meanwhile, the distributed rehearsal buffer extracts candidates from the current minibatch to update each B_n locally, then collects the next r representatives using global sampling. 63

4.1 Gradients are exchanged using MPI `allreduce` collectives. Representatives are exchanged between remote processes using RPC requests. 67

4.2 Margo is a library built on top of Mercury and Argobots, running a Mercury progress loop in a ULT and converting RPC handlers into ULTs. 70

4.3 The RPC system implemented in Mercury has a focus on high concurrency and support explicit bulk transfers. 74

5.1 Accuracy w.r.t. different rehearsal buffer sizes $|\mathcal{B}|$ (percentage of the input dataset). Each data point is the average of the top-5 accuracy obtained on all previous tasks. 83

5.2 $|\mathcal{B}| = 30\%$ and $r = 7$. Left: accuracy w.r.t. epoch number. Our rehearsal-based approach achieves a final accuracy of 80.55%. Right: training time w.r.t. epoch number. Our approach induces a small runtime increase compared with incremental training, which stays linear. 85

5.3 Top-5 accuracy for ResNet-50, 16 GPUs, ImageNet (4 tasks). 85

5.4 Time breakdown for the training loop and rehearsal buffer management, for each of the three models and for different number of GPUs, averaged across 35 minibatches. Training iterations sometimes take longer as the time required for communication and synchronization between GPUs increases with their number. Nonlinearity can be explained by the topology of GPU interconnects, as well as other factors such as communication overhead and resource contention. 86

5.5 Accuracy and runtime, $|\mathcal{B}| = 30\%$, $b = 56$ and $r = 7$ for all 3 models. For ResNet-50, colors match those in Fig. 5.2. 87

7.1 The first row shows 5 different diffraction patterns obtained from the same specimen perspective. For each of these 5 inputs, the following lines show in order: the structure ("amplitude") computed by Tike [158] (conventional iterative algorithm), the amplitude predicted by PtychoNN, the difference between the amplitude computed by Tike and that predicted by PtychoNN, the phase computed by Tike, the phase predicted by PtychoNN, the difference between the phase computed by Tike and that predicted by PtychoNN. 104

7.2 Illustration of DL-enabled workflow for real-time streaming ptychography imaging, featuring the high-intensity beam produced by the light source, the specimen ("sample") under analysis, the HPC cluster performing the continual training of the DNN, and the edge device for live inference. Figure borrowed from [21]. . . . 105
7.3 With generative workloads, representatives of a single class are stored in the rehearsal buffer $\mathcal{B}_n$. This includes training samples as well as associated structure and phase reconstructions. Our generic design allows to hold additional information associated to these representatives: this example holds activations. This data is served using global sampling to be made available to the training procedure alongside representatives. 107
7.4 Illustration of the spatial path (left-right, top-bottom) taken by the X-ray beam to scan a single perspective of a specimen. This procedure generates 950 diffraction patterns in the far field, which are then used to compute structure and phase reconstructions using Tike. 109
7.5 PtychoNN architecture 110
7.6 Evolution of the validation loss for an increasing number of tasks. 113
7.7 Stitched reconstructions obtained from four different approaches. No task gets revisited during training (one epoch per task). From left to right: incremental training, vanilla Experience Replay, Experience Replay + knowledge distillation (DER++), and the ground-truth reconstruction computed by Tike. One can notice a gradual improvement in reconstruction quality. 114

LIST OF TABLES

2.1 Supervised Learning notation 30

3.1 Continual Learning notation 53

4.1 Neomem public API 68
4.2 Description of the Neomem functions used to issue RPCs 72
4.3 Description of the Neomem functions taking advantage of the CUDA API 73

7.1 PSRN and SSIM of incremental, ER and DER++ relative to *Tike* which is used as the baseline. 115

RÉSUMÉ EN FRANÇAIS

0.1 Contexte

Les modèles d'apprentissage profond (*Deep Learning*, DL) ont rapidement gagné en popularité à la fois dans l'industrie et dans de nombreux domaines liés au calcul scientifique, comme par exemple la reconnaissance vocale et la vision par ordinateur [1], la climatologie [2], la recherche sur la fusion nucléaire [3], la recherche sur le cancer [4], la médecine personnalisée [5] ou l'épidémiologie [6]. À mesure que les volumes de données et leur complexité augmentent, les modèles d'apprentissage profond ont évolué sous tous les aspects : en taille (nombre de paramètres), en profondeur (nombre de couches de neurones) et architecturalement. Malgré la convergence grandissante entre apprentissage automatique et calcul haute performance (*High-Performance Computing*, HPC) [7], qui a conduit à l'adoption de diverses techniques de parallélisation [8] (parallélisme de données, de modèle, ou hybride), l'entraînement des modèles reste une tâche très gourmande en ressources. En particulier, la puissance de calcul requise pour l'entraînement de modèles d'apprentissage automatique double environ tous les 6 mois depuis 2010 [9].

Les modèles d'apprentissage automatique sont généralement entraînés sur de grands systèmes multi-GPU qui ont accès au jeu de données d'entrée dès le début de la procédure (par exemple, via un système de fichiers parallèle). Une technique d'optimisation itérative est utilisée (par exemple, l'algorithme du gradient stochastique) pour revisiter les données d'entraînement plusieurs fois jusqu'à convergence. Cependant, de plus en plus d'applications nécessitent d'être entraînées avec des ensembles de données illimités, fréquemment mis à jour. Par exemple, les applications scientifiques acquérant des données expérimentales via des capteurs doivent rapidement les analyser afin d'ajuster une expérience en cours (par exemple, pour déclencher une décision automatisée). Dans ce cas de figure, réentraîner le modèle de zéro à chaque fois que de nouveaux échantillons sont rendus disponibles n'est pas envisageable : à mesure que les données d'entraînement s'accumulent, cette procédure prendrait de plus en plus de temps et consommerait toujours plus de ressources (de calcul et de stockage), et donnant lieu à des temps d'exécution prohibitifs. D'autres cas d'utilisation comme les jumeaux numériques [10, 11] nécessitent généralement une infrastructure d'exécution hybride : les dispositifs périphériques (*edge devices*) créent des flux de données d'entrée, qui sont traités par des applications d'analyse de données et d'apprentissage automatique dans le Cloud, et des simulations sur de grands systèmes HPC spécialisés fournissent des prédictions sur l'état futur du système. Ces applica-

tions reposent sur des interactions dynamiques qui nécessitent de prendre en compte les données acquises en temps réel. Ce paradigme ouvre de nouvelles perspectives pour le pilotage en temps réel des calculs, la reconfiguration dynamique des flux de travaux (*workflows*) ou le recalibrage des paramètres en cours d'exécution. Cette approche d'apprentissage en temps réel, que nous appelons ***Apprentissage Continu*** [12, 13], permet l'incorporation au modèle de nouvelles connaissances au fil du temps, garantissant qu'il reste efficace à tout moment pendant l'entraînement.

Une approche d'apprentissage continu (*continual learning*, CL) consiste à entraîner le modèle de manière incrémentale (en poursuivant l'entraînement avec des mises à jour relativement peu coûteuses basées uniquement sur les nouveaux échantillons de données). Si les incréments de données sont petits, une telle approche permet de limiter l'utilisation des ressources de calcul. Malheureusement, cela peut également entraîner une détérioration rapide des performances prédictives du modèle — un phénomène connu sous le nom d'*oubli catastrophique* [14]. Plus précisément, l'entraînement incrémental introduit un biais en faveur des nouveaux échantillons, conduisant le modèle à renforcer les connaissances récentes au détriment de celles précédemment acquises. Des différences plus importantes entre les distributions des anciennes et nouvelles données d'entraînement amplifient le biais, souvent au point où une seule passe (*epoch*) sur les nouvelles données suffit à effacer la plupart, sinon la totalité, des connaissances apprises précédemment.

Nous sommes donc confrontés au défi d'éviter efficacement l'oubli catastrophique. **D'une part, nous visons à atteindre une précision proche de celle obtenue en réentraînant le modèle à partir de zéro, tandis que, d'autre part, nous visons à atteindre des performances élevées (diminuant ainsi le temps d'entraînement), une bonne scalabilité ainsi qu'une faible utilisation des ressources comme dans le cas de l'entraînement incrémental**. L'apprentissage continu vise à résoudre ce compromis : en un sens, il atténue l'oubli catastrophique en dotant l'entraînement incrémental d'une stratégie pour maintenir les connaissances acquises plus tôt.

Les stratégies d'apprentissage continu proposées dans la littérature couvrent diverses méthodes, dont les suivantes : *répétition* (*rehearsal*) des échantillons d'entraînement historiques [15, 16, 17], co-entraînement d'un modèle génératif capable de reproduire les anciennes données en générant de nouveaux échantillons à la demande, régularisation contraignant les mises à jour des paramètres du modèle afin de limiter l'oubli catastrophique, et utilisation de composants spécifiques à la tâche courante (impliquant l'augmentation de l'architecture du modèle d'apprentissage profond avec des structures spécifiquement conçues pour éviter l'oubli catastrophique).

Dans cette thèse, nous nous concentrons sur **l'apprentissage continu basé sur la répétition**. Avec cette stratégie, les échantillons d'entraînement historiques représentatifs des données vues précédemment sont conservés dans un tampon de répétition de taille limitée. De

petits sous-ensembles d'échantillons du jeu de données (appelés *minibatches*) sont ensuite *augmentés* pour inclure des échantillons supplémentaires venant du tampon de répétition. Enfin, le tampon de répétition est mis à jour en remplaçant certains de ses échantillons d'entraînement par de nouveaux. Un avantage de cette stratégie d'apprentissage continu est qu'elle ne nécessite aucune modification de l'architecture du modèle, ni du processus d'entraînement. La *répétition* est intégrée dans le flux de données ingéré par le modèle. Par contraste, les autres approches nécessitent du code supplémentaire pour implémenter la régularisation, des modèles génératifs supplémentaires et/ou un composant architectural spécifiquement conçu pour le problème d'apprentissage en question.

Objectifs de recherche

Dans cette section, nous énumérons les trois principales contributions de cette thèse.

Mise à l'échelle de l'apprentissage continu basé sur la répétition intégré à l'entraînement avec parallélisme de données

Les travaux antérieurs sur l'apprentissage continu basé sur la répétition [15, 16, 17] utilisent un tampon de répétition local unique pour stocker les échantillons d'entraînement historiques à fournir au modèle d'apprentissage profond. Par ailleurs, l'*entraînement avec parallélisme de données* est largement utilisé pour réduire le temps d'entraînement. Il implique de répliquer le modèle d'apprentissage profond, afin que chaque réplique soit entraînée en parallèle avec différents fragments de données, tandis que les gradients sont moyennés pendant la rétropropagation pour maintenir les répliques synchronisées. Dans une telle configuration distributée, il est important de permettre un apprentissage continu basé sur la répétition performant, scalable et efficace en ressources sur **plusieurs GPU** exploitant le parallélisme de données. Résoudre cette limitation des approches de l'état de l'art est difficile pour deux raisons principales : (1) le tampon de répétition doit s'intégrer de manière transparente au pipeline de données responsable de la lecture asynchrone des données d'entrée et de leur mise à disposition aux itérations d'entraînement, ce qui implique de superposer la gestion du tampon de répétition avec à la fois le pipeline de données et la boucle d'entraînement du modèle ; (2) il n'est pas suffisant de simplement instancier des tampons de répétition indépendants, associés à chaque réplique du modèle pour permettre le parallélisme de données : au lieu de cela, des techniques distribuées sont nécessaires pour permettre aux tampons de répétition de collaborer au niveau global afin d'éviter les biais introduits par l'échantillonnage localisé.

Les recherches existantes traitent généralement des questions de l'apprentissage continu et de l'apprentissage profond distribué séparément, c'est-à-dire que les études sur l'apprentissage continu sont souvent menées sur un seul nœud de calcul [18, 19, 20]. Un premier objectif de

cette thèse est d'étudier comment les méthodes d'apprentissage continu peuvent tirer parti du parallélisme de données sur plusieurs nœuds, ce qui est l'une des principales techniques pour atteindre la scalabilité de la procédure d'entraînement sur les systèmes HPC. Les performances prédictives de tels algorithmes pourrait bénéficier de la mémoire aggrégée par l'instanciation **de tampons de répétition distribués**.

Permettre la mise à l'échelle des techniques d'apprentissage continu basées sur la répétition plus avancées

Avec une diversité croissante des stratégies de répétition pour l'apprentissage continu, il est important de découpler le tampon de répétition de la tâche d'apprentissage, de sorte qu'il devienne une abstraction générique qui implémente ses propres optimisations complémentaires. Rendre le tampon générique est difficile pour deux raisons principales : (1) le tampon de répétition doit stocker, échantillonner et remplacer des données hétérogènes indépendamment de la stratégie de répétition et de la tâche d'apprentissage en cours (par exemple, les tâches de classification vs. les tâches génératives) ; et (2) les techniques distribuées permettant le passage à l'échelle mentionnées précédemment doivent être exploitées pour que le système puisse stocker, supprimer et exposer ces données hétérogènes efficacement, à grande échelle, en utilisant des techniques spécialisées. De plus, rendre le tampon de répétition distribué générique par rapport aux échantillons de données stockés permet l'applicabilité de notre approche à des tâches d'apprentissage génératif, qui utilisent généralement plus de données par échantillon d'entraînement (plus précisément, les données de vérité (*ground truth*) ne consistent pas en une seule étiquette de classe comme avec les tâches de classification). Ainsi, dans le cas de telles tâches d'apprentissage continu génératif, le tampon doit également permettre la gestion unifiée des échantillons d'entraînement au lieu de la gestion séparée par classe des échantillons d'entraînement comme cela est fait avec les tâches de classification.

Nous ne sommes pas au courant d'une abstraction de tampon de répétition qui offre un support pour une large gamme de paramètres d'apprentissage continu basés sur la répétition. Un deuxième objectif de cette thèse est d'étudier comment une disposition de données générique peut aider à implémenter des stratégies de répétition plus avancées, améliorant ainsi la précision tout en bénéficiant de l'évolutivité de l'entraînement sur les systèmes HPC.

Illustration des avantages de l'apprentissage continu génératif pour un cas d'utilisation réel

Les flux de travaux HPC modernes ne se limitent pas aux clusters HPC : ils doivent acquérir des données en temps réel à partir d'instruments scientifiques situés à la périphérie (*edge*), les envoyer aux machines HPC pour des analyses supplémentaires, et éventuellement agir sur les résultats en temps réel (par exemple, pour calibrer l'instrument scientifique afin d'**orienter**

l'expérience dans une direction spécifique). Cependant, les algorithmes et simulations conventionnels utilisés dans ces flux de travaux pour l'analyse de données sont intensifs en calcul, et ne peuvent donc pas traiter les grandes quantités de données générées par ces instruments à la volée. Des débits de données supérieurs à 10 Gbps [21] peuvent être atteints. Pour relever ce défi, les scientifiques se sont tournés vers les méthodes d'apprentissage profond, remplaçant les techniques d'analyse conventionnelles par des modèles de substitution beaucoup plus rapides. Cependant, ces modèles nécessitent des mises à jour continues pour maintenir leur performance prédictive et s'adapter aux conditions expérimentales évolutives. Ainsi, les chercheurs ont développé des flux de travaux innovants qui intègrent les trois composantes suivantes : (1) l'acquisition de données à la périphérie, (2) l'entraînement en ligne, en parallèle, d'un modèle de substitution utilisant des techniques d'apprentissage continu sur un cluster HPC, et après un certain temps, (3) l'inférence en temps réel utilisant le modèle obtenu à la périphérie. En exploitant l'apprentissage continu, ces flux de travaux peuvent améliorer leur précision au cours de l'expérience, atteignant finalement un état où le modèle peut être utilisé seul pour l'inférence. Cette approche a le potentiel de réduire considérablement les coûts de calcul et de permettre le traitement des données en temps réel dans les configurations idéales.

Pour démontrer l'efficacité de cette approche dans un cadre réel, nous visons à appliquer nos contributions à un cas d'utilisation spécifique, en exploitant l'apprentissage continu pour améliorer la qualité de prédiction d'un modèle génératif. Notre objectif est de montrer que cette méthode peut atteindre des performances acceptables par rapport aux simulations conventionnelles, tout en accélérant la convergence de l'entraînement du modèle grâce à l'apprentissage continu.

Contributions

Les principales contributions de cette thèse, répondant à chacun des trois objectifs précédents, peuvent être résumées comme suit :

Un tampon de répétition distribué pour permettre l'apprentissage continu à grande échelle

Nous proposons une nouvelle abstraction de tampon de répétition visant à exploiter efficacement les systèmes *distribués*. Plus précisément, nous (1) définissons le concept de tampons de répétition pour traiter l'apprentissage continu et introduisons des **extensions pour les utiliser dans l'entraînement avec parallélisme de données**, nous (2) présentons des principes clés tels que **des techniques asynchrones** permettant de masquer le surcoût de gestion des tampons de répétition et de générer un large éventail de combinaisons pour **les augmentations de mini-batches**, et nous (3) illustrons les avantages d'une telle approche sur une tâche d'apprentissage de classification couramment utilisées dans la communauté de vision par ordinateur.

Nous étudions ensuite les compromis introduits par l'apprentissage continu à grande échelle en termes de temps d'entraînement, de précision du modèle et d'utilisation de la mémoire en utilisant le benchmark ImageNet. Pour ce faire, nous réalisons des expériences jusqu'à 128 GPU du supercalculateur ThetaGPU, afin de comparer notre approche avec des références représentatives de l'entraînement à partir de zéro (la limite supérieure –optimale– en termes de précision) et de l'entraînement incrémental (la limite inférieure –optimale– en termes de temps d'entraînement). Les résultats montrent que l'apprentissage continu basé sur la répétition atteint une précision de classification top-5 proche de la limite supérieure, tout en présentant un temps d'exécution proche de la limite inférieure.

Ce travail est une collaboration avec Bogdan Nicolae et Ian Foster du Laboratoire National d'Argonne (USA), dans le cadre de l'équipe associée UNIFY. Il a été publié lors de la conférence IEEE/ACM CCGrid 2024 (veuillez consulter [22]).

Un tampon distribué générique en soutien à des techniques de répétition plus avancées

Cette contribution introduit un ensemble d'abstractions qui n'imposent aucune contrainte particulière sur la la forme des données d'entraînement stockés dans le tampon de répétition, permettant ainsi l'adaptation du système à une large gamme de tâches d'apprentissage profond et de stratégies de répétition. En outre, nous introduisons le concept de **tuples annotés** de tenseurs afin de servir les échantillons d'entraînement représentatifs et leurs états associés au framework d'IA. Une telle approche (1) répond au besoin de supporter davantage de stratégies de répétition (notamment des stratégies utilisant la distillation de connaissances) tout en (2) augmentant de manière transparente les minibatches produits par les pipelines de données des différents nœuds de calcul.

Les premiers résultats de ce travail ont été obtenus lors du stage de license de Malvin Chevallier, dans le cadre de l'obtention de son diplôme d'ingénieur.

Intégration avec une application de streaming HPC réelle bénéficiant de l'apprentissage continu génératif

Les algorithmes d'analyse de données conventionnels sont souvent intensifs en calcul et peinent à traiter les grandes quantités de données générées par les instruments scientifiques à la volée. Pour relever ce défi, des modèles d'apprentissage automatique sont intégrés dans les flux de travaux HPC afin d'accélérer l'analyse de données. Une approche courante pour l'application de tels modèles dans ce contexte est la suivante : au début de l'expérience, une analyse classique, coûteuse en calcul, est effectuée sur les données d'entrée envoyées au cluster HPC. Le résultat de ce flux de travaux consommateur dédié sert de données de vérité (*ground truth*) pour un modèle entraîné en parallèle. À mesure que l'expérience progresse, si le modèle apprend à prédire les données de vérité avec une grande fidélité, une copie peut être envoyée à la périphérie et être

utilisée à la place de l'analyse classique, améliorant ainsi la latence et le débit du traitement à long terme. Bien que cette approche entraîne un surcoût initial plus élevé en raison du besoin de ressources HPC supplémentaires pour entraîner le modèle, elle permet finalement une analyse de données plus rapide. De plus, l'entraînement du modèle peut être poursuivi même après le passage au régime d'inférence, permettant l'envoi de mises à jour périodiques à la périphérique garantissant une qualité améliorée avec le temps. Pour que ce flux de travaux soit efficace, deux métriques clés sont cruciales : (1) la qualité des prédictions du modèle et (2) la vitesse à laquelle le modèle converge dans le cadre de l'apprentissage continu.

Nous présentons des résultats pour le cas des tâches d'apprentissage génératif, que nous illustrons avec une application reposant sur des flux de données : la reconstruction d'images ptychographiques [23]. À cette fin, nous présentons des résultats obtenus en utilisant le modèle génératif PtychoNN [24] dans le contexte de l'apprentissage continu pour deux approches différentes à la répétition (sélection aléatoire simple des échantillons représentatifs et Dark Experience Replay [25]). Les résultats montrent un gain par rapport à l'apprentissage continu basé sur l'entraînement incrémental et à la reconstruction conventionnelle basée sur des algorithmes intensifs en calcul, en qualité dans un cas et en vitesse d'exécution dans l'autre.

Ce travail est une collaboration avec Bogdan Nicolae, Tekin Bicer et Ian Foster du Laboratoire National d'Argonne (USA), dans le cadre de l'initiative Joint Laboratory for Extreme Scale Computing (JLESC). Il a été publié sous forme d'article de journal dans **FGCS** - *Future Generation Computer Systems, JLESC Special Issue* (veuillez consulter [26]).

Publications

Revues Internationales

- **Thomas Bouvier**, Bogdan Nicolae, Hugo Chaugier, Alexandru Costan, Ian Foster, Gabriel Antoniu. Efficient Data-Parallel Continual Learning with Asynchronous Distributed Rehearsal Buffers. **ACM/IEEE CCGrid 2024** - *The 24th IEEE/ACM international Symposium on Cluster, Cloud and Internet Computing*, May 2024, Philadelphia, United States. Citation : [22].

Conférences Internationales

- **Thomas Bouvier**, Bogdan Nicolae, Alexandru Costan, Tekin Bicer, Ian Foster, Gabriel Antoniu. Efficient Distributed Continual Learning for Steering Experiments in Real-Time. **FGCS** - *Future Generation Computer Systems, JLESC (Joint Laboratory for Extreme Scale Computing) Special Issue*. Citation : [26].

Posters dans des Conférences Internationales

— **Thomas Bouvier**, Alexandru Costan, Gabriel Antoniu. Heterogeneity-aware Deep Learning Workload Deployments on the Computing Continuum. **IPDPS 2021** - *35th IEEE International Parallel and Distributed Processing Symposium*, May 2021, Portland (virtuel), United States. Citation : [27].

Posters dans des Conférences Nationales

— **Thomas Bouvier**, Alexandru Costan, Gabriel Antoniu. Deploying Heterogeneity-aware Deep Learning Workloads on the Computing Continuum. **BDA 2021** - *37ème Conférence sur la Gestion de Données - Principes, Technologies et Applications*, Oct 2021, Paris, France. Citation : [28].

Logiciels

Contributions Principales

Neomem [29]

Description scientifique : L'apprentissage continu basé sur la répétition est une approche prometteuse pour contrer le problème de l'oubli catastrophique, mais les recherches à ce jour n'ont pas abordé les questions de performance et de scalabilité de l'entraînement. Pour combler ce manque, nous proposons une approche basée sur un tampon de répétition distribué qui complète efficacement l'entraînement avec parallélisme de données sur plusieurs GPU, afin d'atteindre une grande précision, un temps d'entraînement court, et une bonne scalabilité. Cette approche exploite un ensemble de tampons (locaux à chaque GPU) et utilise plusieurs techniques asynchrones pour mettre à jour ces tampons locaux de manière parallèle (*embarrassingly parallel*), tout en gérant le surcoût de communications nécessaires pour augmenter les mini-batches d'entrée en utilisant un échantillonnage global non biaisé.

Description fonctionnelle : L'entraînement des réseaux de neurones à partir de données générées en continu pose des défis liés à l'oubli des connaissances précédemment acquises, un phénomène connu sous le nom d'*oubli catastrophique*. Une solution efficace consiste à rejouer certaines données précédemment observées pour maintenir les connaissances associées. *Neomem* met en œuvre cette approche, visant à obtenir d'excellentes performances prédictives au prix d'une légère augmentation du temps d'entraînement. Notre approche permet l'exploitation du parallélisme de données sur des dizaines de GPU, la rendant applicable aux simulations scientifiques au sein de la communauté du calcul haute performance (HPC).

— **Lien :** https://github.com/thomas-bouvier/neomem

— **Taille et langage(s) :** ~2000 lignes, C++ et ~1000 lignes de code de test, Python.

— **Licence :** GNU General Public License v3.0

distributed-continual-learning [30]

Description scientifique : L'apprentissage continu nécessite d'apprendre à partir d'un flux de données non stationnaire, un problème fondamental pour un entraînement durable et efficace des réseaux de neurones profonds au fil du temps. Malheureusement, les bibliothèques d'apprentissage profond ne fournissent que des primitives pour l'entraînement hors ligne, supposant que les données d'entrée sont (1) indépendantes et identiquement distribuées (i.i.d.) et (2) disponibles avant le processus d'entraînement. *distributed-continual-learning* fournit un ensemble d'outils et d'abstractions nécessaires pour expérimenter avec l'apprentissage continu.

Description fonctionnelle : *distributed-continual-learning* est une bibliothèque open source qui étend PyTorch en fournissant un support de première classe pour les flux de jeux de données, l'entraînement incrémental et le parallélisme de données en utilisant des tampons de répétition distribués. Cette bibliothèque offre un support pour une large gamme de scénarios d'apprentissage continu, de tâches d'apprentissage (problèmes de classification et de génération) et de stratégies d'apprentissage continu basées sur la répétition.

— **Lien :** https://github.com/thomas-bouvier/distributed-continual-learning
— **Taille et langage(s) :** ∼4000 lignes, Python.
— **Licence :** GNU General Public License v3.0

Contribution à un logiciel existant

Spack [31]

Description scientifique : La complexité croissante des librairies HPC pose des défis significatifs pour les utilisateurs de clusters HPC. Les dépendances complexes des applications scientifiques, qui nécessitent des versions spécifiques de compilateurs, d'implémentations de standards comme Message Passing Interface (MPI), de plateformes comme Compute Unified Device Architecture (CUDA) et d'autres bibliothèques de dépendances, rendent l'adoption d'une seule pile logicielle standardisée difficile. De plus, la taille combinatoire de l'espace de configuration complexifie la gestion de plusieurs configurations. En réponse à ces défis, Spack apparaît comme un outil indispensable conçu pour automatiser la gestion de tels environnements HPC. En fournissant un cadre flexible et extensible pour la gestion des dépendances et des configurations logicielles, Spack améliore la productivité de la recherche scientifique.

Description fonctionnelle : Spack fournit une syntaxe de spécification récursive pour invoquer des compilations paramétrées de packages et de dépendances. Il permet à un nombre quelconque de résultats de compilation de coexister sur le même système et garantit que les packages installés peuvent localiser leurs dépendances indépendamment de l'environnement. Spack

utilise des directives de package déclaratives (appelées *specs*) pour exprimer toutes les compatibilités intersectées et disjointes qu'une version d'un package a avec une autre. Spack manipule également le concept de *variantes*, ou données clé-valeur, qui sont déclarées dans la recette (*recipe*) du package, et avec les versions peuvent également être configurées par l'utilisateur final via des paramètres en ligne de commande.

Notre contribution : Les recettes de packages permettent à Spack d'encapsuler la logique de compilation pour différentes versions, compilateurs, options de compilation, plateformes, microarchitectures GPU et combinaisons de dépendances au même endroit. En substance, une recette traduit une *spec* en logique de compilation exécutée par le système de compilation du package en question. Les *variantes* correspondent souvent directement aux flags qui y sont intégrés. J'ai écrit des recettes pour de nombreux packages dont *Neomem* et *distributed-continual-learning* dépendent, pour qu'ils puissent être installés sur des supercalculateurs facilement en utilisant Spack. En particulier : `py-nvidia-dali`, `py-continuum`, `nvtx`, `py-imagehash`, `thrift`, `py-datasets`, `fribidi`, `py-wandb`, `py-e2clab`, et `py-codecarbon`. J'ai également corrigé des problèmes dans les recettes suivantes : `arrow`, `py-arrow`, `py-horovod`, `scipy`, `py-pytest-mock`, et `py-dm-tree`.

- **Lien :** https://github.com/spack/spack
- **Taille et langage(s) :** le volume de mes contributions à la base de code existante est d'environ 800 lignes, Python.
- **Licence :** Apache 2.0

Horovod [32]

Description scientifique : L'entraînement des modèles d'apprentissage profond nécessite de grandes quantités de calcul, souvent fournies par des GPUs dans des clusters spécialisés. Afin d'exploiter de nombreux GPUs permettant d'accélérer ce processus, Horovod utilise les principes de MPI pour distribuer efficacement les gradients. Quelques-unes des primitives utilisées sont `size`, `rank`, `local rank`, `allreduce`, `allgather`, `broadcast`, et `alltoall`. Selon les techniques employées, cette communication peut entraîner un surcoût allant de négligeable à significatif : Horovod utilise ainsi une communication inter-GPU efficace via une réduction en anneau (*ring reduction*). En outre, Horovod prend en charge la bibliothèque de communications collectives NVIDIA (NCCL) pour optimiser davantage la communication entre les GPUs, lui permettant d'obtenir une bonne efficacité de mise à l'échelle sur des tâches d'entraînement distribuées.

Description fonctionnelle : Horovod est conçu pour prendre un script d'entraînement mono-GPU et l'exécuter sur plusieurs GPUs ou même plusieurs machines en exploitant le parallélisme de données, tout en nécessitant seulement quelques lignes de modification du code utilisateur. Horovod s'intègre parfaitement avec des frameworks d'apprentissage profond populaires tels

que TensorFlow ou PyTorch, tout en permettant un entraînement distribué efficace grâce à une communication inter-GPU efficace via une réduction en anneau (*ring reduction*).

Notre contribution : J'ai corrigé certains bugs dans la base de code originale, notamment ceux liés à la prise en charge de C++17, nécessaire pour compiler avec PyTorch >= 2.1. Au cours de ma thèse de trois ans, j'ai observé une diminution des contributions d'Uber au projet Horovod. Cette tendance s'est poursuivie au point où le projet n'est plus maintenu depuis un an. La communauté utilise maintenant les solutions d'entraînement distribuées intégrées dans PyTorch, qui peuvent désormais tirer parti du matériel HPC. Afin de conserver la même base de code pour tous mes travaux de doctorat, j'ai forké le projet original pour le maintenir moi-même sous le nom de Khorovod.

— **Lien :** https://github.com/khorovod-ai/khorovod
— **Taille et langage(s) :** $\sim$100 lignes, Python/C++.
— **Licence :** Apache 2.0

E2Clab [33]

Description scientifique : E2Clab est un framework conçu pour permettre aux chercheurs de reproduire le comportement des applications dans un environnement contrôlé. Son objectif principal est d'analyser les performances de bout en bout des applications en liant les résultats à des configurations de paramètres spécifiques. E2Clab offre une méthodologie pour répondre à des questions telles que : *Comment identifier les goulots d'étranglement de l'infrastructure ? Quels paramètres système et réglages d'infrastructure influencent les performances, et dans quelle mesure ?*

Description fonctionnelle : E2Clab présente des fonctionnalités de haut niveau qui permettent : (1) des expériences garantissant la répétabilité, la réplicabilité et la reproductibilité de la recherche ; (2) la cartographie des composants de l'application (edge, fog et cloud/HPC) sur le banc d'essai physique ; (3) la variation et la mise à l'échelle progressive des scénarios expérimentaux ; et (4) la gestion des expériences pour le déploiement, l'exécution et le monitoring sur des plateformes telles que Grid'5000, Chameleon et FIT IoT LAB.

Notre contribution : J'ai ajouté le concept de *configurations d'application* pour initier des déploiements se rattachant à une configuration encapsulant des hyperparamètres spécifiques. Cette fonctionnalité permet une expérimentation plus rapide en réutilisant des configurations déjà testées et s'intègre parfaitement avec *distributed-continual-learning*. Des hyperparamètres spécifiques peuvent être remplacés via la ligne de commande.

— **Lien :** https://gitlab.inria.fr/E2Clab/e2clab
— **Taille et langage(s) :** $\sim$3000 lignes, Python
— **Licence :** GNU General Public License v3.0

Organisation de cette thèse

Cette thèse est organisée en huit chapitres.

Le Chapitre 1 résume les objectifs de notre recherche, ainsi que les contributions et publications résultantes.

Le Chapitre 2 présente le contexte de notre recherche. Nous y introduisons la pertinence de l'Apprentissage Continu pour réduire la latence entre l'acquisition des données et la génération des informations issues par les modèles d'apprentissage. Ensuite, il discute des défis à résoudre liés à (1) l'apprentissage à partir de données non stationnaires, un problème introduisant un biais connu sous le nom d'oubli catastrophique dans la communauté d'apprentissage continu ; (2) la mise à l'échelle des tâches d'entraînement de modèles d'apprentissage profond, en guarantissant à la fois une bonne généralisation statistique et une bonne utilisation des ressources ; et (3) la mise à l'échelle des tâches d'entraînement de modèles d'apprentissage continu en tenant compte de ces deux contraintes.

Le Chapitre 3 définit le concept de tampons de répétition pour remédier à l'oubli catastrophique dans l'apprentissage continu, et introduit les **extensions nécessaires pour les utiliser dans l'entraînement avec parallélisme de données** sur de nombreux GPUs.

Dans le Chapitre 4, nous introduisons des principes de conception tels que des **techniques asynchrones pour masquer le surcoût de gestion des tampons de répétition**, afin de permettre un spectre complet de combinaisons pour les augmentations de minibatches. Nous y parvenons en échantillonnant les tampons de répétition des répliques de modèles d'apprentissage résidant sur les nœuds distants, en utilisant des schémas de communication all-to-all à faible surcoût, et compatibles RDMA. Nous implémentons un **prototype de tampon de répétition distribué que nous avons intégré avec PyTorch**, un framework d'IA largement utilisé.

Dans le Chapitre 5, nous étudions les compromis introduits par l'apprentissage continu à grande échelle en termes de temps d'entraînement, de précision et d'utilisation de la mémoire en utilisant le benchmark ImageNet. Nous rapportons des expériences approfondies utilisant **128 GPUs du supercalculateur ThetaGPU du Laboratoire National d'Argonne** pour le cas des tâches de classification. À cette fin, nous présentons trois modèles différents (ResNet-50, ResNet-18, GhostNet-50) et quatre tâches dérivées du jeu de données ImageNet-1K. Les résultats montrent que notre approche peut améliorer la précision d'évaluation top-5 de 23,1% à 80,55% par rapport à l'entraînement incrémental, avec seulement une légère augmentation du temps d'exécution.

Dans le Chapitre 6, nous proposons une **généralisation des tampons de répétition** capables de stocker des **données hétérogènes**, et de les servir sous forme de **tuples annotés** de tenseurs. Cette extension permet d'exposer des échantillons représentatifs ainsi que leurs états associés au framework d'AI durant l'entraînement, ce qui répond au besoin de supporter des stratégies de répétition plus avancées. Nous discutons ensuite de la manière dont le tampon de répétition

distribué peut être utilisé pour permettre des tâches d'apprentissage continu avec des modèles génératifs sur plusieurs GPUs.

Dans le Chapitre 7, nous motivons les avantages de l'apprentissage continu pour les applications scientifiques basées sur le streaming, en les illustrant dans le contexte de la reconstruction ptychographique [23] avec des modèles génératifs —une application de streaming HPC en conditions réelles —. Nous rapportons des résultats expérimentaux obtenus avec le modèle PtychoNN [24] pour deux types différents de répétition (sélection aléatoire simple et Dark Experience Replay), qui se montrent plus efficaces que l'apprentissage continu basé sur l'entraînement incrémental et à la reconstruction traditionnelle basée sur des algorithmes intensifs en calcul.

Enfin, le Chapitre 8 conclut cette thèse et présente les perspectives ouvertes par nos contributions.

Chapter 1

Introduction

Contents

1.1 Context 14

1.2 Research Objectives 16

1.3 Contributions 18

1.4 Publications 19

1.5 Software 20

1.5.1 Main Contributions 20

1.5.2 Contribution to Existing Software (via Pull Requests) 21

1.6 Organization of this Dissertation 23

1.1 Context

Deep learning (DL) models are rapidly gaining traction both in industry and scientific computing in many areas, including speech and vision [1], climate science [2], fusion energy science [3], cancer research [4], personalized medicine [5] and pandemics [6]. As data sizes and pattern complexity keep increasing, DL models capable of learning such data patterns have evolved from all perspectives: size (number of parameters), depth (number of layers/tensors), and structure (directed graphs that feature divergent branches, fork-join, etc.). Despite increasing convergence between DL and High-Performance Computing (HPC) [7], which has led to the adoption of various parallelization techniques [8] (data-parallel, model parallel, hybrid), the training of DL models remains a time-consuming and resource-intensive task. Indeed, the amount of computation used in the largest AI training runs has doubled about every 6 months since 2010 [9].

DL models are typically trained on large, many-GPU systems that have access to all training data from the beginning (e.g., from a parallel file system), by using an iterative optimization technique (e.g., stochastic gradient descent) to revisit the training data repeatedly until convergence. Today, however, DL applications increasingly need to be trained with unbounded datasets that are updated frequently. For example, scientific applications using experimental

devices such as sensors need to quickly analyze the experimental data in order to steer an ongoing experiment (e.g., trigger an automated decision). In this case, repeatedly retraining the model from scratch as new samples arrive is not an option: as training data keeps accumulating, this would take increasingly longer and consume more resources (compute and storage space), leading to prohibitive runtimes. Other use cases like digital twins [10, 11] typically require a hybrid execution infrastructure: edge devices create streams of input data, which are processed by data analytics and machine learning applications in the Cloud, and simulations on large, specialized HPC systems provide insights into and prediction of future system state. Such applications rely on dynamic interactions that prompt the need to accommodate data acquired in real-time. This paradigm opens new perspectives for real-time steering of computations, dynamic workflow reconfiguration, or re-calibration of parameters at runtime. Such a real-time learning approach, that we refer to as ***Continual Learning*** [12, 13], enables the incorporation of new knowledge over time, ensuring that the model remains accurate at any time during the training.

One approach to the Continual Learning (CL) problem is to train the DL model incrementally (i.e., the training proceeds with relatively inexpensive updates to the model's parameters based on just the new data samples). If data increments are small, such an approach achieves low resource utilization. Unfortunately, it can also cause the accuracy of the DL model to deteriorate quickly—a phenomenon known as *catastrophic forgetting* [14]. Specifically, incremental training introduces a bias in favor of new samples, causing the model to reinforce recent patterns at the expense of previously acquired knowledge. Larger differences between the distributions of the old vs. new training data amplifies the bias, often to the point where a single pass over the new training data is enough to erase most, if not all, of the patterns learned previously.

Thus, we are faced with the challenge of avoiding catastrophic forgetting efficiently. **On the one hand, we aim to achieve an accuracy close to the one achieved by retraining the DL model from scratch, while, on the other hand, we aim to achieve high performance (decreasing the overall training time), scalability, and low resource utilization just like incremental training**. Continual learning aims to address this trade-off: in a broad sense, it mitigates catastrophic forgetting by complementing incremental training with a strategy to retain patterns seen earlier.

Proposed CL strategies include various methods: *rehearsing* historic training samples [34], co-training a generative DL model that can mimic old patterns by generating new samples on demand, regularization (i.e., rules that constrain DL model parameter updates to prevent catastrophic forgetting), and instantiating task-specific components (i.e., augmenting the DL model architecture with structures specifically designed to avoid catastrophic forgetting).

In this thesis, we focus on **continual learning based on rehearsal**. With this strategy, historic training samples that are representative of patterns seen earlier are retained in a limited-size rehearsal buffer. Small subsets of the dataset's training samples (called *minibatches*) are then

augmented to include additional samples from the rehearsal buffer. Finally, the rehearsal buffer is updated by replacing some of its training samples with newer ones. A benefit of this CL strategy is that it requires no modifications to either the DL model architecture or the training process. Instead, *rehearsal* is integrated into the data pipeline ingested by the model. In contrast, other CL approaches require additional code to implement regularization, additional generative DL models, and/or additional architectural component specifically designed for the learning problem at hand.

1.2 Research Objectives

In this thesis, we are interested in the three following research objectives:

Scaling Rehearsal-based Continual Learning in the Context of Data-parallel Training

Prior work on rehearsal-based CL [16, 17] has employed a single local rehearsal buffer to retain historic training samples fed to the DL model. On another note, *data parallel* training is widely used to reduce the training time. It involves DL model replicas that are trained in parallel with different data shards, while the gradients are averaged during the backpropagation to keep the replicas in sync. In such a distributed setting, it becomes important to enable high-performance, scalable, and resource-efficient rehearsal based CL on **multiple GPUs** leveraging data-parallel training. Addressing this limitation of state of art approaches is challenging for two main reasons: (1) the rehearsal buffer needs to seamlessly integrate with the data pipeline responsible for asynchronously reading the input data and feeding it to the training iterations, which implies overlapping the management of the rehearsal buffer with both the data pipeline and the training loop; (2) it is not enough to simply instantiate independent rehearsal buffers associated with each DL model replica to enable data parallelism: instead, distributed techniques are needed to enable the rehearsal buffers to collaborate at global level in order to avoid biases introduced by localized sampling.

Existing research typically addresses CL and distributed DL separately i.e., studies on continual learning are often conducted on a single compute node [18, 19, 20]. A first objective of this thesis is to study how CL methods can take advantage of data parallelization across nodes, which is one of the main techniques to achieve training scalability on HPC systems. The aggregated memory could benefit the accuracy achieved by such algorithms by instantiating **distributed rehearsal buffers**.

Enabling Scalability of More Advanced Rehearsal-based Continual Learning Techniques

With a growing diversity of rehearsal strategies for continual learning, it becomes important to decouple the rehearsal buffer from the learning task, such that it becomes a generic

abstraction that implements its own complementary optimizations. Making the buffer generic is challenging for two main reasons: (1) the rehearsal buffer needs to store, sample and replace heterogeneous data independently from both the rehearsal strategy and the learning task at hand (e.g., classification tasks vs. generative tasks); and (2) the scalable distributed techniques mentioned previously should be leveraged to enable the system to store, remove and expose such heterogeneous data efficiently at scale using specialized data management techniques. Besides, making the distributed rehearsal buffer generic with respect to the retained data samples further allows for the applicability of our approach to generative learning tasks, which typically leverage more data per training sample (specifically, the *ground truth* data doesn't consist of a single class label as with classification tasks). Thus, in the case of such generative CL tasks, the buffer should also allow for the unified management of training samples instead of the separate per-class management of training samples as done with classification tasks.

We are not aware of any rehearsal buffer abstraction that provides support for a broad range of rehearsal-based CL settings. A second objective of this thesis is to study how a generic data layout can help implement more advanced rehearsal strategies, further improving accuracy while benefiting from training scalability on HPC systems.

Illustrating the Benefits of Generative Continual Learning for a Real-life Use-case

Modern HPC workflows are not limited to HPC clusters: they need to acquire data in real-time from scientific instruments located at the edge, send it to HPC machines for further analytics, and optionally act on the results in real-time (e.g., calibrate the scientific instrument to **steer the experiment in a specific direction**). However, conventional algorithms and simulations used in these workflows for data analysis are computationally intensive and cannot keep up with the exploding amounts of data generated by these instruments. Data rates greater than 10 Gbps [21] can be reached. To address this challenge, scientists have turned to deep learning methods, replacing conventional analysis techniques with much faster surrogate DL models. However, these models require continuous updates to maintain their accuracy and adapt to changing experimental conditions. Thus, researchers have developed innovative workflows that integrate the three following components: (1) data acquisition at the edge, (2) online training of a surrogate DL model using continual learning techniques on an HPC cluster, and eventually, (3) live DL inference using the model after convergence at the edge. By leveraging continual learning, these workflows can improve in accuracy over the course of the experiment, ultimately achieving a state where the model alone can be used for inference. This approach has the potential to significantly reduce computational costs and enable real-time data processing in ideal settings.

To demonstrate the effectiveness of this approach in a real-life setting, we aim to apply our contributions to a specific use-case, leveraging continual learning to improve the accuracy of a

generative model. Our goal is to show that this method can achieve acceptable accuracy compared with conventional simulations, while also accelerating the training convergence of the model through continual learning.

1.3 Contributions

The main contributions of this dissertation addressing each of the three previous objectives can be summarized as follows:

A Distributed Rehearsal Buffer to Enable Continual Learning at Scale

We propose a novel rehearsal buffer abstraction that aims to leverage *distributed* systems effectively. Specifically, we (1) define the concept of rehearsal buffers to address continual learning, and introduce **extensions to leverage them for data-parallel training**, we (2) present key design principles such as **asynchronous techniques** allowing to hide the overhead of managing rehearsal buffers and to generate a full spectrum of combinations for **minibatch augmentations**, and we (3) illustrate the benefits of such an approach using a classification learning task commonly used in the computer vision community.

We then study the trade-offs introduced by large-scale CL in terms of training time, accuracy and memory usage using the ImageNet benchmark. To do so, we run extensive experiments on up to 128 GPUs of the ThetaGPU supercomputer to compare our approach with baselines representative of training-from-scratch (the upper bound in terms of accuracy) and incremental training (the lower bound in terms of training time). Results show that rehearsal-based continual learning achieves a top-5 classification accuracy close to the upper bound, while simultaneously exhibiting a runtime close to the lower bound.

This work is a collaboration with Bogdan Nicolae and Ian Foster from Argonne National Laboratory (USA), held in the context of the UNIFY associate team. It has been published at IEEE/ACM CCGrid 2024 conference (please see [22]).

A Generic Distributed Buffer Generic in Support of More Advanced Rehearsal Techniques

This contribution introduces a set of abstractions that do not impose any particular constraints on the data shape of training samples retained in the rehearsal buffer, allowing to accommodate a large range of deep learning workloads and rehearsal strategies. Besides, we introduce the concept of **annotated tuples** of tensors to serve representative training samples and their associated states conveniently to the AI runtime. Such an approach (1) addresses the need to support more rehearsal strategies (notably strategies leveraging knowledge distillation) while (2) transparently augmenting minibatches produced by data pipelines of data-parallel CL training instances.

Early results of this work were obtained during the bachelor internship of Malvin Chevallier.

Integration with a Real-life HPC Streaming Application Benefiting from Generative Continual Learning

Conventional data analytics algorithms are often computationally intensive, preventing the processing of the large amounts of data generated by scientific instruments in real time. To address this challenge, DL models are integrated into HPC workflows to accelerate data analysis. A common pattern for the application of DL models in this context is as follows: at the beginning of the experiment, a classic, computationally expensive analysis is performed on the input data sent to the HPC cluster. The output generated by this dedicated consumer workflow is served as the ground-truth data for a DL model trained in parallel. As the experiment progresses, if the DL model learns to predict the ground-truth data with high fidelity, one can send a copy of it to the edge and use it instead of the classic analysis, thus improving the long-term latency and throughput. Although this approach incurs a higher initial overhead due to the need for additional HPC resources to train the model, it ultimately enables faster data analytics. Furthermore, model training may be continued even after the switch to the inference regime, allowing for periodic updates to be sent to the edge device ensuring improved quality over time. For this workflow to be effective, two key metrics are crucial: (1) the quality of the model's predictions and (2) the speed at which the model converges under continual learning.

We report on experiments for the case of generative learning tasks illustrated through a real-life HPC streaming application: ptychographic image reconstruction [23]. To this end, we showcase results obtained using the PtychoNN [24] generative DL model in the context of CL for two different approaches for rehearsal (simple random selection of representative samples and Dark Experience Replay [25]). Results compare favorably to CL based on incremental training and to conventional reconstruction based on algorithms that are computationally intensive, in terms of result quality in one case and in terms of time-to-solution in the other.

This work is a collaboration with Bogdan Nicolae, Tekin Bicer and Ian Foster from Argonne National Laboratory (USA), in the context of the Joint Laboratory for Extreme Scale Computing (JLESC) initiative. It has been published as a journal paper at **FGCS** - *Future Generation Computer Systems, JLESC Special Issue* (please see [26]).

1.4 Publications

International Conferences

— **Thomas Bouvier**, Bogdan Nicolae, Hugo Chaugier, Alexandru Costan, Ian Foster, Gabriel Antoniu. Efficient Data-Parallel Continual Learning with Asynchronous Distributed Re-

hearsal Buffers. **ACM/IEEE CCGrid 2024** - *The 24th IEEE/ACM international Symposium on Cluster, Cloud and Internet Computing*, May 2024, Philadelphia, United States. Citation: [22].

Journal Articles

— **Thomas Bouvier**, Bogdan Nicolae, Alexandru Costan, Tekin Bicer, Ian Foster, Gabriel Antoniu. Efficient Distributed Continual Learning for Steering Experiments in Real-Time. **FGCS** - *Future Generation Computer Systems, JLESC (Joint Laboratory for Extreme Scale Computing) Special Issue*. Citation: [26].

Posters at International Conferences

— **Thomas Bouvier**, Alexandru Costan, Gabriel Antoniu. Heterogeneity-aware Deep Learning Workload Deployments on the Computing Continuum. **IPDPS 2021** - *35th IEEE International Parallel and Distributed Processing Symposium*, May 2021, Portland (virtual), United States. Citation: [27].

Posters at National Conferences

— **Thomas Bouvier**, Alexandru Costan, Gabriel Antoniu. Deploying Heterogeneity-aware Deep Learning Workloads on the Computing Continuum. **BDA 2021** - *37ème Conférence sur la Gestion de Données - Principes, Technologies et Applications*, Oct 2021, Paris, France. Citation: [28].

1.5 Software

1.5.1 Main Contributions

Neomem [29]

Scientific Description: Rehearsal-based continual learning has shown promise for addressing the catastrophic forgetting challenge, but research to date has not addressed training performance and scalability. To fill this gap, we propose an approach based on a distributed rehearsal buffer that efficiently complements data-parallel training on multiple GPUs to achieve high accuracy, short runtime, and scalability. It leverages a set of buffers (local to each GPU) and uses several asynchronous techniques for updating these local buffers in an embarrassingly parallel fashion, all while handling the communication overheads necessary to augment input minibatches using unbiased, global sampling.

Functional Description: Training neural networks with continuously generated data poses challenges related to forgetting previously acquired knowledge, a phenomenon known as *catastrophic forgetting*. An effective solution involves replaying certain previously observed data to maintain associated knowledge. *Neomem* implements this approach, aiming to achieve excellent predictive performance at the cost of a slight increase in training time. Our approach allows for the data-parallel utilization of dozens of GPUs, making it applicable to scientific simulations within the high-performance computing (HPC) community.

— **Link:** https://github.com/thomas-bouvier/neomem
— **Size and language(s):** ∼2000 lines, C++ and ∼1000 lines of testing code, Python.
— **License:** GNU General Public License v3.0

distributed-continual-learning [30]

Scientific Description: Continual learning is the problem of learning from a non-stationary stream of data, a fundamental issue for sustainable and efficient training of deep neural networks over time. Unfortunately, deep learning libraries only provide primitives for offline training, assuming that the input data is (1) independent and identically distributed (i.i.d.) and (2) available before the training procedure. *distributed-continual-learning* provides a set of tools and abstractions needed to experiment with Continual Learning.

Functional Description: *distributed-continual-learning* is an open source library that extends PyTorch by providing first-class support for streams of datasets, incremental training and data parallelism leveraging distributed rehearsal buffers. This library provides supports for a wide range of continual learning scenarios, learning tasks (classification and generative problems) and rehearsal-based CL strategies.

— **Link:** https://github.com/thomas-bouvier/distributed-continual-learning
— **Size and language(s):** ∼4000 lines, Python.
— **License:** GNU General Public License v3.0

1.5.2 Contribution to Existing Software (via Pull Requests)

Spack [31]

Scientific Description: The escalating complexity of HPC libraries poses significant challenges for users of HPC clusters. The intricate dependencies of scientific applications, which necessitate specific versions of compilers, implementations of standards like Message Passing Interface (MPI), platforms like Compute Unified Device Architecture (CUDA), and other dependency libraries, make the adoption of a single, standardized software stack impractical. Furthermore, the combinatorial size of the configuration space exacerbates the difficulty of managing multiple configurations. In response to these challenges, Spack has emerged as a critical

tool designed to automate the management of such HPC environments. By providing a flexible and extensible framework for managing software dependencies and configurations, Spack enhances the productivity of scientific research.

Functional Description: Spack provides a recursive specification syntax to invoke parametric builds of packages and dependencies. It allows any number of build outputs to coexist on the same system, and it ensures that installed packages can locate their dependencies regardless of the environment. Spack leverages declarative package directives (named *specs*) to express all the intersecting and disjoint compatibilities that one version of a package has with another. Spack also leverages the concept of *variants*, or key-value data, which is declared in the package recipe, and along with versions can also be configured by the end user via command-line parameters.

Our Contribution: Package recipes allow Spack to encapsulate the build logic for different versions, compilers, build options, platforms, GPU microarchitectures, and dependency combinations in the same place. Essentially, a recipe translates a spec into build logic executed by the package's build system. Variants often correspond directly to flags piped into it. I wrote recipes for many packages whose *Neomem* and *distributed-continual-learning* depend on, for them to be installable on supercomputers using Spack. In particular: `py-nvidia-dali`, `py-continuum`, `nvtx`, `py-imagehash`, `thrift`, `py-datasets`, `fribidi`, `py-wandb`, `py-e2clab`, `py-codecarbon`. I also fixed issues in the following recipes: `arrow`, `py-arrow`, `py-horovod`, `scipy`, `py-pytest-mock`, `py-dm-tree`.

— **Link:** https://github.com/spack/spack
— **Size and language(s):** the volume of my contributions to the existing codebase is ∼800 lines, Python.
— **License:** Apache 2.0

Horovod [32]

Scientific Description: Training deep learning models requires large amounts of computation, often provided by GPUs in specialized clusters. In order to leverage many GPUs to speed up the training procedure, Horovod leverages the principles of MPI to efficiently distribute gradients. Some of these primitives are `size`, `rank`, `local rank`, `allreduce`, `allgather`, `broadcast`, and `alltoall`. Depending on the techniques employed, this communication may entail anywhere from negligible to significant overhead : thus, Horovod employs efficient inter-GPU communication via ring reduction. In addition, Horovod supports the NVIDIA Collective Communications Library (NCCL) to further optimize the communication between GPUs, enabling it to achieve good scaling efficiency on large-scale distributed training tasks.

Functional Description: Horovod is designed to take a single-GPU training script and run it on multiple GPUs or even multiple machines in a data-parallel fashion, only requiring a few lines of modification to user code. Horovod integrates seamlessly with popular deep learning

frameworks such as TensorFlow or PyTorch, enabling efficient inter-GPU communication via ring reduction.

Our Contribution: I fixed some bugs in the original codebase, notably related to the support of C++17, needed to compile against PyTorch >= 2.1. Over the course of my three-year thesis, I observed a decline in Uber's contributions to the Horovod project. This trend continued to the point where the project has not been maintained in the last year. The community is now using the distributed training solutions packaged in PyTorch, which are now able to take advantage of HPC hardware. In order to keep the same codebase for all my PhD work, I forked the original project to maintain it myself under the name Khorovod.

— **Link:** https://github.com/khorovod-ai/khorovod
— **Size and language(s):** ∼100 lines, Python/C++.
— **License:** Apache 2.0

E2Clab [33]

Scientific Description: E2Clab is a framework designed to enable researchers to replicate application behavior in a controlled setting. Its primary goal is to analyze the end-to-end performance of applications by linking results to specific parameter configurations. E2Clab proposes a methodology for addressing questions such as: *How can infrastructure bottlenecks be identified? Which system parameters and infrastructure settings influence performance, and to what extent?*

Functional Description: E2Clab presents high-level features that enable: (1) experiments ensuring repeatability, replicability, and reproducibility of research; (2) mapping of application components (edge, fog, and cloud/HPC) to the physical testbed; (3) variation & scaling of experimental scenarios; and (4) experiment management for deployment, execution, and monitoring on platforms such as Grid'5000, Chameleon, and FIT IoT LAB.

Our Contribution: I added the concept of *application configurations* to initiate deployments based on custom configurations, each encapsulating specific hyperparameters. This feature allows for faster experimentation by re-using already-tested configurations, and integrates seamlessly with *distributed-continual-learning*. Cherry-picked hyperparameters can be overridden via the command line.

— **Link:** https://gitlab.inria.fr/E2Clab/e2clab
— **Size and language(s):** ∼3000 lines, Python
— **License:** GNU General Public License v3.0

1.6 Organization of this Dissertation

This dissertation is organized into eight chapters.

This Chapter summarized the objectives of our research, as well as the resulting contributions and publications.

Chapter 2 presents the context of our research. It introduces the relevance of Continual Learning to decrease the latency between the data acquisition and the availability of insights from learning models. Then, it discusses the open challenges related to (1) learning from non-stationary data, an issue causing a bias referred to as catastrophic forgetting in the CL community; (2) scaling DL training workloads while retaining a good statistical generalization and resource utilization; and (3) scaling CL training workloads considering the two aforementioned constraints.

Chapter 3 defines the concept of rehearsal buffers to address catastrophic forgetting in continual learning, and introduces **extensions to leverage them for data-parallel training**.

In Chapter 4, we introduce key design principles such as **asynchronous techniques to hide the overhead of managing rehearsal buffers** and to enable a full spectrum of combinations for minibatch augmentations. We achieve this by sampling the rehearsal buffers of DL model replicas residing on remote nodes using low-overhead, RDMA-aware, all-to-all communication patterns. We implement a **distributed rehearsal buffer prototype that we integrated with PyTorch**, a popular AI runtime.

In Chapter 5, we study the trade-offs introduced by large-scale CL in terms of training time, accuracy and memory usage using the ImageNet benchmark. We report on extensive experiments using **128 GPUs of the Argonne National Laboratory's ThetaGPU supercomputer** for the case of classification tasks. To this end, we showcase three different models (ResNet-50, ResNet-18, GhostNet-50) and four tasks derived from the ImageNet-1K dataset. Results show our approach can improve the top-5 evaluation accuracy from 23.1% to 80.55% compared with incremental training, with just a small runtime increase.

In Chapter 6, we propose a **generalization of rehearsal buffers** capable of storing **heterogeneous data**, and serve it in the form of **annotated tuples** of tensors. This extension allows to expose representative samples alongside to their associated states to the AI runtime during training, which addresses the need to support more advanced rehearsal strategies. We then discuss how the distributed rehearsal buffer can be used to enable CL tasks using generative models on multiple GPUs.

In Chapter 7, we motivate the benefits of continual learning for scientific applications leveraging streams of data, illustrating them in the context of ptychographic reconstruction [23] with generative DL models. We report on experiments showcasing results obtained using the PtychoNN [24] model for two different types of rehearsal (simple random selection and Dark Experience Replay), which compare favorably to CL based on incremental training and to traditional reconstruction based on algorithms that are computationally intensive.

Finally, Chapter 8 concludes and presents the prospects opened by our contributions.

Chapter 2

Background

Contents

2.1 Towards Integrated HPC/ML Ecosystems 26

2.1.1 Definitions and Landscape 26

2.1.2 Patterns for Integration of Instruments and Computing 28

2.2 Continual Learning 29

2.2.1 Supervised Learning 29

2.2.2 Deep Neural Networks 30

2.2.3 Catastrophic Forgetting 37

2.2.4 Addressing Catastrophic Forgetting 39

2.3 Parallel and Distributed Deep Learning 41

2.3.1 Single/Multi Machine Parallelism 41

2.3.2 Parallel Algorithms for Deep Learning 42

2.4 Challenges Considered in this Work 47

2.4.1 Mitigating Catastrophic Forgetting through Continual Learning at Scale 47

2.4.2 The Efficiency Tradeoff: Generalization Gap vs. Compute Efficiency . . . 48

During the past decade, Machine Learning (ML) supported the shift from rule-based systems towards statistical models. Deep Neural Networks (DNNs) revolutionized how we address problems in a wide range of applications by extracting patterns from complex yet labeled datasets. In the same way that more-powerful computers made it possible to design networks with vastly more neurons, ever-growing volumes of data act as a driving force for advancements in this field. Bigger models and larger datasets demand for parallel & distributed strategies to leverage multiple compute nodes.

Most existing supervised learning algorithms operate under the assumptions that (1) training data samples are independent and identically distributed (i.i.d.), a common assumption in ML; and (2) available before the training process. However, these constraints exclude many real-life scenarios where the aforementioned datasets are replaced by high volume, high velocity data streams generated over time by distributed (sometimes geographically) devices. It is unfeasible to keep training the models in an offline fashion from scratch every time new data arrives, as this would lead to prohibitive time and/or resource constraints. Also, typical DNNs

suffer from catastrophic forgetting in this context, a phenomenon causing them to reinforce new patterns at the expense of previously acquired knowledge (i.e., a bias towards new samples). The problem of Continual Learning (CL) remains an open research question.

In this dissertation, we are interested in instantiating 1) parallel and distributed Deep Learning and 2) Continual Learning together.

This chapter reviews the concepts used in the rest of the dissertation. First, we describe how the ML and High-Performance Computing (HPC) ecosystems are converging, and identify the general trends arising from this. Next, we introduce important concepts about DNNs. We then focus on the application of deep learning to data whose distribution evolves continuously over time, a typical case when ingesting data streams. Next, we explore parallelization strategies applied to usual deep learning workloads. Finally, we discuss about the open challenges brought by CL methods when leveraging of data parallelization across nodes, which is one of the main techniques to achieve training scalability on HPC systems.

2.1 Towards Integrated HPC/ML Ecosystems

Historically, the fields of HPC and ML have evolved in isolation. For decades, the HPC community has focused on performance optimization, measurement, and reproducibility, emphasizing on both results and performance. On the other hand, the ML community did not prioritize performance, focusing instead on improving the accuracy of models. However, as ML models become more complex and are deployed in time-sensitive applications, the need for performance optimization is becoming more important. This has led to a growing interest in the intersection of HPC and ML, leading to the emergence of machine learning tracks at traditional HPC conferences and systems tracks at traditional ML conferences.

2.1.1 Definitions and Landscape

Modern Big Data analytics in science relies on three fundamental components: ML for adaptable analysis, HPC for managing large volumes of data and/or executing computationally intensive simulations, and workflow technologies for ensuring reproducibility across experiments. Despite the importance of these components, they have traditionally been studied separately, resulting in integration challenges. However, there is a growing trend towards convergence, with the interplay between HPC and ML driving advancements in both fields [35]. The integration of ML into scientific workflows introduces new requirements for HPC architectures, such as support for GPUs (Graphical Processing Units) and low-precision floating-point math.

A possible definition of an HPC/ML integrated workflow is a structured sequence of automated tasks that are executed in a specific order to achieve a particular goal, with at least one HPC task (i.e., workflow management, data preprocessing and post-processing, simula-

tion, visualization), and one ML task (i.e., surrogate modeling, hyperparameter tuning, model selection). It typically involves the use of HPC systems to train and deploy ML models, even if the task may be limited to running inferences on an already-trained model. As introduced in [36, 37], ML and HPC can be coupled in various modes:

— *ML in HPC* denotes scenarios where an ML model replaces a computationally intensive HPC component or the entire HPC simulation within a workflow. In this setting, only the ML inference phase of the model is integrated into the workflow, enabling faster insights by substituting costly simulations. However, scientists must continuously asses the ML model's performance to ensure its predictions remain accurate. If the model's accuracy degrades significantly, it should be retrained in an offline fashion, separately from the workflow.

— *ML out HPC* refers to scenarios where ML is used to steer HPC components or generate new data in parallel. This approach is getting common in modern scientific applications that require simultaneous training and inferences on ML models adapting to highly dynamic patterns. Rapid updates to the ML model in response to new training data are crucial in such applications. The ML model thus resides *outside* the primary HPC simulation but is typically trained in parallel, allowing for computational resource savings if the cost of training and inferences is lower than traditional computations.

— *ML for HPC* describes scenarios where the ML model is tightly integrated with the primary HPC task to enable system-wide optimizations. For instance, ML models can optimize the performance of runtime systems, resource managers, workflow managers, or even schedulers. In this context, the HPC component's results are used to train the ML component in real-time, fostering a tightly-coupled relationship between the two.

As noted by the same authors [37], an increasing number of scientific applications rely on dynamic, real-time interactions that require to accommodate data acquired in real-time. Such interactions open new perspectives for real-time steering of computations, reduced latency between data acquisition and generation of subsequent insights, dynamic workflow reconfiguration, or re-calibration of parameters at runtime. This marks a paradigm shift away from conventional batch-oriented systems, which is reflected in the way DL models are trained. When learning in an offline fashion (as typically done), the input data is a fixed dataset with the assumption that it encompasses all potential variations of the training data. However, learning from data generated continuously requires seamless adaptation to distribution shifts, as the model's performance can degrade significantly if it is not able to adjust to the changing data distribution in real-time. Distribution shifts occur when the underlying probability distribution of the training data changes, often due to factors such as non-stationarity, concept drift [38], or covariate shift [39]. These changes can render the original training data less representative of the current data distribution, making the model's assumptions and learned representations

outdated or even invalid. Therefore, it is crucial to have mechanisms in place that can detect and adapt to such shifts in a timely and efficient manner, in order to ensure that the model remains accurate and effective over time. We refer to such solutions as CL. This real-time learning paradigm enables the incorporation of new patterns and trends over time, ensuring that the model remains accurate at any time during the training.

Applications coupling HPC and DL do not necessarily execute all of their components on HPC systems. In fact, the landscape is changing quickly, with large centralized datasets being replaced by high-volume, high-speed data streams. In some cases, these streams originate from many geographically dispersed, loosely interconnected devices, such as mobile phones, sensors, or industrial machines. On the other hand, deep learning typically involves centralizing the data and processing it in parallel within high-performance clusters, calling for a hybrid execution infrastructure. This assembly of resource-constrained devices producing streams of input data at the edge, and Cloud or HPC systems executing DL-powered workloads providing insights about future system states, is known as the *Computing Continuum* [40].

2.1.2 Patterns for Integration of Instruments and Computing

The escalating complexity of scientific problems poses significant challenges for computer scientists. As noted in [41], the advent of instruments such as synchrotron light sources [42], telescopes [43], or microscopes [44] provide new powerful means to conduct research. By capturing data at an unprecedented level of detail, these tools allow researchers to gain novel insights into the behavior and properties of materials, biological systems, and other complex phenomena. However, the data volume generated by these instruments is massive, and powerful HPC clusters are needed to process it. A notable example is the Advanced Photon Source (APS) located at Argonne National Laboratory, which generates experimental data at a rate on the order of tens of Gbps [21] during high-fidelity X-ray imaging at microscopic level. At such generation rates, the experimental data cannot be sent fast enough over WAN (Wide Area Network) links to an HPC machine, prompting the need for pre-processing near the data acquisition in order to reduce its size. This can be achieved using compression [45], partial discarding, or feature detection via light computational stages (e.g., detecting diffraction peaks in X-ray imaging [46]). Unfortunately, such data reduction techniques imposed by limited computational capabilities of the edge infrastructure reduce the quality of end results.

To address this challenge, an emerging approach leverages previously collected experimental data (from the ongoing acquisition) to train a DL model in parallel on HPC clusters, which is then deployed to run compute-efficient live inferences at the edge. A common pattern for applying DL models in this context involves performing a classic, computationally expensive processing on the data stream sent to the HPC cluster at the beginning of the data acquisition. The output generated by this dedicated consumer workflow serves as ground-truth data for

a DL model trained in parallel. As the experiment progresses, if the DL model learns to predict ground-truth data with high fidelity, it can be deployed to the edge to replace conventional processing, thereby reducing the latency in generating insights. Although this approach incurs a higher initial overhead due to the need for additional HPC resources to train the model, it ultimately enables faster data analytics, with methods based on DL models achieving up to 1000-fold [47] speedups in X-ray imaging.

Furthermore, the DL training procedure might be continued even after switching to the inference regime, allowing for periodic updates of the model deployed at the edge to improve long-term quality of end results. Besides saving HPC resources, this approach can eventually achieve the quality obtained with the replaced algorithm in the most favorable cases. This necessitates efficient real-time updates of the DL model via continual learning, enabling the model to adapt to changing experimental conditions. Finally, such updates can be used to adjust the course of the experiment, such as **steering it in a different direction**, or calibrating the instrument through a feedback loop.

2.2 Continual Learning

Supervised learning is a fundamental paradigm in machine learning, underpinning a diverse array of big data applications. This paradigm involves the estimation of a mapping function that relates input variables to output variables, guided by labeled examples to inform the learning procedure. Deep learning is a subfield of machine learning that is closely related to supervised learning. In fact, many DL models are trained using supervised learning techniques. However, in recent years, the focus has gradually shifted towards a more dynamic and adaptive learning paradigm, known as *continual learning*. Unlike deep learning, which typically assumes a static dataset, continual learning enables models to learn from a continuous stream of data, accommodating new information while retaining previously acquired knowledge. This paradigm aligns more closely with real-world scenarios where data is often encountered sequentially and may exhibit evolving patterns over time. Thus, continual learning can be positioned as an extension of supervised learning, designed to address the challenges of non-stationary and dynamic learning environments.

2.2.1 Supervised Learning

In the context of supervised learning, a deep learning (DL) model can be used to learn the mapping function from input variables to output variables. Given a dataset D, a probability distribution $\mathcal{D}$ over the training data D and a random variable z that takes on values in the space of all possible input-output pairs (x, y), the training of a DL model is an iterative process that progressively updates its parameters w in order to predict the ground-truth value y given

Table 2.1 – Supervised Learning notation

D	dataset
$\mathcal{D}$	data probability distribution
z	random variable
(x, y)	input/label pair
$w \in \mathcal{H}$	model parameters
$w_i^{(k)}$	model parameters, i denotes parameters of operator i at iteration k
$f_w(z)$	model function (predictor)
$h_w(z)$	activations (pre-softmax responses i.e., logits)
$\mathcal{L}$	loss function
$\ell(w, z)$	per-sample loss function
$\nabla \ell(w, z)$	gradient of ℓ
$u(w, g)$	parameter update rule, function of parameters w and loss gradient g
$\vec{m}$	minibatch
b	default minibatch size
η	learning rate
I	number of training iterations (steps)

an input x. The ground truth refers to the actual or true value (e.g., a class label or id) of the data that the DL model is trained on. More formally, it can be cast as an optimization problem: we want to find w^* that minimizes the loss function $\mathcal{L}$:

$$w^* = \operatorname*{argmin}_{w \in \mathcal{H}} \mathcal{L} = \operatorname*{argmin}_{w \in \mathcal{H}} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(f_w(x), y)] \tag{2.1}$$

where $\mathcal{H}$ is the hypothesis set containing all possible combinations of parameters w, $f_w(z)$ is the prediction of the DL model with w fixed, ℓ is the loss function that estimates the error between the prediction and the ground-truth label for a single training sample, and $\mathbb{E}$ is the overall error for all pairs $z = (x, y)$.

2.2.2 Deep Neural Networks

At their core, Deep Neural Networks (DNNs) are structured layers of interconnected nodes inspired by the human brain, capable of learning intricate patterns and representations from labeled data. We now describe their building blocks by matching the concepts of supervised learning to their terminology.

Feed-forward Operator In Figure 2.1, we illustrate a DNN *single operator* (also referred to as *fully connected layer*) where two *nodes* (also referred to as *neurons*) are represented in yellow. After establishing an input layer that we represent in blue, composed of input data x, a matrix of weights w is allocated to the *connections* (also referred to as *synapses*) between layers. These

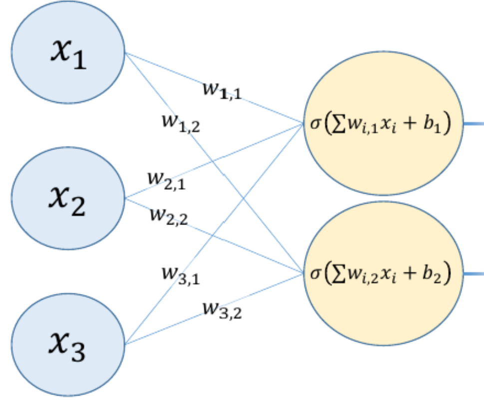


Figure 2.1 – The single operator, also called *fully connected layer*, is represented in yellow. The input layer is represented in blue. Figure borrowed from [8].

weights assess the importance of each subsequent neuron, with greater weights having a more substantial impact on the neuron's output than other inputs. Every single neuron can be considered as a distinct linear regression model. In this model, each input x_i is multiplied by its respective weight $w_{i,j}$. Additionally, a bias bi_j is assigned to each neuron and is added to the outcome:

$$\sum_{j=1}^{m}\sum_{i=1}^{n} w_{i,j}x_i + bi_j = \sum_{j=1}^{m}(w_{1,j}x_1 + w_{2,j}x_2 + ... + w_{n,j}x_n + bi_j) \tag{2.2}$$

where m is the number of neurons in the layer, n the number of input features, x_i is the i-th input feature, $w_{i,j}$ is the weight associated with the i-th input feature and the j-th neuron, and bi_j is the bias term for the j-th neuron.

The output is then processed through an activation function, which adds non-linearity to the network and determines the final output of a neuron. If this output is greater than a predetermined threshold, it triggers (or *activates*) the neuron, transmitting information to the subsequent layer in the network. Using the Heaviside step function as a naive activation function σ, each neuron can be considered as a binary classifier with an output given by:

$$\text{output} = \sigma(x) = \begin{cases} 1, & \text{if } \sum_{j=1}^{m}\sum_{i=1}^{n} w_{i,j}x_i + bi_j \geq 0 \\ 0, & \text{if } \sum_{j=1}^{m}\sum_{i=1}^{n} w_{i,j}x_i + bi_j < 0 \end{cases} \tag{2.3}$$

In practice, Rectified Linear Units (ReLU), sigmoid, softmax, hyperbolic tangents or variants [48] are used as activation functions (also referred to as *axons*). Consequently, the output of one neuron serves as the input for the next neuron in the subsequent operator (layer). This method of transferring data from one operator to the next via weighted connections character-

izes *feed-forward* neural networks.

Stochastic Gradient Descent (SGD) To solve the optimization problem formulated in Equation (2.1), the most commonly used technique is stochastic gradient descent (SGD) [49, 50, 51]. Given an initial point $w^{(0)} \in \mathcal{H}$, SGD attempts to decrease the objective J by optimizing parameters defined by the sequence $\{w^{(k)}\}_{k=0}^{I}$ as detailed in Algorithm 1.

Algorithm 1: Stochastic gradient descent (SGD)

```
1 for k = 0 to I                                          // Optimize for I iterations
2 do
3     (x, y) ← Sample random sample from D   // Obtain one sample from dataset D
4     g ← ∇ℓ(f_{w^(k)}(x), y)                            // Compute stochastic gradient
5     w^(k+1) ← w^(k) + u_SGD(g)                                 // Update parameters
```

Variable g forms an unbiased estimate of the gradient of the objective J that we call a *stochastic gradient*. The SGD algorithm updates the parameters for each training sample separately, resulting in frequent updates with high variability. This high variance can cause the convergence to be noisy.

Variants of SGD commonly used with neural networks include SGD with momentum [52, 51, 53], RMSProp [54], AdaGrad [55] and Adam [56]. All of these optimization procedures, or optimizers, interact with the training samples only by repeatedly computing stochastic gradients (line 4).

Function u (line 5) is responsible for updating the model parameters. The basic SGD update rule is $u_{SGD}(g) = -\eta \times g$, where η represents the *learning rate*. The learning rate modulates the amplitude of the stochastic gradient, controlling how much training iteration t affects the next model update $w^{(t+1)}$. Scientists typically set it to a large value at the beginning of the training, and then decay it multiple times during the process. This is empirically observed to help both optimization and generalization. As noted in [57], common beliefs in how learning rate decay works come from the optimization analysis of gradient descent: 1) an initially large learning rate accelerates training and helps the network escape spurious local minima, and 2) decaying the learning rate helps the network converge to a local minimum faster.

Backpropagation A deep neural network is constructed as a composition of operators

$L_G\left(f_{w_G}, \ldots, L_2\left(f_{w_2}, L_1\left(f_{w_1}(x)\right)\right)\right)$ for which partial derivatives are easily computed, where each function L_l is an operator and each tensor w_l represents operator l's weights. The *chain rule* is a fundamental concept in calculus that allows to compute the derivative of a composite function. In the context of deep learning, the chain rule is used to compute the gradients of the loss function with respect to the model's parameters w. Starting with an initial $w^{(0)}$ chosen randomly, a *forward pass* computes the intermediate predictions for each training sample in the

chain of composed operators to obtain a loss value $\ell(f_w(x), y)$ computed by a loss function ℓ. Then, a *backward pass* is performed, where the gradients of the loss function with respect to the model's parameters are computed using the chain rule. Specifically, the gradient of the loss function with respect to the parameters of the last operator L_G is computed first, and then the gradients of the previous operators are computed recursively, until the gradients of the first operator L_1 are obtained. Information is thus propagated through the network via intermediate gradients $\nabla_{w_l} = \frac{dL}{dw_l}$ (w.r.t. weights w_l) and $-\nabla_x$ (w.r.t. input data x). Finally, these gradients are used to update the model's parameters w using an optimization algorithm with an update proportional to the partial derivative of the loss value. This process referred to as *backpropagation* is repeated iteratively in order to come closer to a minimum of $\mathbb{E}$ defined in Equation 2.1, until some termination criteria are satisfied.

Backpropagation applied to feed-forward networks initially encountered little success for a reason identified as the *vanishing gradient* problem [58]. This issue occurs when the composition of operators lengthens, causing the intermediate gradients magnitude to decrease (or grow uncontrollably), slowing the training process [59] as a result. In the worst case, this may completely stop the DNN from further training.

DNN Training Procedure Reusing the notation introduced in [60], the procedure of training a DNN using Algorithm 1 can be divided into three main stages: (**IO**) loading and pre-processing a sample from the dataset; (**FB**) performing a forward pass where the sample passes through the DNN, followed by a backward phase (backpropagation) to compute the gradient encoding how sensitive the loss function ℓ is to each parameter w.r.t. the current sample; and (**WU**) updating the DNN parameters (weights) according to the gradient. In practice, a typical optimization taking advantage of the highly parallel architectures of modern accelerators (e.g., GPUs) is to process the gradients for small, uniformly random subsets of training samples called *minibatches*. Step (**FB**) is then computed w.r.t. the minibatch, with each minibatch containing b samples. The optimization procedure as implemented by the *minibatch stochastic gradient descent* algorithm is described in Algorithm 2.

Algorithm 2: Minibatch stochastic gradient descent (SGD)

1 **for** $k = 0$ ***to*** $\frac{|D|}{b}$ **do**

2 $\quad \vec{m} \leftarrow$ `sample(D, b)` `// Obtain a minibatch from dataset` D

3 $\quad g_m \leftarrow \frac{1}{b} \sum_{(x,y) \in \vec{m}} \nabla \ell(f_{w^{(k)}}(x), y)$ `// Compute gradient using backpropagation`

4 $\quad w^{(k+1)} \leftarrow w^{(k)} + u_{GD}(g_m)$ `// Weight update rule`

Minibatch SGD strikes a balance between *stochastic gradient descent* (SGD), which processes one training sample per iteration (as detailed in Section 2.2.2) resulting in as many training iterations as there are samples, and *batch gradient descent* (GD) which processes all training samples at once, leading to a single occurrence of the (**IO**), (**FB**), and (**WU**) steps described above. While

GD computes a single, comprehensive gradient over the entire dataset, SGD performs frequent updates at a lower computational cost, under the assumption that these noisy gradients still roughly point in the correct direction. The variable g_m represents a stochastic gradient of the objective J computed using minibatch $\vec{m}$ of input data that is assumed to be independently and identically distributed (i.i.d.). This can be expressed as:

$$g_m(w^{(k)}, \vec{m}) = \frac{1}{b} \sum_{(x,y)\in\vec{m}} \nabla\ell(f_{w^{(k)}}(x), y) \tag{2.4}$$

By determining how many samples are processed in one training iteration, the minibatch size b affects the algorithm's convergence [61]. An *epoch* refers to one complete pass through the entire training dataset, as depicted in Algorithm 2. The number of iterations (each involving the (**IO**), (**FB**), and (**WU**) steps mentioned above) per epoch is equal to the number of samples divided by the minibatch size b. Minibatches are typically sampled with replacement, following the practice of cycling through permutations of the dataset at each epoch. Sampling without replacement was proven [62] to provide similar convergence guarantees.

DNN Notation In the case of a plain DNN, the input data is typically flattened into a one-dimensional vector before being fed into the network. When a minibatch of color images serves as input for a DNN, it is represented as a four-dimensional tensor sized $b \times C \times H \times W$. Here, each image is characterized by C channels (e.g., which might be RGB components with $C = 3$), with each of these channels spanning a grid of $H \times W$ pixels. In a DNN presenting G layers, a fully connected layer as in Figure 2.1 operates on the following tensors:

- The input of layer l with a minibatch containing b samples, each sample include C_l channels, each channel is a tuple of d-dimension: $x_l[b, C_l, X_l^d]$. When working with images, we replace X_l^d with $[W_l, H_l]$ i.e., $x_l[b, C_l, W_l \times H_l]$. To simplify, we omit the layer index l and the dimension d e.g., $x[b, C, X]$.
- The output (activations) of layer l with b samples and $C_{l_{\text{out}}}$ output channels, each of shape $y_l[b, C_{l_{\text{out}}}, Y_l^d]$.
- The weights $w_l[C_l, C_{l_{\text{out}}}]$.
- The bias $bi_l[C_{l_{\text{out}}}]$.

Layer l is defined on the group of neurons x_l by $y_{l,*} = \sigma(wx_{l,*} + bi)$, where w is the weight matrix sized $C_l \times C_{l_{\text{out}}}$ and bi_l is a per-layer trainable bias vector sized $C_{l_{\text{out}}}$.

Convolutional Neural Networks (CNNs) When working with high-dimensional data such as color images, flattening the data into a one-dimensional vector to be fed into a DNN can lead to a large number of input features and a correspondingly large number of network parameters. This can lead to two issues: (1) the risk of *overfitting*, which occurs when a model is too

complex and fits the training data too closely, capturing noise and failing to generalize to new, unseen data and (2) the network can become more computationally expensive to train. Feature engineering is the process of manually transforming raw data into a format that is more easily operable by neural networks, which requires significant effort and domain expertise to design relevant features for the task at hand.

Convolutional Neural Networks (CNNs) provide a solution to the challenges of image recognition by automatically extracting meaningful features directly from the raw input data during the training procedure. This ability to learn feature representations from data makes CNNs well-suited for image recognition. As the input image propagates through the DNN, *convolutional operators* reduce the spatial dimensions (width and height) of the image, as well as the number of feature maps via *filters* (also referred to as *kernels*). Each feature map is a spatial representation of the input data, where the values in the map correspond to the presence of specific features or patterns in the input data. The number of output filters F determines the number of feature maps produced by the layer, and each filter is responsible for detecting a specific pattern or feature in the input data. This dimensionality reduction achieved by CNNs has two key benefits: (1) it enabled the capture of hierarchical patterns within the data, and (2) it mitigates the vanishing gradients issue by optimizing over fewer connections [63].

In the context of CNNs, input channels C refer to the number of input data streams that are processed by a layer. For example, the input channels might correspond to the RGB color channels of an input image. Each input channel is processed independently by the layer, and the outputs are combined to produce the final output. To formalize this concept, consider a convolutional operator that takes a 4-dimensional minibatch of images as input. Each image in the minibatch is represented as a 3D tensor x_i, which is convolved with $C_{l_{\text{out}}}$ filters of size $C_l \times K_l^d$. Here, C_l represents the number of input channels, K_l represents the spatial dimensions of the filter, and d represents the number of spatial dimensions in the tuple (e.g., width and height). We reuse and adapt the notation introduced in [60]:

— The parameters $w_l[C_l, F_l, K_l^d]$ with F_l filters. Each filter sized K_l^d has C_l channels. To simplify, we omit the kernel size e.g., $w_l[C_l, F_l]$.

— The output (activations) of layer l with b samples and F_l output filters $y_l[b, F_l, Y_l^d]$. Each output filter is sized Y_l^d.

— The bias $bi_l[F_l]$.

— The input gradients $\frac{dL}{dx_l}[b, C_l, X_l^d]$.

— The parameter gradients $\frac{dL}{dw_l}[C_l, F_l, K_l^d]$.

— The activation gradients $\frac{dL}{dy_l}[b, F_l, Y_l^d]$.

To set the scene for future sections, we use the notation from [60] to describe steps (**IO**), (**FB**) and (**WU**) in further detail when performed in the context of CNNs. The following steps are performed for each operator l:

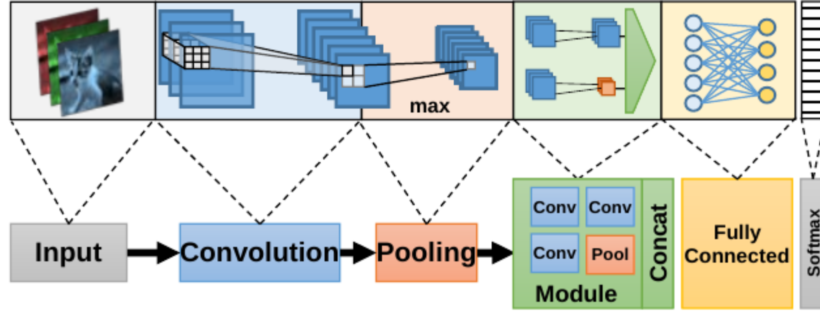


Figure 2.2 – A DNN is composed of many operators. Figure borrowed from [8].

$$(\mathbf{IO}) \quad x[b,C,X] \leftarrow IO(D,b) \text{ in the first layer} \tag{2.5}$$

$$(\mathbf{FB}) \quad y[b,F,Y] \leftarrow FW(x[b,C,X], w[C,F,K]) \tag{2.6}$$

$$(\mathbf{FB}) \quad \frac{dL}{dx}[b,C,X] \leftarrow BW_{\text{data}}\left(\frac{dL}{dy}[b,F,Y], w[C,F,K]\right) \tag{2.7}$$

$$(\mathbf{FB}) \quad \frac{dL}{dw}[C,F,K] \leftarrow BW_{\text{parameters}}\left(\frac{dL}{dy}[b,F,Y], x[b,C,X]\right) \tag{2.8}$$

Parameters of all layers are updated at the end of every training iteration using the learning rate α:

$$(\mathbf{WU}) \quad w[C,F,K] \leftarrow WU\left(\frac{dL}{dw}[C,F,K], \alpha\right) \tag{2.9}$$

Another important concept of CNNs is *pooling*, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling, where max pooling is the most common. The equations above apply.

To increase the likelihood of the cheaper minibatch gradients pointing in the same direction, this training procedure revisits dataset D repeatedly over multiple *epochs*. In practice, the training phase typically consists of 40-300 epochs. Each of these epochs represents a pass over the entire training data, which in turn is randomly shuffled to guarantee the training samples are seen in a different order at different epochs.

2.2.3 Catastrophic Forgetting

Although efficient on static training data, optimization algorithms like SGD and minibatch GD do not perform well when training data arrives continuously over time. One approach to continual learning is to train the DL model incrementally (i.e., the training proceeds with relatively inexpensive updates to the model's parameters based on just the new data samples). If data increments are small, such an approach achieves high performance and low resource utilization. Unfortunately, it can also cause the accuracy of the DL model to deteriorate quickly—a phenomenon known as *catastrophic forgetting* [14]. This naive approach to continual learning introduces a bias in favor of new samples, effectively causing the model to reinforce recent patterns at the expense of previously acquired knowledge. Specifically, if a new training dataset D' is available in addition to D, but we sample new minibatches only from D', then our DL model will drift in the direction of minimizing $\mathbb{E}_{\mathcal{D}'}$ (the overall error corresponding to D'), which may no longer be representative of $\mathbb{E}_{\mathcal{D}+\mathcal{D}'}$. Larger differences between the distributions of the old vs. new training data exacerbates the bias, often to the point where a single pass over the new training data is enough to erase most, if not all, of the patterns learned previously.

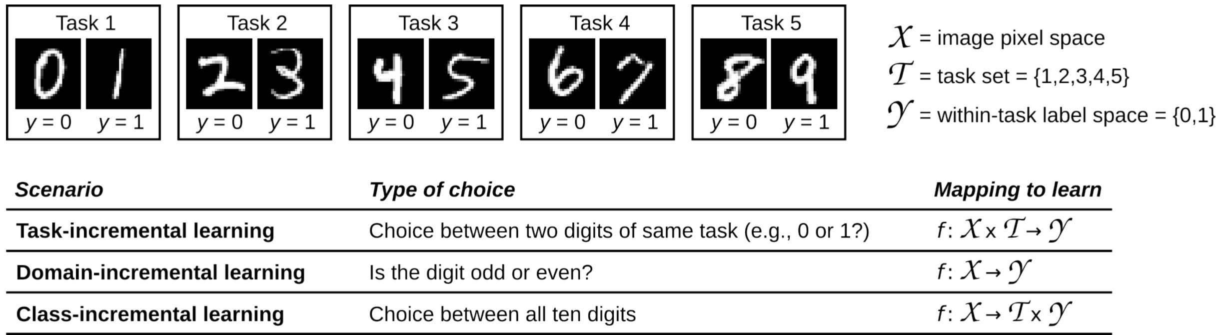
In the context of continual learning, we call each new training dataset D' a new *task* (sometimes referred to as *data increments*). Given T tasks and their probability distributions of data $\mathcal{D}_t$, the optimization problem defined in Equation 2.1 becomes:

$$w^* = \underset{w \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{T} \sum_{t=1}^{T} \mathcal{L}_t, \quad \text{where} \quad \mathcal{L}_t \triangleq \mathbb{E}_{(x,y)\sim\mathcal{D}_t}[\ell(f_w(x), y)] \tag{2.10}$$

Catastrophic forgetting echoes the more general plasticity-stability dilemma [64], where (1) plasticity refers to the ability of the model to learn concepts in the current task, and (2) stability refers to its ability to preserve knowledge acquired in previous tasks.

In continual learning, the model is trained on a sequence of tasks $\mathcal{T} = (T_1, \ldots, T_t)$, where $T_i = \{(x_i, y_i)\}$ denotes the input-output tuples attached to task T_i. The goal is to perform sequential training while preserving knowledge gained in previous tasks. Authors in [65] differentiate three CL scenarios "based on whether, at test time, task identity is provided and, if not, whether task identity must be inferred". We illustrate them in Figure 2.3 using the split-MNIST [66] dataset as example.

Task-incremental Scenario A popular CL setting is the task-incremental ("Task-IL") scenario, in which the output space $\mathcal{Y}$ is fixed (e.g., a fixed number of classes). In this setting, the model is aware of the task identity when running inferences (e.g., the task id is provided or obvious to infer). This configuration makes the mitigation of catastrophic forgetting trivial, as the model can use a specific component devised for the task at hand. A high-level example would be to imagine tasks corresponding to learning musical instruments. At inference time, to recognize



Scenario	***Type of choice***	***Mapping to learn***
Task-incremental learning	Choice between two digits of same task (e.g., 0 or 1?)	$f: \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{Y}$
Domain-incremental learning	Is the digit odd or even?	$f: \mathcal{X} \rightarrow \mathcal{Y}$
Class-incremental learning	Choice between all ten digits	$f: \mathcal{X} \rightarrow \mathcal{T} \times \mathcal{Y}$

Figure 2.3 – The split-MNIST dataset according to each of the three continual learning scenarios. With Task-IL, the choice for the model is between two digits from a given task (e.g., zero or one). With Domain-IL, the choice is still between two options (e.g., even or odd), but the problem is more difficult as task identity is not provided at inference time. With Class-IL, the model must choose between all ten digits, and the output space is growing as new tasks are observed. Borrowed from [67], where it was modified from [68].

a chord being played, the model is first given the instrument that produced the sound. Consequently, it can activate the relevant task-specific component to use the right knowledge. For this reason, the task-incremental scenario is sometimes referred to as *multi-headed*.

Domain-incremental Scenario Domain-incremental ("Domain-IL") learning refers to a scenario where the underlying problem structure remains the same, but the context varies over time, resulting in domain shifts. As with Task-IL, the output space $\mathcal{Y}$ is fixed. However, it is not provided which domain a sample belongs to at inference time. As a result, using a model with components specific to each domain is not feasible in this setting. To return to the illustration using musical instruments, a model would have to recognize the chord without being aware of which instrument produced it. The model may or may not first infer the domain.

Class-incremental Scenario A more complex CL problem is the class-incremental ("Class-IL") scenario, which is the problem of incrementally learning to discriminate between an increasing number of classes. In this setting, the output space $\mathcal{T} \times \mathcal{Y}$ changes from one task to another (e.g., new images of a new classes not encountered in previous tasks). In other words, the model is expected to discriminate between classes originating from different tasks, unlike previous scenarios, where the model had to discriminate within a task or domain. To take the example of musical instruments again, a model must recognize the musical note and also the musical instrument that emitted it. New instruments are learned over time. A challenging aspect of Class-IL, sometimes referred to as *single-headed*, is to discriminate between classes that were not observed together.

2.2.4 Addressing Catastrophic Forgetting

In this section, we focus on approaches that have been proposed in the literature to address the catastrophic forgetting problem. We rely on the categorization carried out in [67].

Experience Replay Experience Replay, that we refer to as *rehearsal* in this dissertation, is a simple continual learning technique in which the model knowledge is reinforced by replaying samples from previous tasks [69, 34]. This idea stems from neuroscience, where the re-occurrence of neuronal activity patterns that represent previous experiences is believed to play a crucial role in the consolidation of new memories in the brain [70, 71].

In practice, this approach selectively stores previously encountered raw data samples, called *representatives* (sometimes referred to as *exemplars*), into a *rehearsal buffer* denoted $\mathcal{B}$. Representatives are then sampled back from this buffer to *augment* the minibatches of new training tasks. Such augmentations involve appending a fixed number of representatives to each minibatch corresponding to the most recent training data, in order to obtain a large minibatch that mixes new and old training samples. When learning the current task t_c, one seek to minimize the following objective preserving the knowledge acquired on previous tasks $\{1, \ldots, t_c - 1\}$, $t_c \leq T$:

$$\mathcal{L}_{t_c} + \mathbb{E}_{(x,y)\sim\mathcal{B}}[\ell(f_w(x), y)] \tag{2.11}$$

The advantage of the Experience Replay approach is that it can mitigate catastrophic forgetting transparently [72, 73], without the need to change existing model architectures or training methods. This claim is supported by studies that not only emphasize its effectiveness compared to alternative methods [74], but also propose diverse extensions to enhance its performance [75]. Most of these optimizations are related to how representatives are selected and stored into the rehearsal buffer. As such, the sampling strategies for representatives might rely on their associated gradients [76], training loss [75], as well as alternative sampling policies from the buffer [77].

Another approach to rehearsal, sometimes referred to as *pseudo-rehearsal*, learns the input distribution using a generative model. Generative replay leverages DL models such as GANs to construct synthetic data that mimics past training samples [78, 79]. The effectiveness of this approach highly depends on the capability of generative models to deal with complex datasets [80]. This limitation might be alleviated by replaying latent features instead of raw training samples [81], although this approach requires a pretraining phase to achieve satisfactory results. Besides, generative replay requires higher computational overheads compared with sampling.

A limitation of rehearsal, as noted by authors in [82], is its tendency to prevent the model from exploring beyond the first encountered low-loss region. Their observations indicate that this behavior can lead to overfitting, where the model becomes too specialized towards the edges

of the rehearsal buffer's low-loss region, ultimately impairing its ability to generalize effectively. We believe this effect can be mitigated by increasing the buffer capacity. Another concern with rehearsal comes from the additional computational cost, proportional to the number of representatives to be replayed during the training procedure. Fortunately, research suggests that it is not essential to replay representatives from all previous tasks each time a new task is learned. As suggested in [83], acquiring new knowledge is generally more challenging than preserving it once it has been learned. Consequently, replaying subsets of representatives often suffices to limit forgetting.

Regularization Parameter regularization applies constraints to the parameter update rule to prevent forgetting knowledge acquired on previous tasks. Many authors propose adding penalty terms to the loss function in order to adjust the learning objective [84, 66, 85], mitigating large changes to parameters that would degrade the performance on previous tasks. A crucial aspect of this approach is to estimate how important parameters are for previously learned tasks. The Fisher information matrix can provide such an estimate [84]. Overall, most methods based on parameter regularization can not learn a correct solution in class-incremental scenarios (i.e., without knowing the task identifier at inference [86, 87]).

Functional regularization is a similar approach, with large changes to parameters being prevented with respect to a set of specific inputs called *anchors*. For instance, the Hindsight Anchor Learning (HAL) [88] algorithm complements Experience Replay with regularization to align the model responses with such data points encoding classes encountered in previous tasks. This approach bears similarity with meta-learning approaches [89], which aim to maximize transfer learning capabilities. Other studies leverage the *knowledge distillation* loss [90] to regularize over the feature representations of previous tasks. However, this approach tends to lead to a *representation drift* [15] when the number of tasks is large (i.e., as parameters adapt to new tasks the values that other parameters are constrained towards become obsolete). Both Dark Experience Replay (DER) and DER++ algorithms [25] demonstrate that injecting a distillation term (obtained from activations of a previous version of the model) into the loss calculation instead of replaying past representatives (or doing both) yields to a better achieved accuracy than rehearsal alone. A close proposal is Function Distance Regularization (FDR) [91], with the limitation of storing activations at task boundaries only. Besides, X-DER [92] takes an extra step over previous methods by preparing future classification heads to accommodate future classes.

As discussed in [67], the line between rehearsal and functional regularization is blur, as the latter can sometimes be seen as a variant of the former. In this light, the replayed data consists in past anchors labeled with the corresponding predictions, as made by a previous version of the model [78]. In the case of regularization, inputs and associated predictions can be stored in an external rehearsal buffer, but also internally in the model itself (or in a model checkpoint).

Optimization-based strategies Instead of modifying the loss function, optimization-based strategies leverage novel optimization routines more suitable to continual learning scenarios instead of the SGD variants mentioned in Section 2.2.2. For instance, some studies explore the use of gradient projection [93] to restrict parameter updates to directions that avoid erasing knowledge from previous tasks. Some studies explore the use of adaptive learning rates, with modulation depending on the importance of the parameters for the previous tasks [94, 95]. Another strategy is to control the optimization trajectory is through probabilistic parameter updates [96].

Template-based strategies Another approach to continual learning is template-based classification. In this approach, a class template is learned for each class, and classification is performed based on which class template is closest for the sample to be classified. Templates can be defined in an embedding space [97] using an embedding model to convert training samples. When embeddings need to be updated as new data arrives, multiple embeddings can be stored per class, as proposed in iCaRL [18]. Generative classification [98] is a similar strategy in which templates are learned by generative models. However, since such models are often limited by their ability to learn complex datasets, an alternative is to train an energy-based model and compute energy values per class [99].

2.3 Parallel and Distributed Deep Learning

If training a model on a single process is too slow or if the model's weights exceed the memory capacity of a single process, transitioning to multiple distributed compute processes becomes necessary. Before making this transition, some techniques should be explored for efficient training on a single GPU, as they are universally applicable to model training on any number of GPUs. Transitioning from a single GPU to many GPUs requires introducing some parallelism, as the workload must be distributed across the compute resources. The procedure of training a DNN in a distributed fashion can be divided into four main stages. Building upon the notation introduced in Section 5: (**IO**) data loading and pre-processing, (**FB**) a forward phase where training samples pass through the DNN, followed by a backward phase (backpropagation) to compute the corresponding gradients, (**GE**) the gradient exchange across distributed training processes and (**WU**) updating the DNN parameters.

2.3.1 Single/Multi Machine Parallelism

Techniques for implementing parallel learning algorithms range from simple threaded implementations to using OpenMP [100] on individual machines, leveraging shared memory for

efficient parallelization. Accelerators such as GPUs or FPGAs (Field-Programmable Gate Arrays) typically require programming with specialized languages like CUDA, OpenCL, or hardware design languages. However many applications require the same low level mathematical routines. Libraries dedicated to accelerating these common routines allow to easily make full use of the available hardware without requiring low level knowledge of the hardware themselves. Such libraries are often provided by manufacturers for specific platforms like cuDNN [101] (NVIDIA), or MKL-DNN (Intel). SYCL-DNN is another hardware-agnostic library dedicated to providing accelerated routines for neural network operations.

In scenarios where multiple machines with distributed memory are involved, one can select between communication mechanisms like TCP/IP or RDMA (Remote Direct Memory Access [102]). This technology adds hardware support to allow a process hosted on one compute node to access the memory of another process hosted on a different compute nodes over the network without involving the CPU. Using this approach, the latency of small accesses can be significantly improved, which enables our approach to be efficient. Distributed memory machines also allow for the use of more convenient libraries like the Message Passing Interface (MPI) or Apache Spark [103]. MPI leverages *collective communication primitives*, such as `broadcast`, `reduce`, `scatter`, `reduce_scatter`, `gather`, and `allreduce`, to enable efficient communication and synchronization between processes running on different compute nodes. In contrast, Spark uses a different approach to achieve parallelism, based on the concept of Resilient Distributed Datasets (RDDs). RDDs are immutable distributed collections of data that can be transformed and processed in parallel using a set of high-level operators, such as `map`, `filter`, `reduce`, and `join`.

2.3.2 Parallel Algorithms for Deep Learning

Training DL models is computationally expensive, e.g. training ResNet-50 [104] over a single V100 GPU requires about 30 hours. Hence distributed training on HPC systems is common for large models and datasets. The distributed minibatch algorithm [105] provides a general parallelization scheme applicable across various machine learning algorithms. The fundamental principle underlying this approach is the following: within any gradient-based ML algorithm, such as minibatch SGD as presented in Algorithm 2, local gradients are computed independently and subsequently sent to a coordinator node, or reduced via a collective communication primitive. When using a coordinator, e.g., a *parameter server*, the latter aggregates these gradients, thereby obtaining a comprehensive representation of the global gradient with respect to all local data points, and executes a singular update step. The updated model is then disseminated to the local processes, which proceed to compute subsequent gradients. Extensive theoretical analyses have validated this approach, and empirical evidence attests to its efficacy in practical settings. Notably, the scalability of this methodology is often characterized as *embarrassingly*

parallel [106] (training processes proceed concurrently), highlighting its simplicity and effectiveness.

In this section, we discuss the main parallelization techniques for training neural networks, differing in the way the data or model dimensions are split: (1) distributing the training data among processes (*data parallelism*), (2) vertically partitioning the DL model along its depth (*layer parallelism*), allowing the overlap of computations between one layer and the next layer (*pipeline parallelism*), (3) horizontally partitioning the DL model (*model parallelism*). We also mention extensions to these approaches and existing implementations.

Data Parallelism In minibatch SGD (detailed in Algorithm 2), the input data is processed in minibatches containing b samples. Most operators in DNNs and CNNs are independent with respect to b, allowing to partition the computation along the data dimension. This idea was first formalized in [105] as the *distributed minibatch algorithm*. In this context, the *local minibatch size*, denoted as b, refers to the number of data samples processed by each process for a given training iteration, while the *effective batch size*, defined as $B = p \times b$, represents the ”true“ batch size in terms of its effect on model convergence, taking into account the parallelization across all p processes. In other words, the local minibatch size is the value seen by any individual training process, while the effective batch size is the global batch size resulting from their summation.

Data parallelism leverages minibatch SGD by creating multiple DL model replicas among the p training processes, each of which is trained in parallel on a different shard (i.e., partition) of the training data. The forward and backward passes can then proceed independently, except that after each backward pass, the gradients computed in Equation 2.4 by all replicas are averaged across training processes before adjusting parameters w. This approach ensures that the DL model replicas always apply the same updates on w and are thus in sync (assuming they started from the same initial $w^{(0)}$).

In controlled environments leveraging high-performance interconnects, data parallelism can be implemented using *collective communication primitives* among all participating processes. Data parallelism performs a single communication step per minibatch, which involves sending gradients and aggregating them using the `allreduce` operation. This results in better resource utilization, reduced communication overhead, and overall faster training times. Data parallelism using communication collectives also facilitates the development of training scripts as the same code can be run on all processes [32], making it a preferred choice for large-scale distributed deep learning training.

Reusing the same notation as in [60], when presenting tensors such as x (input data), y (labels), and w (parameters), we use the $*$ symbol to present a dimension for which its values are replicated between all training processes. To emphasize that a tensor dimension is partitioned among different processes, we use the number of processes p. For example, in data parallelism,

$x[p,*,*]$ implies that the input data x is split equally in dimension b (number of samples, as defined in Section 4) and partitioned to p processes. The Allreduce arrow $\xleftarrow{\text{Allreduce}}$ presents `allreduce` communications. We define operations occurring on the training process n in data parallelism as:

$$\textbf{(IO)} \quad (x)_n[p,*,*] \leftarrow IO(shard_n, b) \text{ in the first layer} \tag{2.12}$$

$$\textbf{(FB)} \quad (y)_n[p,*,*] \leftarrow FW\left(x_n[p,*,*], w[*,*,*]\right) \tag{2.13}$$

$$\textbf{(FB)} \quad \left(\frac{dL}{dx}\right)_n [p,*,*] \leftarrow BW_{\text{data}}\left(\left(\frac{dL}{dy}\right)_n [p,*,*], w[*,*,*]\right) \tag{2.14}$$

$$\textbf{(FB)} \quad \left(\frac{dL}{dw}\right)_n [*,*,*] \leftarrow BW_{\text{parameters}}\left(\left(\frac{dL}{dy}\right)_n [p,*,*], (x)_n[p,*,*]\right) \tag{2.15}$$

$$\textbf{(GE)} \quad \frac{dL}{dw}[*,*,*] \xleftarrow{\text{Allreduce}} \sum_{n=1}^{p}\left((\frac{dL}{dw})_n[*,*,*]\right) \tag{2.16}$$

$$\textbf{(WU)} \quad w[*,*,*] \leftarrow WU\left(\frac{dL}{dw}[*,*,*]\right) \tag{2.17}$$

The primary challenge with data parallelism is managing large effective minibatch sizes, as they can impact the model's ability to generalize well to unseen data, as noted in [107]. Through the implementation of various adjustments to the training procedure (use SGD with momentum, gradient clipping, an initial warmup ramping up the learning rate, followed by a decreasing learning rate schedule, and adapt the learning rate with the minibatch size), more recent works have successfully managed to increase the effective batch size to 8K samples [108], 32K samples [109], and even 64K [110] without losing considerable accuracy. While the generalization issue still exists, it is not as severe as claimed in prior studies [111]. Finally, data parallelism assumes that the model parameters can fit into the memory of each processing unit. This can be a limiting factor for very large models which may require more memory than is available on a single processing unit. In such a case, other forms of parallelism should complement data parallelism.

Model-vertical Parallelism (Layer Parallelism) Despite early research on model parallelism in DL [112], these efforts remained largely confined to academic circles, as data parallelism proved sufficient for most production deployments. However, the rapid growth in model sizes soon outpaced the advancements in compute and memory capabilities of individual processing units [113]. This scalability bottleneck hindered the effectiveness of data parallelism, prompting the exploration of alternative parallelization strategies. One such strategy is model-vertical parallelism, also known as *layer parallelism*, which partitions the workload along the G layers of a DNN. In this scheme, each training process receives a copy of the same minibatch, and differ-

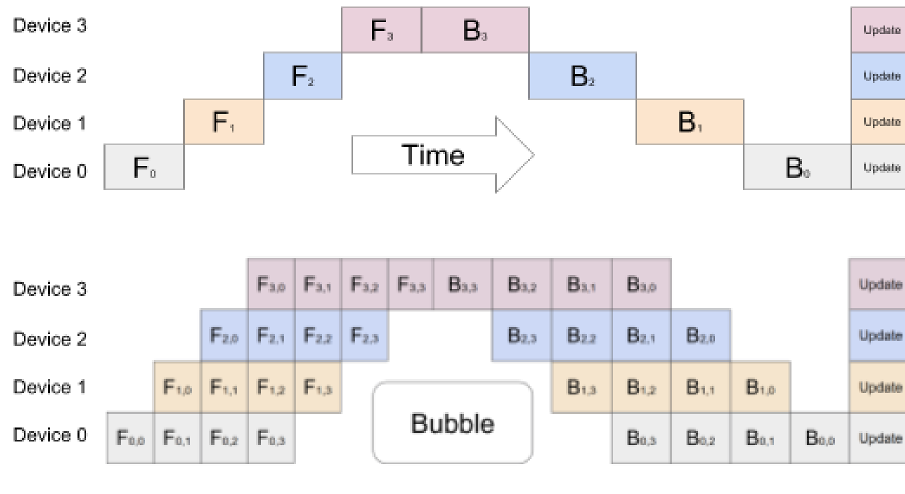


Figure 2.4 – Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one training process is active at a time. Bottom: GPipe [113] divides the input minibatch into smaller microbatches, enabling different processes to work with separate microbatches in parallel. Figure borrowed from [113].

ent layers are computed by separate training processes. The DNN architecture introduces layer inter-dependencies, resulting in complex communication patterns that determine overall training performance. A major limitation of model-vertical parallelism is that, at any given moment, all but one training process remains idle, as illustrated in Figure 2.4. The idling problem significantly hinders the compute efficiency of this approach, limiting its potential for scalability.

To address these challenges, several frameworks have been developed to facilitate the training of large-scale DNNs using model parallelism. For instance, DistBelief [112] was designed to support distributed computation in a model-parallel fashion, enabling the training of very large DNNs. Mesh-TensorFlow [114] is a language specifically designed for specifying distributed tensor computations. These frameworks also support data parallelism.

Pipeline Parallelism Pipeline parallelism addresses the idling problem by dividing the incoming minibatch into smaller microbatches, creating a pipeline that enables concurrent computation across different training processes (Figure 2.4), improving overall efficiency. The minibatch is first replicated across all processes and then divided locally into S microbatches of size $\frac{b}{S}$. During each stage, the forward computation of a layer l_i on a data segment s_i is performed simultaneously with the computation of layer l_{i+1} on data segment s_{i-1}, and so on. The backward computation is executed in reversed order.

Pipeline parallelism was first introduced in GPipe [113], which partitions both model parameters and neural activations. However, GPipe requires a minibatch size proportional to the number of pipeline partitions to hide the pipeline bubble (sum of the idle times of all the stages

in the pipeline). PipeDream [115] addresses this limitation by maintaining copies of stale parameters, decreasing the pipeline bubble size. Subsequent research efforts, such as Chimera [116], aim to reduce the number of bubbles by leveraging sophisticated scheduling of bidirectional pipelines. Some authors [117] have even eliminated bubbles entirely by splitting the activation gradient and parameter gradient in backward computation. Several recent frameworks have implemented pipeline parallelism, yielding robust and production-ready solutions. Megatron-LM [118] implements pipeline parallelism for large-scale language models. DeepSpeed [119] accelerates DL training by leveraging pipeline parallelism and other optimization techniques. SageMaker [120], is a managed service for training and deploying DL models, including support for pipeline parallelism.

Model-horizontal Parallelism (Tensor Parallelism) Model-horizontal parallelism, also referred to as *tensor parallelism*, involves partitioning individual layers of the model across multiple training processes. Specifically, each layer is divided equally among the number of output filters [121] F or input channels C and distributed across p processes. Each process retains a subset of the parameters for a given layer and performs partial computations of the output in the forward and backward phases. With filter parallelism, each process n retains $\frac{F}{p}$ filters and generates a local output denoted $(y)_n$ of size $b \times \frac{F}{p} \times |Y|$. Following the forward computation of each layer, training processes must share their local output via an `allgather` operation, resulting in the global output $y = \bigcup_{n=1}^{p} (y)_n$. Similarly, after completing the backward computation of each layer, training processes must share their gradient of the input via an `allreduce` operation, yielding the global gradient $\frac{dL}{dx} = \sum_{n=1}^{p} \left(\frac{dL}{dx}\right)_n$.

However, tensor parallelism has several limitations. The need for intricate communication during backpropagation at every layer can introduce a significant overhead. Furthermore, unlike data parallelism, filter and channel parallelism necessitate multiple collective communication rounds at each layer, and every individual minibatch must be copied across all processes. Recent implementations of tensor parallelism include Megatron-LM [118] and SageMaker [120].

Hybrid Parallelism (3D Parallelism) Data, model, and pipeline parallelism each perform a specific role in improving memory and compute efficiency when training DNNs. These approaches are complementary and can be combined together to form *3D parallelism*. The model's layers are divided into pipeline stages stored on different accelerators, with each stage further divided through model parallelism, creating a 2D combination that significantly reduces the memory consumption of the model, optimizer, and activations. However, the model cannot be infinitely partitioned due to increased communication overheads, which can hinder compute efficiency.

ZeRO-powered Data parallelism In standard data-parallel training, every process works on a different minibatch and gradients are summed using an `allreduce` operation (involving a communication volume of $2 \times p$). While data parallelism has become very popular, it takes more GPU memory than necessary because model parameters and optimizer states are replicated across all p processes. The Zero Redundancy Optimizer [122, 123] (ZeRO) is a memory optimization for data parallelism, borrowing ideas from model parallelism at the same time. This extension removes the memory redundancies by partitioning model states (parameters w, gradients, and optimizer state) across data-parallel processes. When training, a dynamic communication schedule is used to share the necessary states across distributed training processes, improving memory efficiency with, at most, a total communication volume of $3 \times p$.

Focus of this Work The implementation of data parallelism is straightforward as it does not require any modification to the DL model itself. Instead, the primary impact is on the data pipeline used to feed the training data into the model. The simplicity of this approach allowed it to gain widespread adoption in the community, as shown by its integration into various frameworks such as PyTorch [124], TensorFlow [125] and Horovod [32]. For the same reason, we focus on this form of parallelism in our work. Therefore, we have to deal with the constraint of selecting DL models that fit the memory of a single processing unit.

2.4 Challenges Considered in this Work

In this dissertation, we are interested in how CL methods can take advantage of data parallelization across training processes, which is one of the main techniques to achieve training scalability on HPC systems.

2.4.1 Mitigating Catastrophic Forgetting through Continual Learning at Scale

Scaling Continual Learning (CL) workloads requires a careful balance between mitigating catastrophic forgetting (as discussed in Section 2.2.3), achieving efficient resource utilization, achieving scalability, and maintaining the ability of the model to generalize well to unseen data (as discussed in Section 2.4.2). Existing research often addresses parallel deep learning techniques (quantitative aspect) and continual learning (qualitative aspect) separately. When designing systems, software, or algorithms for Deep Learning, one must consider the intersection between the general fields of data science and systems engineering.

In essence, continual learning appears as an efficient solution to reduce training times and save compute resources. However, the current literature primarily focuses on single-node experiments, with accuracy being the sole metric of interest. The datasets used in these studies

are often small-scale, such as MNIST or CIFAR, which may not fully represent the complexity of real-world data. Moreover, the broader machine learning field tends to prioritize results over the efficiency of the learning process. Training times are seldom reported as a metric, despite their significance in real-world applications where data is continuously generated and the model needs to adapt in real-time or near-real-time.

In contrast, deep learning, which has been around for a longer period, has seen more studies on scalability and parallelization techniques to improve training efficiency. The challenge, therefore, lies in integrating the strengths of both distributed deep learning and continual learning — achieving an accuracy close to that of retraining the DL model from scratch while maintaining the high performance, scalability, and low resource utilization of incremental training.

2.4.2 The Efficiency Tradeoff: Generalization Gap vs. Compute Efficiency

When training a DL model as detailed in Section 2.2.2, the objective is to reduce its inherent error, striving to make its predictions as close as possible to the actual outcomes when dealing with unseen data. This difference is commonly known as the *generalization error*. However, during the optimization (training) procedure, we usually focus on reducing the *empirical error*, which is the error calculated from the training data. Focusing solely on minimizing empirical error can lead to overfitting. This limitation arises when a model becomes too complex and essentially memorizes the training data, performing exceptionally well on it but struggling to accurately predict new, unseen data. Some methods like regularization [126] limit the complexity of the model, creating a balance between reducing empirical error and improving the model's ability to generalize. Unlike gradient descent (GD), which primarily aims to minimize empirical error, stochastic gradient descent (SGD) works towards minimizing the generalization error directly [127]. This is achieved by sampling a single training samples at each iteration, as detailed in Algorithm 1. As a result, while GD is excellent at minimizing empirical error, SGD often produces models with better generalization abilities.

In Section 2.3.2, we discussed how SGD can be data-parallelized using minibatches. In this context, a larger number of training processes increases the effective minibatch size, yielding updates computed with respect to more samples. As a result, the learning algorithm starts to resemble GD rather than SGD, which can negatively affect the model's generalization capability beyond a certain threshold. Thus, choosing the minibatch size is a trade-off between statistical generalization, as a large minibatch can lead to diminishing returns beyond a certain point — resulting in a decrease in the model's achieved accuracy—, and a small minibatch does not take full advantage of parallel processing capabilities of accelerators like GPUs, resulting in poor compute efficiency.

Over the last 10 years, a number of tricks have been devised to increase the effective minibatch size beyond which generalization deteriorates. Depending on the workload, "the end of

the perfect scaling regime is anywhere from an effective minibatch size of 2^4 to a size of 2^{13} data samples“ [128]. Beyond this threshold, adding more processes to the training procedure becomes counterproductive, as the degradation in accuracy outweighs the benefits of increased compute efficiency. To continue scaling the system beyond this point, one need to address the *strong scaling* problem, which refers to the ability of a system to maintain consistent performance as the number of computing resources increases.This means finding ways to effectively utilize additional computing nodes without increasing the effective minibatch size, which can involve techniques such as reducing communication overhead or employing more advanced parallelization strategies.

CHAPTER 3

DISTRIBUTED REHEARSAL BUFFERS: A FLEXIBLE CONTINUAL LEARNING ABSTRACTION

Contents

3.1 Distributed Rehearsal Buffers and Data Parallelism 52

3.1.1 The choice for Data-parallel Experience Replay 52

3.1.2 Aggregated Memory Space of Rehearsal Buffers 53

3.1.3 Selection and Eviction Policies 55

3.1.4 Sampling Strategies for Continual Learning 56

3.2 Transparent Global Sampling of Representatives 57

3.2.1 The Need for Global Sampling of Representatives 57

3.2.2 Efficient Augmentation of Minibatches 60

3.2.3 Asynchronous Management of Rehearsal Buffers 62

We introduce our main contribution: a distributed rehearsal buffer specifically designed to enable data-parallel training for continual learning. We discuss the key design principles that are at the foundation of our proposal.

Positioning. In this dissertation, we propose asynchronous data management techniques that enable the design and implementation of a scalable distributed rehearsal buffer abstraction, which is instrumental in enabling continual learning to take advantage of data-parallel techniques.

We summarize the contributions of this chapter as follows:

— We define the concept of rehearsal buffers to address continual learning, and introduce **extensions to leverage them for data-parallel training** (Section 3.1).

— We introduce key design principles such as **asynchronous techniques to hide the overhead of managing rehearsal buffers** and to enable a full spectrum of combinations for **minibatch augmentations**. We achieve this by sampling the rehearsal buffers of remote DL model replicas using low-overhead, RDMA-aware, all-to-all communication patterns (Section 3.2).

3.1 Distributed Rehearsal Buffers and Data Parallelism

In this section, we discuss the rationale of combining data parallelism and experience replay to tackle the catastrophic forgetting problem. Both techniques can benefit from distributed systems in their own way: data parallelism speeds up the training procedure, and the instantiation of local buffers on each training process aggregates a lot of memory allowing the storage of numerous representatives. Thus, HPC techniques support both compute and statistical efficiency.

3.1.1 The choice for Data-parallel Experience Replay

Unlike vertical- and horizontal- model parallelism (Section 2.3) that require significant alterations to the DL model itself, **data parallelism** leaves the core model untouched, making it a less intrusive method. Instead, the focus is solely on the data pipeline, which is responsible for feeding training data into the model. This approach involves splitting the dataset into shards and distributing them across multiple processing units. Each unit then processes its own shard of data simultaneously, while the gradients are averaged during the back-propagation to keep the replicas in sync. This approach effectively increases the overall training speed of learning workloads. Moreover, its simplicity has led to its widespread adoption within the DL community. Researchers appreciate the ease of implementation and the significant performance boost it provides.

Experience Replay (Section 2.2.4), or *rehearsal*, is a simple continual learning technique in which the model knowledge is reinforced by replaying samples from previous tasks [69, 34]. This technique has proven to be an effective way to reduce catastrophic forgetting [74, 72]. Rehearsal is achieved by appending a fixed number of *representative samples* to each minibatch corresponding to the new training data, in order to obtain a larger *augmented minibatch* that mixes new and old training samples. With this strategy, historic training samples that are representative of patterns seen earlier are then reinjected into the training process. Similarly to data parallelism, a benefit of rehearsal is that it requires no modifications to either the DL model architecture or the training process, but relies solely on a modified data pipeline managing *augmented minibatches* containing representatives. In contrast, other CL approaches require different hyperparameters, additional code to implement regularization, and/or additional generative DL models.

In this dissertation, we focus on **data-parallel continual learning based on rehearsal**.

Prior work on rehearsal-based CL [15, 16, 17] has employed a single rehearsal buffer, with the goal of leveraging a single GPU. Here, we tackle the problem of enabling high-performance, scalable, and resource-efficient rehearsal-based CL on multiple GPUs. Efficient continual learning based on rehearsal that delivers high performance, scalability, and low resource utilization in combination with data-parallel training is challenging for two reasons: (1) the cost of man-

Table 3.1 – Continual Learning notation

T	number of CL tasks
K	number of classes
$\mathcal{B}_n$	local rehearsal buffer for process n
R_n^i	subset of $\mathcal{B}_n$ containing representatives of class i
$\mathcal{B}$	distributed rehearsal buffer
R^i	subset of $\mathcal{B}$ containing representatives of class i
N	number of representatives stored per class
c	number of candidates per minibatch
b	default minibatch size (number of samples per minibatch)
r	number of representatives added to augmented minibatches

aging a rehearsal buffer under concurrency (minibatch augmentations and constant updates) is significant, and (2) efficient data-parallel training requires to instantiate multiple independent rehearsal buffers (one per DL model replica), thus limiting the possible combinations for minibatch augmentations (i.e., reducing their diversity) to representatives observed locally. To address these challenges, we propose the use of a **distributed rehearsal buffer**: it focuses on how to minimize the overheads involved by the rehearsal buffer management while retaining the quality of minibatch augmentations under data-parallel training.

3.1.2 Aggregated Memory Space of Rehearsal Buffers

In practice, rehearsal selectively stores previously encountered raw data samples, called *representatives*, into a *rehearsal buffer*. Representatives are then sampled back from this buffer to *augment* the minibatches of new training tasks. With this strategy, historic training samples that are representative of patterns seen earlier are retained in a limited-size rehearsal buffer. Finally, the rehearsal buffer is updated by replacing some of its samples with newer ones.

In a basic version of rehearsal, a buffer denoted $\mathcal{B}_n$ stores *representative* training samples from previous tasks. Every class i observed so far is attached to its own memory space $R_n^i \in \mathcal{B}_n$, with all memories R_n^i having the same capacity. At each iteration, r representatives from $\mathcal{B}_n$ are used to *augment* the incoming minibatch m of size b, such that we obtain a larger minibatch of size $b + r$ mixing representatives and new training samples. After training with this augmented minibatch, c training samples, called *candidates*, are selected from minibatch m to be inserted into the memory R_n^i relevant to class i. If any of the memories R_n^i is full, then the new candidates replace old representatives as needed (e.g., at random or using a different strategy). This process ensures that the buffer $\mathcal{B}_n$ remains up-to-date at fine granularity (i.e., after each iteration), holding representatives of both the current and all previous tasks. For clarity, Table 3.1 provides a summary of the notation used to characterize rehearsal buffers.

Starting from this basic version, we propose the design of a *distributed rehearsal buffer* that can

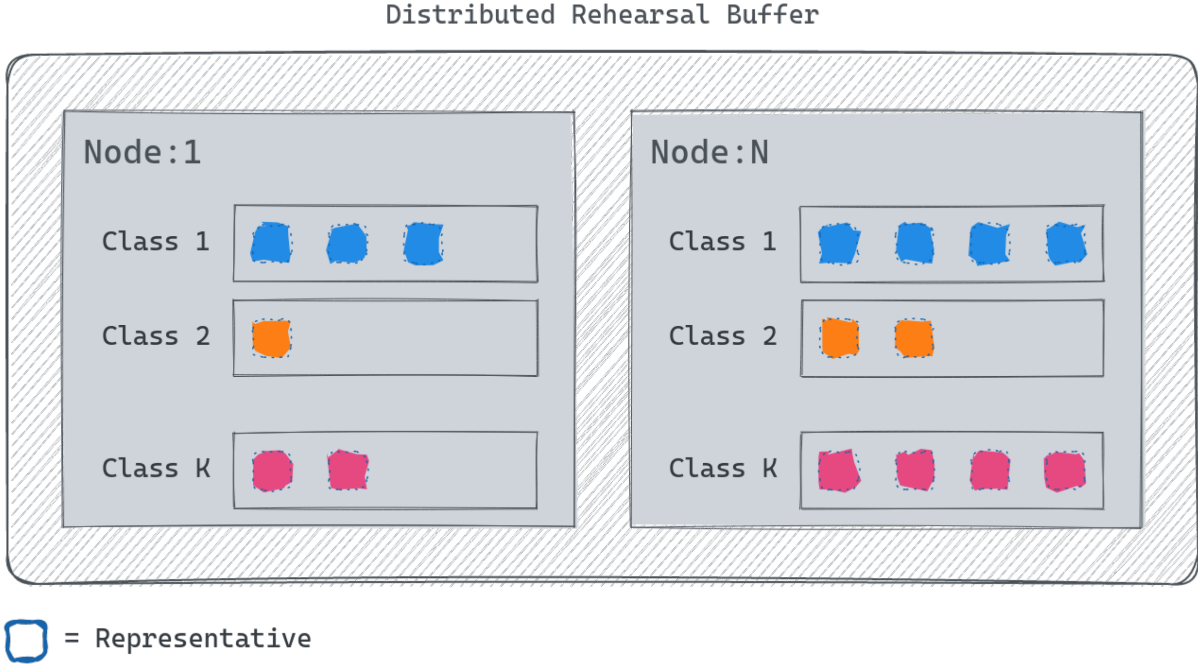


Figure 3.1 – For every process n, a rehearsal buffer $\mathcal{B}_n$ contains representatives from the classes seen so far. The distributed rehearsal buffer $\mathcal{B}$ contains representatives from the K classes.

be used with data-parallel training. The training procedure uses p distributed processes (each attached to a dedicated GPU in our case). Each process maintains its own rehearsal buffer $\mathcal{B}_n$. This approach leverages the aggregated spare memory provided by a large number of compute nodes, enabling the storage of a larger and more diverse set of representatives compared to a single, centralized buffer. Previous studies have indicated that repeatedly training over a limited number of representatives may result in overfitting, potentially impairing generalization in continual learning [82]. In addition, large models have been shown to memorize small data amounts, such as those found in rehearsal buffers with limited representatives, without acquiring generalization capabilities [129]. To mitigate this issue, our method allows the aggregated size of the rehearsal buffers to scale proportionally with the number of training processes, retaining a larger and more diverse set of representatives. Conceptually, the disjoint union of local rehearsal buffers $\mathcal{B}_n$ can be seen as a single distributed rehearsal buffer $\mathcal{B}$ as depicted in Figure 3.1:

$$\mathcal{B} = \bigsqcup_{n=1}^{p} \bigsqcup_{i=1}^{K} R_n^i = \bigsqcup_{n=1}^{p} \mathcal{B}_n$$

Assume each process can spare up to S_{max} local memory for storing $\mathcal{B}_n$. Given increasing DL model sizes, the spare host and GPU memory is under pressure, thus S_{max} is limited. On the

other hand, we need to divide S_{max} evenly between the classes to avoid a bias in the selection of the representatives (note that every local buffer $\mathcal{B}_n$ stores representatives of all possible K classes). Therefore, each memory space R_n^i can grow up to a size of S_{max}/K, which means with increasing number of classes K, each memory R_n^i shrinks. However, by using a distributed rehearsal buffer, each R^i scales with the number of processes to a size of $|R^i|_{max} = p \times S_{max}/K$, which increases the number of representatives stored per class N and therefore the diversity of minibatch augmentations. This complements data-parallel training well, since data-parallel training improves performance and scalability, not the quality of the results.

3.1.3 Selection and Eviction Policies

Since the rehearsal buffer $\mathcal{B}$ is smaller than the dataset D, we are interested in selection and eviction policies for managing the distributed rehearsal buffer. One approach to populate the local rehearsal buffers is to select candidate samples from incoming minibatches at random. To this end, we propose Algorithm 3, which is executed by each process n at every training iteration. Specifically, we pass the current minibatch m_n of size b. Every sample of m_n has a c/b probability to be pushed into the memory R_n^i corresponding to the class i. As such, c acts like an update rate: i.e., the higher the value of c, the more often representatives are renewed in rehearsal buffer $\mathcal{B}_n$. This approach has been implemented in the Naive Incremental Learning (NIL) algorithm [20] and demonstrates low computational complexity.

Algorithm 3: Rehearsal buffer updates with c new candidates for each process n

```
1  Function update_buffer(m, c):
2      C ← select c random candidates from minibatch m
3      for x ∈ C do
4          if |R_n^i| >= |R_n^i|_max then
5              replace a random representative from R_n^i with sample x
6          else
7              append sample x to R_n^i
```

In this work, since representatives are distributed among memories depicted R_n^i according to their class labels. Thus, candidate samples belonging to a specific class compete against the existing representatives of the same class to be retained in the buffer. We refer to this strategy as the *per-class management of representatives*. As depicted in Figure 3.2, a candidate sample of class i replaces a random representative in R_n^i if the latter is full, with each R_n^i having the same capacity. Furthermore, our random selection policy means that each training sample of a given class has the same probability of being replaced, regardless of whether it is a recent or old sample. This approach both increases the diversity of the augmentations and forms an embarrassingly

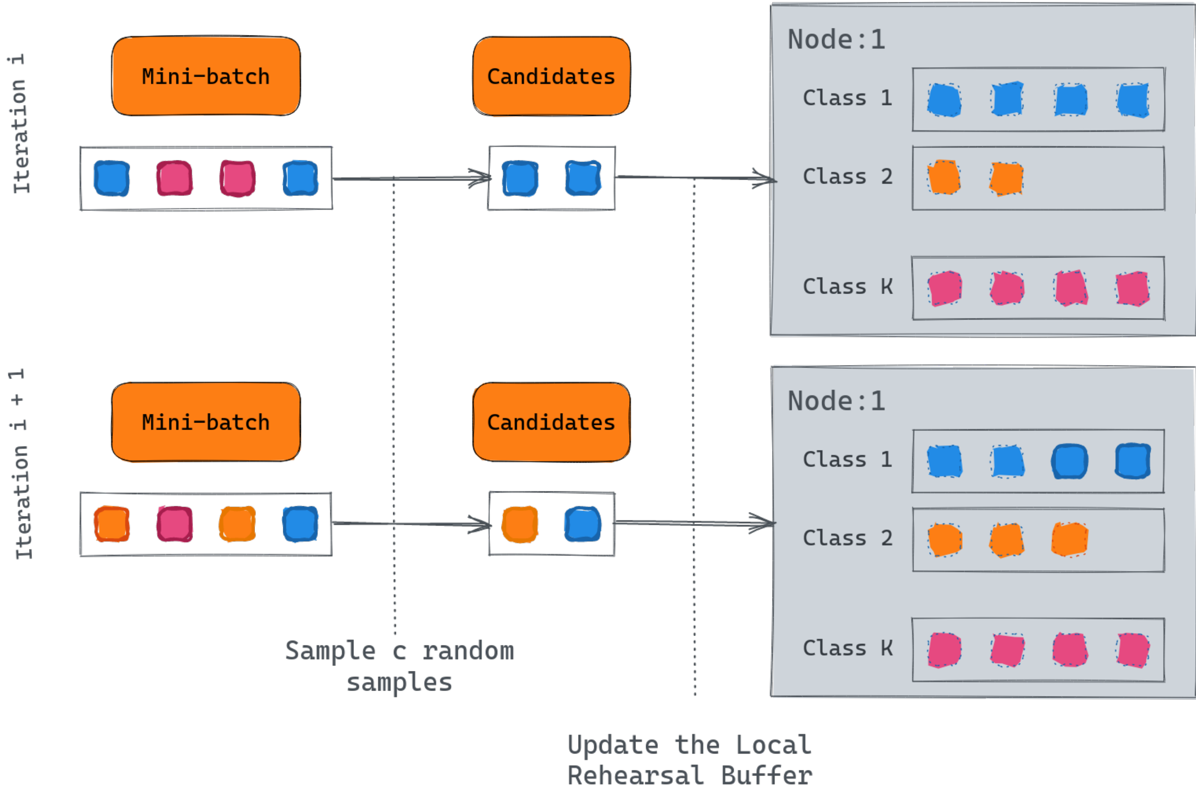


Figure 3.2 – For a given process n, c candidates from the incoming minibatch are sampled and used to populate $\mathcal{B}_n$. If the buffer for class i is full, representatives from R_n^i are replaced at random. The figure depicts the rehearsal buffer $\mathcal{B}_n$ state for two subsequent iterations for $c = 2$.

parallel pattern that is easy to implement and that has a low performance overhead.

3.1.4 Sampling Strategies for Continual Learning

In our work, we have chosen to fix the size of local rehearsal buffers. We opt for a random sampling strategy as illustrated in Algorithm 3, one of whose characteristics is that recent representatives have a higher probability to be present in the buffer than early ones. Random sampling might be sub-optimal given that a task learned long ago is more likely to be forgotten than a task learned recently, although this effect is mitigated by the per-class allocation of memory spaces R_n^i. In particular, representatives of classes seen only in the first task cannot be overwritten by representatives of classes seen only in the last task. Still, more sophisticated strategies than random sampling might further improve the achieved accuracy.

Reservoir sampling [89, 72] is a strategy to populate the rehearsal buffer of size $|\mathcal{B}_n|$ with samples coming from a stream, with sample i on process n having a $|\mathcal{B}_n|/i$ probability to be selected. This strategy guarantees the buffer to contain representatives uniformly sampled from

the input data, which makes it equivalent to an offline random sampling at each time step. Authors in [130] propose a variant of reservoir sampling that distinguishes between already-served and not-served-yet representatives to make sure that they are replayed at least once before being evicted. Another study [75] propose Balanced Reservoir Sampling, which operates at the class level to encourage the balance within the rehearsal buffer in terms of number of representatives per class i.e., a random representative from the most represented class is evicted when the buffer is full. This strategy is similar to our random sampling with per-class memory allocations, with the added benefit that the number of classes does not need to be known before the training procedure.

The distributed rehearsal buffer could leverage gradient-based selection [76] or loss-based selection [75] of representatives to be stored. These strategies prioritize samples that contribute most to the learning procedure, either by maximizing the gradient norm or by focusing on samples with higher loss values, respectively. The overall expected loss of the rehearsal buffer can be computed without backpropagation, which makes the former strategy less computationally efficient than the latter. Since they are complementary and address different issues, these strategies can be combined with reservoir sampling or Balanced Reservoir Sampling [75].

We leave such studies for future, as we focus on performance aspects and parallelization techniques in this dissertation.

3.2 Transparent Global Sampling of Representatives

Our distributed rehearsal buffer is built from many local rehearsal buffers that might live on different compute nodes. In this section, we motivate the need (1) to sample representatives from both the local and remote rehearsal buffers i.e., *globally*, to help convergence of the model in all configurations; (2) to leverage HPC-oriented optimizations to mitigate associated overheads; and (3) to manage such global sampling asynchronously in the background, effectively hiding it from the perspective of the main training procedure.

3.2.1 The Need for Global Sampling of Representatives

We propose to leverage rehearsal buffers to retain representative samples on each training process, allowing to later reinject them into the training procedure for rehearsal. We make these local buffers *distributed* thanks to global sampling, so that minibatch augmentations can benefit from representatives stored remotely. In this section, we motivate this particular design decision in two settings : uniform and non-uniform input data distributions.

The minibatch SGD algorithm described in Section 2.2.2 requires individual minibatches being sampled from random permutations of the dataset, which are generated at the start of each epoch. Additionally, as discussed in Section 2.3.2, data-parallel training involves synchro-

nizing replicas of the model, each trained on a different data shard. In practice, distributing the training of a DNN in a data parallel fashion requires loading input samples onto each compute node, typically through a Parallel File System (PFS). This allows each training process to ingest a different shard of the samples during each epoch, with the common assumption that indices should be globally shuffled to enhance generalization [131, 132]. The shuffling can either be performed in-place or by loading the dataset into DRAM space with a random order [133].

However, due to rapidly growing dataset sizes, this approach of storing entire datasets locally has become increasingly infeasible. The authors in [134] proposed *local shuffling* to shuffle training samples from a local data shard, thereby saving I/O operations. In order to study the practical conditions under which the local and global gradients equivalence can be applied without degrading model convergence, the authors quantified the bias introduced by the shuffling error $\epsilon(A, D)$ of algorithm A. The local shuffling scheme can be formulated as insufficient global shuffling that is not uniformly distributed, and has the following convergence rate's upper bound in the non-convex case [135]:

$$O\left(\sqrt{\frac{1}{S|D|}} + \frac{log|D|}{|D|} + \frac{|D|\epsilon(A, D)^2}{p \times b}\right) \tag{3.1}$$

Where $|D|$ is the number of samples in the static training dataset D, p is the number of training processes, b is the minibatch size (per training process), and S is the number of epochs. To ensure that shuffling error does not dominate the bound in Equation 3.1, the following condition has to be satisfied: $\epsilon(A, D) \leq \sqrt{\frac{p \times b}{|D|}}$ [135]. To simplify the notation, we use $d = |D|$ to denote the number of samples in the dataset. The shuffling error is defined as follows:

$$\epsilon(A, D) = \frac{1}{2} \sum_{\pi_i([d]) \in \pi([d])} |u_{\pi_i}[d] - v_{\pi_i}[d](A, D)| \tag{3.2}$$

Where u_π is the uniform distribution on the set that contains all permutations of $\pi([d])$, i.e. permutations of all different ways the samples can be picked from the dataset ($|D|!$ permutations), and $v_\pi[d](A, d)$ is the distribution after shuffling the dataset containing $d = |D|$ samples using algorithm A. Furthermore, the authors introduce the *partial local sampling* scheme as a method to control the fraction Q of input data that is globally shuffled. This stands as a middle ground between *global shuffling*, where the entire dataset is shuffled and distributed across training processes, and *local shuffling*, where each process reuses the same data shard during each epoch. The aim of partial shuffling is to assure that the shuffling error $\epsilon(A, |D|)$ does not dominate the convergence bound by setting Q accordingly. The number of permutations σ that would include the desired partial local shuffling factor Q between the p shards is as follows:

$$\sigma = \frac{|D|}{p}! \times P_{\frac{Q|D|}{p}}^{\frac{|D|(p-1)}{p}} \times P_{\frac{Q|D|}{p}}^{\frac{|D|}{p}} \times \left(\frac{|D|}{p}(p-1)\right)! \tag{3.3}$$

The authors simplify Equation 3.2 which can be written as:

$$\epsilon(A, D) = 1 - \frac{\sigma}{|D|!} \tag{3.4}$$

The authors conclude that "for practical dataset sizes and number of training processes, the shuffling error $\epsilon(A, D)$ would approach the value 1, and the shuffling error would hence dominate the convergence rate in Equation 3.1. For instance, practical settings for training ImageNet-1K ($d = |D| = 1.2 \times 10^6$) on any number of training processes $4 \leq p \leq 100.000$ and with a global, effective minibatch size $p \times b$ of less than 100K yields a shuffling error $\epsilon(A, D) \approx 1$. This result shows that more studies for locality schemes are needed to improve on the convergence bounds." As it stands, one can not rule out the usefulness of global shuffling, even in the ideal data parallel setting where local and global data distributions are uniform.

The authors conducted extensive experiments to verify this point in practice. In most of them, local shuffling achieves similar accuracy as the global shuffling approach. However, this is not always the case. For instance, when the training procedure is scaled up to 2,048 GPUs using the ResNet-50 model with ImageNet-1K, global shuffling achieves a higher accuracy than local sampling, with a gap of 9%. Similarly, for ImageNet-50 (a subset of 50 classes from the original dataset) using 128 GPUs, the gap between global and local shuffling results in up to a 30% drop in accuracy. Even using 32 GPUs, a 10% decrease in accuracy is observed. For ImageNet-50 with 128 training processes, a high exchange rate of $Q \geq 70\%$ is necessary to achieve convergence to an accuracy that is substantially closer to that of global shuffling compared to local shuffling. This observation suggests that global shuffling is particularly important when the number of classes to be learned is low. The experiments reveal that local shuffling achieves a similar accuracy as global shuffling in most, but not all cases, suggesting that some DNNs are more sensitive to sample diversity than others. Therefore, setting Q to a sufficiently high value is necessary to exchange a fraction of samples globally, which guarantees that model convergence will not be degraded in a data parallel context.

Authors in [136] report on extensive experiments using different shuffling algorithms, and conclude that the more random training samples are, the better the convergence rate of SGD is. Notably, the Sliding-Window Shuffle using a sliding window to perform partial data shuffles, and partial shuffling based on reservoir sampling, both yield a degraded accuracy.

Performing minibatch augmentations using the local rehearsal buffer $\mathcal{B}_n$ (without global sampling) is akin to partial local sampling. However, authors of the results discussed in [135] consider a fixed dataset D whose permutations can be calculated before each epoch, whereas we sample r random representatives from the rehearsal buffer each time a minibatch is augmented. Consequently, the convergence upper bound in Equation 3.1 does not apply to our specific setting, as 1) a given representative might be sampled from $\mathcal{B}_n$ multiple times during an epoch; and 2) some random representatives are updated after every training iteration. Despite this, as

shown in the articles mentioned above, information exchange between training processes can be beneficial in the long term for learning convergence. We choose to opt for global sampling of representatives based on these results. We are aware that additional work is necessary to formally demonstrate the extent to which these results apply to our continual learning setting based on rehearsal. We share the following hypotheses:

— As a given representative might be sampled multiple times to be reinjected into the training procedure when using rehearsal (especially if the update rate c is set to a low value), the bias introduced by locality would be amplified, affecting model convergence negatively. In contrast, global sampling enhances the diversity of minibatch augmentations by sampling from the distributed rehearsal buffer.

— As local buffers are progressively populated, the same representatives tend to be sampled often towards the early stage of the training procedure. The previous bias is then amplified during the early stage of training, which can be particularly detrimental for training stability on the long run.

— In continual learning scenarios, the input data is typically acquired over time, which often leads to tasks with few classes to learn. As shown by experiments in [134], such settings do not benefit locality schemes.

To ensure the wide applicability of rehearsal with data parallelism, global sampling might be useful as a means to accommodate all scenarios, batched vs. streamed settings, models, training scales, and datasets. This design decision bolsters the robustness and versatility of our distributed rehearsal buffer, regardless of the specific context or problem at hand.

3.2.2 Efficient Augmentation of Minibatches

Experience Replay consists in interleaving representatives with the current minibatch m to build a new augmented minibatch m'. As depicted in Figure 3.3, at every training iteration, r representatives are sampled without replacement from $\mathcal{B}$ to assemble m', whose size is $b+r$. We call this operation *minibatch augmentation*. Existing research has shown that uniform sampling from a rehearsal buffer is effective in many cases [20, 73], while demonstrating no additional computational complexity. Thus, we adopt the same principle in our proposal.

With a distributed rehearsal buffer $\mathcal{B}$, each process n needs to sample r representatives concurrently with the other processes. To this end, we could simply adopt a naive *embarrassingly parallel* strategy that chooses the r representatives of each process n from the local rehearsal buffer $\mathcal{B}_n$. Although highly efficient and easy to implement, such a strategy limits the number of combinations possible for the selection of the r representatives relative to the global rehearsal buffer $\mathcal{B}$, which reduces the diversity and the quality of the augmentations. As a consequence, one need to provide a fair sampling that gives every training sample in $\mathcal{B}$, regardless of its location, an equal opportunity to be selected among the r representatives of each process. This

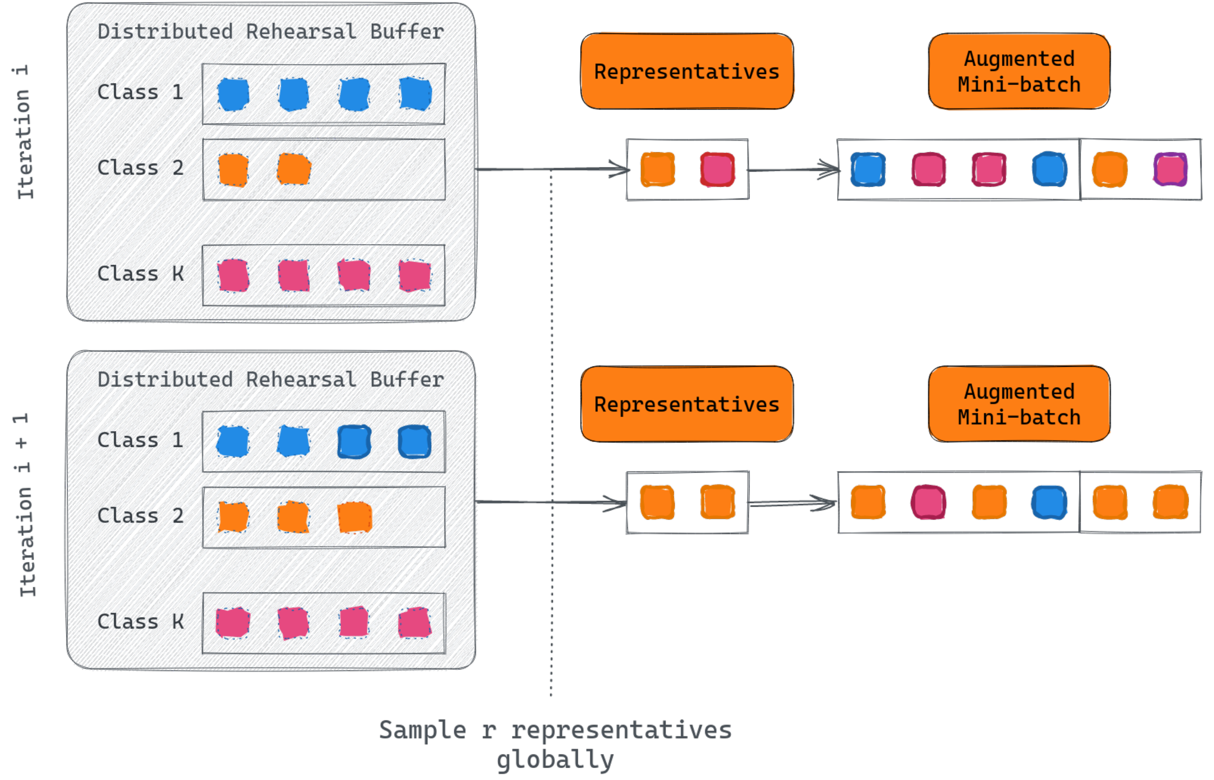


Figure 3.3 – On a given process n, every incoming minibatch is augmented with r representatives sampled randomly and without replacement from the distributed rehearsal buffer $\mathcal{B}$. Here, $r = 2$ on two subsequent iterations. Sampling from $\mathcal{B}$ introduces communication between the p distributed processes.

approach allows *any* model replica to ingest *any* training sample as input. Algorithm 4 details the procedure for selecting representatives to be sampled. Each process n generates a list of r indices between 0 and the maximum number of representatives stored in the distributed rehearsal buffer $|\mathcal{B}|$. This includes the aggregation of the p local buffers. Once such global indices have been generated, indices local to each $\mathcal{B}_n$ are derived considering the number of different classes K and the number of representatives N stored per class.

To address global sampling efficiently, we leverage two technologies commonly used in HPC. First, we propose to pin the space reserved for each local rehearsal buffer B_n into the memory of the compute node hosting process n. Then, we expose the pinned memory for RDMA access. Thus, we enable low-overhead, fine-grain access to the rehearsal buffer of each process from every other process. Secondly, while global sampling is synchronized with `allreduce` operations for gradient updates, we aim to decouple buffer management from model training. This decoupling allows independent servers to respond to requests without relying on additional synchronization barriers that would introduce unnecessary delays. Therefore, we

Algorithm 4: Pick r random representatives from the distributed rehearsal buffer $\mathcal{B}$

```
1 Function pick_random_indices(r):
2     map ← {}
3     C ← select r indices without replacement from [0, |B|]
4     for i ∈ C do
5         local_index ← i%(K × N)
6         n ← i/(K × N)
7         append local_index to map[n]
8     return map
```

propose a point-to-point communication pattern powered by low-overhead, remote procedure calls (RPCs). Specifically, we introduce several key concepts: (1) concurrency control based on fine-grain locking to guarantee consistency and mitigate contention between updates to the rehearsal buffers and local/remote reads issued by augmentations; (2) progressive assembly of augmented minibatches using concurrent asynchronous RPCs, which hide the remote access latency; (3) RPC consolidation to transfer the training samples in bulk from the same remote rehearsal buffer, reducing the number of RPCs.

3.2.3 Asynchronous Management of Rehearsal Buffers

Even with our proposed optimizations, the overheads of managing a distributed rehearsal buffer may still be significant. Therefore, we also devise an asynchronous technique to hide these overheads, such that a training iteration can proceed without blocking every time that it needs to interact with the distributed rehearsal buffer. That way, the distributed rehearsal buffer management is effectively hidden from the perspective of the main training procedure.

To this end, we revisit the major steps (proposed in Section 2.3) of CL based on rehearsal and data-parallel training: ① (**IO**) loading an original minibatch from the data pipeline; ② prepare the augmented minibatches, which involves global sampling from the distributed rehearsal buffer; ③ update the distributed rehearsal buffer using the new samples of the original minibatches; ④ (**FB**) perform a forward pass with the augmented minibatch as input data, followed by a backward pass that averages the gradients and updates the parameters w of each DL model replica; ⑤ (**GE**) the gradient exchange across distributed training processes and ⑥ (**WU**) updating the DNN weights.

Therefore, we can use the following strategy: ② wait until r representatives were collected asynchronously by global sampling started during the previous iteration and concatenate them with the current minibatch to obtain an augmented minibatch; ③ start an asynchronous update of the distributed rehearsal buffer using the original minibatch, followed by asynchronous global sampling of the next r representatives; perform the same steps ④, ⑤ and ⑥ as above

Figure 3.4 – Asynchronous updates of the rehearsal buffers and global augmentations: r representatives sampled globally beginning with the previous iteration are used by the training loop to assemble an augmented minibatch on each process n. Meanwhile, the distributed rehearsal buffer extracts candidates from the current minibatch to update each B_n locally, then collects the next r representatives using global sampling.

so that the rehearsal management is conducted concurrently with the training iteration. This procedure is illustrated in Figure 3.4. The waiting step ②, which acts as a (blocking) synchronization point between the training procedure and minibatch augmentations, is highlighted in red.

Using this approach, training iterations only need to wait if buffer updates and global sampling cannot keep up with them, introducing a delay. In an ideal scenario, the communication and synchronization overheads related to the management of the rehearsal buffer are fully absorbed by training steps.

It is important to note though that even in the case when the rehearsal buffer overhead can be fully absorbed asynchronously (i.e., no wait at step ② blocking the training procedure), the training iteration operates with an augmented minibatch of size $b + r$ (instead of the original size b). Thus, each training iteration is slowed down by a factor of r/b. This overhead is inherent to rehearsal-based CL and cannot be avoided i.e., additional representatives are fed to the DNN, inducing extra time to perform forward and backward passes. However, by fixing r and hiding the rehearsal buffer management overheads through asynchronous techniques, our approach

can deliver performance levels close to the theoretical lower bound at scale.

Our approach assembles augmented minibatches in advance, anticipating the next training iteration. The current training iteration operates on a different, read-only minibatch. This is purely a performance improvement and is functionally equivalent to a synchronous rehearsal CL approach, therefore subject to the same convergence. Furthermore, our approach is embarrassingly parallel because each training process executes Algorithm 4 to sample representatives (without replacement) from other workers without informing or involving them in the decision. This also means that random sampling is conducted *with replacement* at the global level. Besides, synchronization happens only at the process level (i.e., replacement of samples can be delayed until reads have finished) and not across processes.

Conclusion

In this chapter, we described a novel *distributed* rehearsal buffer abstraction that efficiently complements data-parallel training on multiple GPUs, allowing us to achieve short runtime and scalability while retaining high accuracy. It leverages a set of buffers (local to each GPU) and uses several asynchronous techniques for updating these local buffers in an embarrassingly parallel fashion, all while handling the communication overheads necessary to augment input minibatches (groups of training samples fed to the model) using unbiased, global sampling. The novelty of our work lies in the techniques to make rehearsal scalable in the context of data-parallel training, which we believe presents an opportunity for specialized data management techniques that are applicable to a broad class of rehearsal approaches. To our best knowledge, such HPC-oriented aspects aimed at improving training performance using rehearsal were not explored before.

Chapter 4

Neomem: an Efficient Implementation of Rehearsal-based Continual Learning

Contents

4.1 Architectural Overview **66**
4.1.1 Storing and Serving Data Samples 66
4.1.2 Integration with the Training Procedure 67
4.2 Asynchronous Techniques for Efficient Buffer Management **69**
4.2.1 Multi-threaded Concurrency 70
4.2.2 Non-blocking RPCs 71
4.2.3 Asynchronous CUDA Copies 72
4.3 I/O Optimizations in Data Movements **73**
4.3.1 Transferring Data using RDMA-enabled RPCs 74
4.3.2 RPC Consolidation 75
4.3.3 Leveraging Page-locked Buffers for Efficient Data Transfers 75

In this chapter we dive into the details of *Neomem*, the implementation of our proposal of a distributed rehearsal buffer supporting the global sampling of representatives. The architecture consists of multiple instances, each acting as both a client and a server, providing services to interact with rehearsal buffers through Remote Procedure Calls (RPCs).

We summarize the contributions of this chapter as follows:

— We present the **architectural overview** of our implementation (Section 4.1).

— Next, we present **asynchronous techniques** leveraged to prepare augmented minibatches **in advance**, in order to run Neomem in parallel with the training procedure (Section 4.2).

— We also present some **optimizations in data movements**, such as bulk transfers of representatives, in-place preparation of augmented minibatches and pinned memory (Section 4.3).

Our Neomem implementation is publicly available as an **open-source project** [29]. For reproducibility purposes, we also provide the Python library managing the Continual Learning setting named *distributed-continual-learning* [30]. The approach is implemented as a high-performance C++ library with Python bindings.

4.1 Architectural Overview

Neomem is responsible for accumulating representatives in the buffer and serving augmented minibatches. It utilizes instances on multiple nodes to leverage the aggregated memory of many compute resources. Neomem achieves these operations by leveraging high-performance techniques to avoid inducing any overhead that would interfere with the training procedure. The implementation of such optimizations is detailed in next sections.

4.1.1 Storing and Serving Data Samples

Neomem is responsible for the following two functions:

— The *accumulation* of representatives in the buffer: Neomem stores representative samples in host memory in the form of *tensors*. By utilizing instances of Neomem on multiple nodes, the system can leverage the aggregated memory of many compute resources to store a larger number of representatives. This aspect is detailed in Section 3.1.

— The service of augmented minibatches: Neomem manages the assembly of augmented minibatches, sampling representatives stored in both local and remote buffers. This is enabled by global sampling, allowing every process involved in the training procedure to sample representatives stored on other (remote) processes. Global sampling is detailed in Section 3.2.

The architecture of Neomem is a distributed system leveraging point-to-point communication, where each instance functions as both a client and a server simultaneously. In this setup, every instance is capable of providing and consuming services. Each server in this architecture exposes a set of RPCs that can be invoked by any client. This paradigm is appropriate for client-server patterns, in which the server is passive and handles requests from clients on-demand. RPCs enable the execution of functions as if they were local, abstracting the underlying communication details.

Instances that provide RPC services are referred to as *providers*. Providers encapsulate the server-side functionality, handling incoming requests, processing them, and returning the results to the requesting client i.e., the current or another provider. Each provider runs a polling loop in a separate thread `async_process()`, which is responsible for continuously monitoring and processing incoming requests in the distributed system. At the core of this system are two queue objects: a request queue and a response queue.

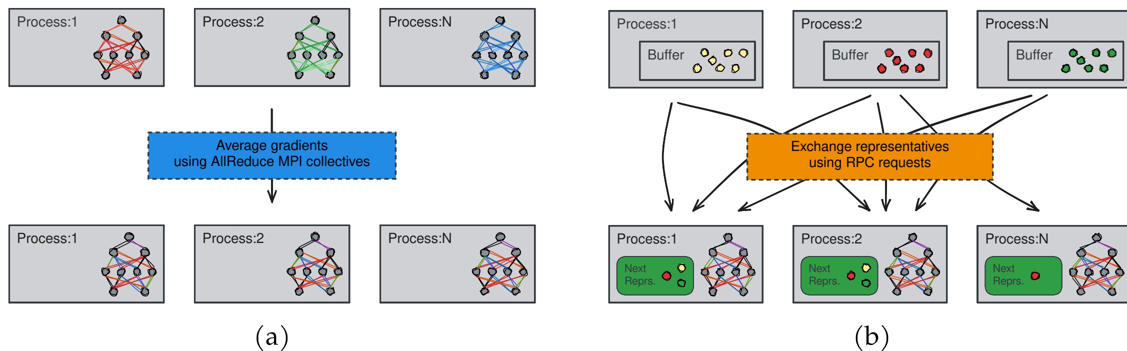


Figure 4.1 – Gradients are exchanged using MPI `allreduce` collectives. Representatives are exchanged between remote processes using RPC requests.

- Request queue: this queue stores incoming requests to be processed by the server. Request objects encapsulate a minibatch of size b originating from the data pipeline, as well as a pre-allocated augmented minibatch of size $b + r$ to be prepared. Such request objects are pushed into the queue via the `accumulate()` function, and consumed by the `async_process()` thread.
- Response queue: this queue is used to signal that an augmented minibatch of size $b+r$ has been prepared. Specifically, such an augmented minibatch contains b original samples and r representatives. The `wait()` function retrieves the responses from the queue, or waits if the response queue has not been populated by `async_process()` yet. Response objects only contain the number of representatives r contained in the last augmented minibatch. The emission of a response object is a signal that the oldest augmented minibatch in the request queue, which had been allocated before the call to `accumulate()`, is ready for subsequent use.

The functions interacting with these two queue objects, `accumulate()` and `wait()` respectively, are further detailed in Table 4.1. Queues are managed using fine-grain locking to avoid inconsistent states.

4.1.2 Integration with the Training Procedure

We implemented our approach as a high-performance C++ library that offers convenient bindings for Python using *pybind11*, exposing the public API detailed in Table 4.1 to the DL model training procedure. The complexity of our proposal is then completely hidden from end-users using Python AI runtimes. Figure 4.1 summarizes the general idea: the training procedure is implemented using any AI runtime, typically using Python, and interacts in a parallel fashion with Neomem (managing the distributed rehearsal buffer) through bindings.

The communication between the model training code and the distributed rehearsal buffer is

Table 4.1 – Neomem public API

Method	Description
`accumulate(minibatch, aug_minibatch)`	This function is responsible for (1) the preparation of the next augmented minibatch, involving the asynchronous global sampling of r representatives; and (2) the accumulation of new representatives in the local buffer following the policy described in Algorithm 3. `accumulate()` should be invoked with an original minibatch originating from the data pipeline, and a pre-allocated augmented minibatch to be prepared as arguments.
`wait() -> num_representatives`	This function waits for (1) an augmented minibatch to be ready, and (2) the local buffer to be updated with new representatives. `wait()` acts as a synchronization point (as illustrated in Figure 3.4), suspending the training procedure in case it is not. Only the number of representatives contained in the pre-allocated augmented minibatch is returned, as the latter is modified in-place.
`async_process()`	This function consumes the data pushed into the request queue by `accumulate()`, processes it, triggers the operations needed to manage the distributed rehearsal buffer, and signals the completion of the current augmentation via the response queue (which will be consumed in turn by `wait()`). This function operates in a separate thread.

achieved using a convenient *update* primitive, which encapsulates all of our contributions. This primitive is illustrated in Listing 4.1.

```
1 minibatch = DataPipeline.get_next_minibatch()
2 aug_minibatch = preallocate_augmented_minibatch()
3
4 def update():
5   r = RehearsalBuffer.wait() # wait if the augmentation is not ready yet
6   RehearsalBuffer.accumulate(minibatch, aug_minibatch)
7   return r # aug_minibatch now contains b+r training samples
```

Listing 4.1 – The *update* primitive (highlighted in pink) waits for the augmented minibatch being prepared to be ready for ingestion, then updates the local rehearsal buffer, then starts the preparation of the next augmented minibatch.

The design proposed in Section 4.1.1 involves queues allowing for multiple minibatches to be augmented in advance. Such a scenario could occur in cases where individual minibatch augmentations (involving global sampling of representatives) were faster than individual training iterations, resulting in multiple calls to `accumulate()` for a single call to `wait()`. However, for simplicity's sake, we have chosen to define the `update()` primitive so that a call to `wait()` corresponds to a call to `accumulate()` in a 1:1 fashion.

Prior to processing by Neomem, a minibatch designated for augmentation must be declared in Python and transferred to Neomem via *pybind11*. Neomem then receives this pre-allocated minibatch and performs in-place augmentation within the C++ code to avoid unnecessary copies. However, given that this augmentation occurs concurrently with model training, the same minibatch cannot be utilized simultaneously for gradient computation. Consequently, two augmented minibatches are required for each iteration: one for model training (acquired prior to the preceding invocation of `wait()`), and another for the augmentation procedure executed by Neomem. Before the subsequent iteration, these two variables are interchanged, as elaborated in Listing 4.2.

```
1  aug_minibatch_1, aug_minibatch_2 = preallocate_augmented_minibatches()
2
3  for i in range(num_steps):
4    minibatch = DataPipeline.get_next_minibatch()
5
6    if i % 2 == 0:
7      r = RehearsalBuffer.update(minibatch, aug_minibatch_1)
8      Model.train(aug_minibatch_2)
9    else:
10     r = RehearsalBuffer.update(minibatch, aug_minibatch_2)
11     Model.train(aug_minibatch_1)
```

Listing 4.2 – Example of a training loop integrating our proposal.

For the purpose of this work, we integrate our proposal with PyTorch [124] and rely on Horovod [32] to enable data parallelism. We rely on NVIDIA DALI [137] as the data pipeline that provides the original minibatches. Thanks to the encapsulation into a separate primitive, our approach can be easily extended to support other AI runtimes (such as TensorFlow [125]), data-parallel implementations or data pipelines.

4.2 Asynchronous Techniques for Efficient Buffer Management

Unlike model training code, which is usually written in Python, we implemented Neomem in C++. There are multiple reasons for this choice: (1) Python has limited support for multi-threaded concurrency due to the global interpreter lock that allows only a single thread to run

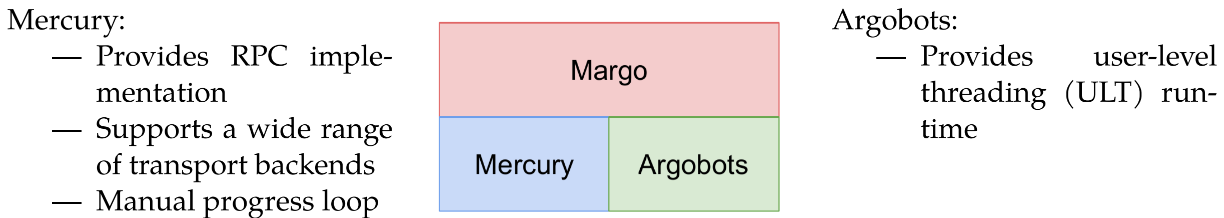


Figure 4.2 – Margo is a library built on top of Mercury and Argobots, running a Mercury progress loop in a ULT and converting RPC handlers into ULTs.

interpreted code; (2) our approach requires low-overhead access to system-level resources, notably operations involving CUDA transfers, and a more direct control over memory management; (3) the overheads of interpreted languages are unacceptable in our case given the need to provide consistency and manage multiple connections under concurrency. Overall, C++ offers robust support for parallelization techniques and asynchronous data movements, enabling us to fully leverage multi-core processors and GPU acceleration. Neomem builds atop the Mochi framework [138] in order to leverage its network abstraction that enables the use of high-performance network transports (e.g., libfabric, UCX), backends (e.g., tcp, verbs, gni) and techniques (RDMA).

4.2.1 Multi-threaded Concurrency

The Mochi framework [138] enables the composition of specialized distributed data services from a collection of connectable subservices. Mercury implements RPCs, while Argobots provides user-level threading (ULT) capabilities for concurrency management. Margo provides Argobots-aware wrappers to Mercury functions, simplifying service development by expressing Mercury operations as conventional blocking functions. This way, the caller does not need to handle progress loops or callbacks. The relationship between these components is illustrated in Figure 4.2. Margo internally suspends callers after initiating a Mercury operation and automatically resumes them upon completion. This design enables other concurrent user-level threads to make progress while Mercury operations are ongoing, without consuming operating system threads.

Each Neomem provider runs a polling loop inside the `async_process()` function executed in a dedicated Argobots ULT. This lightweight thread runs in its own *execution stream* driving the Mercury progress loop. This is necessary to avoid collisions with MPI collective operations (that would result in program hangs) issued by the model training code to synchronize gradients (we recall that the rehearsal management operated by Neomem is cadenced by the `allreduce` operations issued by the AI runtime, acting like synchronization barriers). The interactions with

request and response queues detailed in Section 4.1.1 require locking to ensure thread safety.

From the *client* perspective, each provider is responsible for: (1) preparing the next augmented minibatch (involving global sampling); and (2) accumulating representatives in the local rehearsal buffer via `accumulate()`, as outlined in Table 4.1. Conversely, from the *server* perspective, this entails serving an average of r representatives to other providers. Every local buffer is therefore accessed for both writing and reading operations during each iteration, potentially concurrently, as independent servers respond to requests without necessitating additional synchronization barriers beyond those issued by the AI runtime for gradient reduction. To ensure consistency of local rehearsal buffers and mitigate contention between updates and reads, concurrency control based on fine-grain locking is essential. Specifically, we want to avoid representatives being sampled by a remote process while being replaced locally by `accumulate()`.

To serve concurrent RPCs, we implemented a low-overhead, userspace thread pool on each provider using Argobots. This thread pool allows for efficient handling of multiple RPCs simultaneously. The function `get_representative()` is invoked via RPC and interacts with the local buffer for a read operation following the policy described in Algorithm 4. Threads can be redirected to different cores based on the provider configuration, allowing for resource sandboxing.

4.2.2 Non-blocking RPCs

Non-blocking RPCs enable providers to access services like `get_representatives()` from other instances in the distributed system without waiting for a response. This approach allows providers to continue with other tasks while waiting for the RPC to complete. By using concurrent asynchronous RPCs, one can progressively assemble augmented minibatches, which helps to hide remote access latency introduced by the global sampling of representatives. Specifically, at every iteration, each *client* dispatches r RPCs to sample r remote representatives via `dispatch_rpcs()`. Once all RPCs are issued, clients wait for their resolution using `resolve_rpcs()`. Training samples that are returned are then written in a page-locked buffer as they resolve. These functions are further detailed in Table 4.2.

Using MPI collectives is an alternative to RPCs, albeit with certain limitations. Collective communication assumes that all participants are ready before they can proceed. In our case, we have a client-server model where independent servers respond to requests that are not necessarily synchronized because of contention. One-sided MPI communication could potentially address this limitation, but it is non-trivial to adopt in a consistent fashion (e.g., ensuring one-sided get from the rehearsal buffer does not occur in the middle of an update).

Table 4.2 – Description of the Neomem functions used to issue RPCs

Method	**Description**
`dispatch_rpcs(vector<async_res> responses)`	This function is invoked by `accumulate()` to prepare the next augmented minibatch. It dispatches r `get_representative()` asynchronous RPCs to sample r representatives on remote training processes selected randomly. This function calls `pick_random_indices()` as detailed in Algorithm 4 to generate indices globally. Such indices determine both the process n to contact and the local index of the representative to sample in the corresponding buffer.
`get_representative(request& req, int i)`	This function is invoked remotely as a RPC i.e., it is an RPC handler function to service global sampling of representatives. It interacts with the local buffer for a read operation at index i (passed as argument by the caller). Index i is chosen by `dispatch_rpcs()` via `pick_random_indices()` to ensure a selection without replacement across the entire distributed rehearsal buffer. The representative training sample is written back to the `req` object.
`resolve_rpcs(vector<async_res> responses)`	This function waits for concurrent RPCs issued by `dispatch_rpcs()` to resolve. The returned training samples are written into a pre-allocated page-locked buffer.

4.2.3 Asynchronous CUDA Copies

Minibatches originating from the data pipeline are typically moved onto the GPU by a dataloader for faster ingestion by the model training code. References to these minibatches are then passed to Neomem through bindings, alongside an augmented minibatch pre-allocated on the GPU memory (as detailed in Section 4.1.2) via `accumulate(minibatch, aug_minibatch)`. Neomem works with this data using the CUDA API: variable `minibatch` is only utilized for read-only operations, while `aug_minibatch` is exclusively written into, undergoing in-place modifications visible from Python. Neomem takes advantage of dedicated, asynchronous CUDA streams to improve data transfer performance between augmented minibatches residing in GPU memory and representatives residing in host memory. This enables the overlap of data transfer with computation, minimizing the time spent waiting for data to be transferred. There are several key operations in Neomem that benefit from asynchronous CUDA copies which we summarize in Table 4.3.

Table 4.3 – Description of the Neomem functions taking advantage of the CUDA API

Method	Description
`copy_last_batch(minibatch, aug_minibatch)`	This function is invoked by `accumulate()`. It copies the b training samples from the original minibatch to the b first "slots" of the pre-allocated augmented minibatch (of size $b + r$). In this way, the current minibatch is used to prepare the next augmented minibatch. This operation can be performed asynchronously.
`copy_buffer_to_batch()`	This function is invoked after `resolve_rpcs()` completed. It copies the r representatives that were just sampled globally, and written to a receiving buffer, to the r last "slots" of the pre-allocated augmented minibatch of size $b + r$. This operation can be performed asynchronously.
`update_buffer(minibatch, c)`	This function is invoked by `accumulate()`. It updates the local rehearsal buffer with c random samples from the input minibatch following the policy described in Algorithm 3. This operation can be performed asynchronously.

To ensure correctness, CUDA streams are synchronized at the end of each iteration, with one synchronization point being when the *wait*() function is called. This ensures that all data transfers and computations have completed before proceeding to the next iteration. We have instantiated a different CUDA stream for each of the functions described in Table 4.3. However, it is worth noting that additional CUDA streams can be created to transfer more data without significantly impacting performance, allowing for further overlap of data transfer with computation.

4.3 I/O Optimizations in Data Movements

When building a system leveraging parallelization techniques like Neomem, managing low-level read/write client interfaces presents a significant challenge. This complexity arises from three primary factors: (1) competition for network bandwidth, since many processes sharing the same compute node need to transfer training samples from remote rehearsal buffers at the same time; (2) difficult all-to-all communication patterns, since each process needs to access the rehearsal buffers of every other process; (3) low latency requirements, since each process needs to access a small number of training samples from each remote rehearsal buffer. We propose techniques aiming to optimize the system's performance and enable seamless integration of

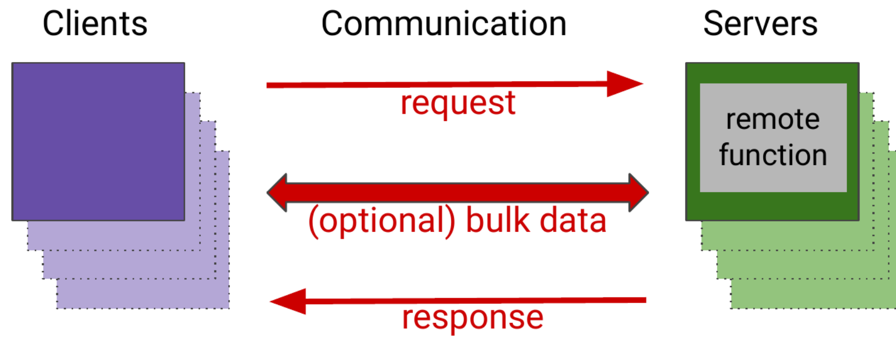


Figure 4.3 – The RPC system implemented in Mercury has a focus on high concurrency and support explicit bulk transfers.

remote rehearsal buffers in distributed environments.

4.3.1 Transferring Data using RDMA-enabled RPCs

To enhance the performance of global sampling, Neomem employs Remote Direct Memory Access [102] (RDMA) technology when it is natively supported by the underlying network fabric. RDMA is a hardware-supported technology that allows a process on one compute node to access the memory of another process on a different node over the network, without involving the CPU. A fast path for I/O operations is therefore created in addition to RPC responses, as illustrated in Figure 4.3. The RDMA protocol supports zero-copy networking by enabling the host adapter to determine, upon a packet's arrival in the network, which application should receive it and where it should be stored within the application's memory space. This eliminates the need to send the packet to the kernel for processing and copying it into the user application's memory. Instead, the host adapter directly places the packet contents into the application buffer, bypassing the kernel intervention in the communication process entirely. This approach significantly reduces the latency of small accesses to remote rehearsal buffers, making it a crucial component of our efficient data transfer strategy.

To achieve RDMA transfers in practice, clients expose buffers of memory of the size required to store r representatives using Mercury. Transfers can then take place in both ways: either the server pulls data from this memory, or the servers write directly to it. In our implementation, the servers are responsible for sampling representatives from their local buffers and writing them to the memory exposed by the clients. Besides, the exposed buffers are instantiated only once and live for the lifetime of the application, ensuring that the memory is always available for RDMA transfers and avoiding the overhead of frequent memory allocation and deallocation.

4.3.2 RPC Consolidation

With RPC consolidation, multiple representatives can be transferred in bulk from the same remote rehearsal buffer, reducing the number of required RPCs. This is useful in cases where multiple representatives need to be sampled from a given rehearsal buffer. By consolidating these transfers into a single RPC, we can improve overall performance and reduce network overhead. In this way, each client samples r representatives from other processes to prepare its augmented minibatch, but issues a maximum of r RPCs for global sampling. For a given iteration, the maximum number of r RPCs is reached when each representative is sampled from a different rehearsal buffer.

Specifically, the `dispatch_rpcs()` function detailed in Table 4.2 generates a list of r indices corresponding to global indices in the distributed rehearsal buffer. A single RPC is then dispatched in the event that two or more indices correspond to the same remote node. In this scenario, `dispatch_rpc()` transmits to the remote node, in addition to the local indices, an offset corresponding to the client's exposed memory slots it should write into when exploiting an RDMA transfer.

4.3.3 Leveraging Page-locked Buffers for Efficient Data Transfers

In our implementation, rehearsal buffers are stored in host memory, and original minibatches used to accumulate representative training samples are residing on the GPU. Host data allocations are pageable by default, and the GPU cannot access data directly from pageable host memory. Consequently, when a data transfer from device memory to pageable host memory is initiated (or vice versa), the CUDA driver must first allocate a temporary page-locked, or "pinned" buffer [139]. It then copies the device data to this buffer and subsequently transfers the data from the buffer to host memory, where representatives eventually reside. One can avoid the cost of data transfers between device and pageable memory by directly allocating rehearsal buffers in pinned memory.

Similarly, on the server side, the `get_representatives()` procedure returns training samples in bulk using RDMA, writing them directly into an exposed memory buffer on the client side. Such representatives (r in total) are then copied into the augmented minibatch to prepare via `copy_buffer_to_batch()`, as detailed in Table 4.3. As augmented minibatches are preallocated in Python, and already moved onto the GPU, this copy implies a data transfer from the host memory to the GPU. For this reason, exposed memory buffers used for RPC consolidation are pinned for buffering representatives in an overlapping fashion with the application execution.

Conclusion

In this chapter, we presented Neomem, an efficient implementation of rehearsal-based continual learning. The architecture consists of multiple Neomem instances, each acting as both a client and a server, providing services to interact with rehearsal buffers through Remote Procedure Calls (RPCs). Neomem is built on top of the Mochi framework and leverages high-performance network transports, backends, and techniques such as RDMA. We described the general design principles, asynchronous techniques, and optimizations in data movements used in Neomem. Our implementation is publicly available as a C++ open-source project, and we also provide a Python codebase for managing the Continual Learning setting. Neomem achieves efficient buffer management by using multi-threaded concurrency, non-blocking RPCs, asynchronous CUDA copies, and I/O optimizations in data movements. These techniques enable Neomem to provide a high-performance implementation of rehearsal-based continual learning in the context of data-parallel training, which we believe presents an opportunity for specialized data management techniques that are applicable to a broad class of rehearsal approaches. To our best knowledge, such HPC-oriented aspects aimed at improving training performance of continual learning workloads were not explored before.

Chapter 5

Experimental Evaluation using a Synthetic Benchmark

Contents

5.1 Experimental Setup and Methodology **78**
- 5.1.1 Training Dataset & Continual Learning Scenario 78
- 5.1.2 Learning Models 79
- 5.1.3 Training Procedure at Scale 80
- 5.1.4 Performance Metrics 81
- 5.1.5 Computing Environment 82

5.2 Experiments and Results **82**
- 5.2.1 Impact of the Rehearsal Buffer Size on Accuracy 83
- 5.2.2 Impact of Other Rehearsal-related Hyperparameters on Accuracy 84
- 5.2.3 Comparison with Baseline Approaches 84
- 5.2.4 Rehearsal Buffer Management Breakdown 85
- 5.2.5 Scalability Study 87

5.3 Discussion **89**
- 5.3.1 Experience Replay is a Strong Approach 89
- 5.3.2 The Rehearsal Buffer Size Matters 89
- 5.3.3 Rehearsal as a Trade-off Between Incremental and From-scratch Training 90
- 5.3.4 Choice of the Performance Metrics 90
- 5.3.5 The Need for Hyperparameter Optimization 91

In this chapter, we present the experimental validation of our proposal for rehearsal-based continual learning using a synthetic benchmark. Our focus is on evaluating the performance and scalability of our approach for classification problems. We aim to answer several questions related to the impact of rehearsal-related hyperparameters, accuracy degradation, training time, and memory cost of our proposal. To achieve this, we use the medium-scale ImageNet-1K dataset and focus on the class-incremental scenario. We use three different convolutional networks to demonstrate the model-agnostic nature of our approach.

We summarize the contributions of this chapter as follows:

— We present our **experimental setup and methodology** in detail (Section 5.1).

— Next, we present results addressing three research questions related to the impact of rehearsal buffer size, other rehearsal-related hyperparameters, and a **comparison with baseline approaches** for three different neural network architectures (Section 5.2).

— Finally, we discuss our experimental results, focusing on the impact of rehearsal buffer size, hyperparameter search, and efficiency at scale (Section 5.3).

The results illustrate the benefits of a distributed buffer for continual learning in combination with data-parallel training.

5.1 Experimental Setup and Methodology

In this section, we define our continual learning scenario and introduce the performance metrics used in our study, emphasizing the use of larger datasets than typically found in continual learning experiments and the introduction of distribution shifts within this context.

5.1.1 Training Dataset & Continual Learning Scenario

As discussed in Section 2.4.1, we aim to evaluate not only the accuracy but also the training time of models, which is why we utilize parallelization techniques that are particularly beneficial in a distributed context. However, the current literature on Continual Learning primarily focuses on single-node experiments using small-scale datasets such as split-MNIST [66], split-CIFAR [68], or mini-ImageNet [140], which may not accurately represent the complexity of real-world data. As a result, we seek to explore the use of larger and more complex datasets in our study. Over the past decade, ImageNet [141] has been a highly influential benchmark in the advancement of machine learning research. Significant improvements have been achieved in terms of the accuracy of image classification models using it. Thus, we use this medium-scale dataset in our Continual Learning scenario, containing 1.2M training images split among 1000 classes (ImageNet-1K). Specifically, we use the variant with face-blurred images [142]. Each class contains about 1300 training and 50 validation samples. We use standard data augmentations of random horizontal flips and crops resized at 224x224 pixels as done in [143]. This method allows to reduce overfitting by artificially enlarging the dataset using label-preserving, visual transformations. Such augmentations on images require very little computation.

In this dissertation, we focus on the class-incremental ("Class-IL") scenario (several Continual Learning scenarios are discussed in Section 2.2.3), as it is more general and more realistic than other settings [144, 76]. In this scenario, **the model is required to incrementally learn to differentiate between an increasing number of classes**. A typical setup for this scenario involves a sequence of classification-based tasks, with each task containing different classes. The algorithm must learn to distinguish between all classes, without the information of which

task this class was observed on at inference time. Unlike task and domain-incremental learning ("Task-IL" and "Domain-IL" scenarios), where the model is not expected to differentiate between classes from different tasks, this is required in class-incremental learning. One of the main challenges in this scenario is learning to distinguish between classes that are not observed together in the same task, which has proven to be very challenging for deep neural networks, as reported in a comprehensive study about Class-IL scenarios [145]. This challenging setting will allow us to thoroughly evaluate the strengths and weaknesses of our rehearsal-based approach. Thus, we design a sequence of four disjoint tasks, each containing 250 classes from ImageNet-1K. Each task is revisited 30 times, resulting in a total of 120 training epochs, and the model is not allowed to revisit previous tasks i.e., tasks are made available sequentially. The number of tasks in the continual learning scenario is denoted T.

5.1.2 Learning Models

To show that our rehearsal-based approach to CL is model-agnostic, we use the three following convolutional networks and their corresponding configurations:

— **ResNet-50** [104] is a convolutional neural network (CNN) architecture that is widely used for image classification tasks. The ResNet-50 architecture is based on the concept of residual learning, which aims to address the problem of vanishing gradients in deep neural networks (discussed in Section 2.2.2 paragraph 5). The idea is to add *skip connections* between layers, allowing gradients to be directly backpropagated to earlier layers. Overall, ResNet-50 has 1 convolutional layer, 48 identity/convolutional blocks, and 1 fully connected layer, for a total of 50 layers and about 25.6M trainable parameters. We select this DL model for our study because it has long been the standard on the ImageNet benchmark, and is still being improved upon [146, 147]. Besides, the ResNet architecture exhibits a high compute to memory ratio thanks to relatively small activations (computation is often memory-bound in modern datacenters [148]). This allows for fast training iterations [146], providing an excellent opportunity to test the performance of the distributed buffer, giving it less time to prepare augmented minibatches.

— **ResNet-18** [104] is a CNN architecture that is a smaller version of the ResNet-50 architecture. Overall, ResNet-18 has 18 layers, including 16 convolutional layers, 1 max pooling layer, and 1 global average pooling layer. ResNet-18 has roughly half the number of parameters of ResNet-50 and is thus faster to train (i.e., its minibatch processing time is shorter). The parameter count does not always determine memory usage during training since the size of the activations often dominates the memory consumption. Nonetheless, ResNet-18's smaller parameter count still leads to faster training times and reduced memory usage compared to ResNet-50.

— **GhostNet-50** [149] is a lightweight CNN architecture designed to be computationally

efficient while still achieving high accuracy on image classification tasks. The GhostNet architecture is based on the concept of Ghost module, which is a more recent type of convolutional layer that generates more feature maps from cheap operations. The Ghost module consists of two parts: a primary convolutional layer and a set of cheap linear transformations. Overall, GhostNet-50 has 50 layers, including 1 convolutional layer, 48 Ghost modules, and 1 fully connected layer. We choose this DL model to show that the benefits of the distributed buffer are model agnostic.

We use the cross-entropy loss as each task is a 250-way classification problem.

5.1.3 Training Procedure at Scale

As extensively discussed in Section 2.3.2, in the minibatch SGD algorithm, the input data is divided into minibatches containing b samples each. In a data-parallel setting involving p training processes, the effective batch size $B = p \times b$ represents the global batch size, resulting from the summation of local minibatches. The main challenge with data parallelism is managing large effective minibatch sizes, as they can impact the model's ability to generalize well to unseen data, as noted in [107]. Using ResNet-50 and the ImageNet-1K dataset, authors in [108] have successfully increased the effective batch size to 8K samples by implementing various adjustments to the training procedure. We incorporate three of their key strategies into our study: (R1) the learning rate η is scaled with the minibatch size b; (R2) a learning rate (LR) schedule is used to decrease the LR over time; and (R3) an appropriate warmup strategy is implemented as part of the LR schedule to avoid optimization difficulties during the early stages of training.

Specifically, we address (R1) by applying the linear scaling rule [108] stating that the learning rate should be multiplied with the number of processes p. For a minibatch size set to $b = 56$ and an augmented minibatch size set to $b + r = 63$ training samples, this rule yields to an effective batch size of $B = p \times 63$ in a data-parallel setting. Thus, the latter becomes greater than 8K with $p = 128$. This requires further consideration to mitigate the instability introduced by such large batches [108, 150]. We do so by setting a maximum scaling factor (independent of the number of processes) equal to 64, as suggested theoretically in [151]. Regarding (R2), a unique aspect of our continual learning scenario is the requirement to learn from four sequential tasks. This necessitates to accommodate the sequential nature of the tasks. Empirically, we determinate that applying the LR schedule independently to each new task —effectively restarting the schedule from scratch for each task— yields optimal results. As a result, the LR is decayed several times during training, so as to draw several plateaus where the LR remains constant for several epochs. Finally, we address (R3) by gradually increasing the LR over the first 5 epochs of each task, progressively approaching the target "base" LR η. Following the warmup period, we revert to the original LR schedule as detailed in (R2).

We sum up the training procedure as follows, recalling that the DL model learns by per-

forming 30 epochs on each task:

— **ResNet architecture**: we use the SGD optimizer with a learning rate $\eta = 0.0125$; a per-task learning rate increase on 5 warmup epochs as in [108]; a gradual decay it from 0.5 to 0.05 to 0.01 at epochs 21, 26 and 28 respectively; and a weight decay of 1e-5.

— **GhostNet architecture**: we use the SGD optimizer with a learning rate $\eta = 0.01$; the same warmup as ResNet's; the same schedule at epochs 15, 21, 28; and a weight decay of 1.5e-5.

We enable Automated Mixed Precision (AMP) introduced in [152] to speed up the training. This optimization has a direct impact on the evaluation of our distributed buffer, as individual training iterations are shortened, giving it less time to prepare augmented minibatches.

5.1.4 Performance Metrics

When assessing performance in a continual learning scenario, two critical questions must be addressed: (Q1) *how* to evaluate the accuracy of the model being trained, and Q2) *when* to evaluate it. Regarding (Q1), continual learning performance is typically evaluated using the average evaluation accuracy over the sequence of past and current tasks trained thus far. To determine the evaluation accuracy achieved by the model, we employ a validation set, which is a subset of the dataset not seen during training. Regarding (Q2), the timing of the evaluation, one approach is to assess the model only at the conclusion of training on all tasks. However, this method obscures the specific advantage of continual learning, which is the ability to have a model ready to run inferences on classes already observed at any point during training. Therefore, a more preferred approach is to evaluate performance periodically throughout the training procedure by interleaving training and evaluation stages. For instance, performance can be evaluated after completing training on each task, after every epoch, or after a fixed number of training steps i.e., multiple times per epoch.

For our performance evaluation, we opt to use the top-5 evaluation accuracy metric, reported after each training epoch on the validation set. Top-5 accuracy means any of the model's top 5 highest probability predictions is considered as correct. Let $a_{j,t}$ denote the top-5 evaluation accuracy achieved on task j when using the current snapshot of the model while learning task t, with $t \geq j$. The accuracy (fraction of correct classifications) assessing the DL model performance on all previous tasks is defined as follows:

$$acc_t = \frac{1}{t} \sum_{j=1}^{t} a_{j,t} \tag{5.1}$$

When training on task t, value acc_t can be calculated many times if the evaluation performance is reported after each training step or epoch, and doesn't change anymore when the training procedure proceeds with task $t+1$. In other words, computing acc_t more often than

once at the end of the current task t would produce a list of accuracy values A_t represented as:

$$A_t = [acc_t^1, acc_t^2, ..., acc_t^n] \tag{5.2}$$

Here, n denotes the total number of times the accuracy is reported during the training on task t i.e., $n = epochs$ if the metric is reported after each epoch. It is worth noting that the ML community working on computer vision usually relies on the top-1 (instead of top-5) evaluation accuracy. However, we are primarily interested in showing that accuracy is greatly improved by leveraging Experience Replay, rather than showing absolute figures tied to a specific use-case (e.g., our sequence of four tasks). Thus, any accuracy metric is deemed appropriate. More broadly, the novelty of our work lies in the techniques to make rehearsal scalable in the context of data-parallel training.

Reporting evaluation accuracy is necessary to quantify model quality. However, we are also interested in the benefits brought by continual learning and parallelization techniques in terms of training time. We therefore report it in minutes. In particular, we want to examine whether our distributed buffer slows down the training procedure.

5.1.5 Computing Environment

In order to conduct our experiments, we utilize up to 16 nodes of the ThetaGPU supercomputer at Argonne National Laboratory, providing a total of 128 NVIDIA A100 GPUs, each with 40 GB of memory. Each node is equipped with two AMD Rome CPUs. The software environment used in our study includes Python 3.10, PyTorch 1.13.1, Horovod 0.28.1, CUDA 11.4, NVIDIA DALI 1.27.0, OpenMPI 4.1.4, Mercury 3.3, and libfabric 1.16, with CUDA support enabled during compilation.

5.2 Experiments and Results

Our series of experiments focuses on evaluating the performance and the scalability of our proposal for classification problems. To this end, we aim to answer the following questions:

- How do parameters r (representative count used to augment minibatches) and $|\mathcal{B}_n|$ (local rehearsal buffer size) impact the achieved classification accuracy?
- How much does accuracy degrade with CL, compared to the case where the model relearns from scratch each time new data arrives?
- Do minibatch augmentations and rehearsal buffer management increase training time?
- How much does training time increase, compared to incremental training?
- What is the memory cost of rehearsal-based learning?

5.2.1 Impact of the Rehearsal Buffer Size on Accuracy

As detailed in Section 3.1.2, distributing the training across p processes allows to leverage the aggregated memory to store more representatives in the rehearsal buffer of size $|\mathcal{B}| = p \times |\mathcal{B}_n|$. Sampling representatives globally allows to distribute a certain percentage of the input dataset over all processes (e.g., storing 10% of the input dataset means in practice storing $10\%/p$ of the data per training process). Besides, as detailed in Section 3.1.3, we recall that representatives are distributed among memories according to their class labels. Thus, candidate samples belonging to a specific class compete against the existing representatives of the same class to be retained in the buffer. In particular, this means that storing a certain proportion of the entire dataset in the buffer involves storing the same proportion of representatives for each class.

To showcase the effect of different rehearsal buffer sizes on the accuracy, we vary $|\mathcal{B}|$ from 2.5%, 5%, 10%, 20%, to 30% of the total number of ImageNet data samples (1.2M images). These values correspond respectively to 1.93 GB, 3.85 GB, 7.71 GB, 15.41 GB and 23.12 GB of raw data stored in the aggregated memory.

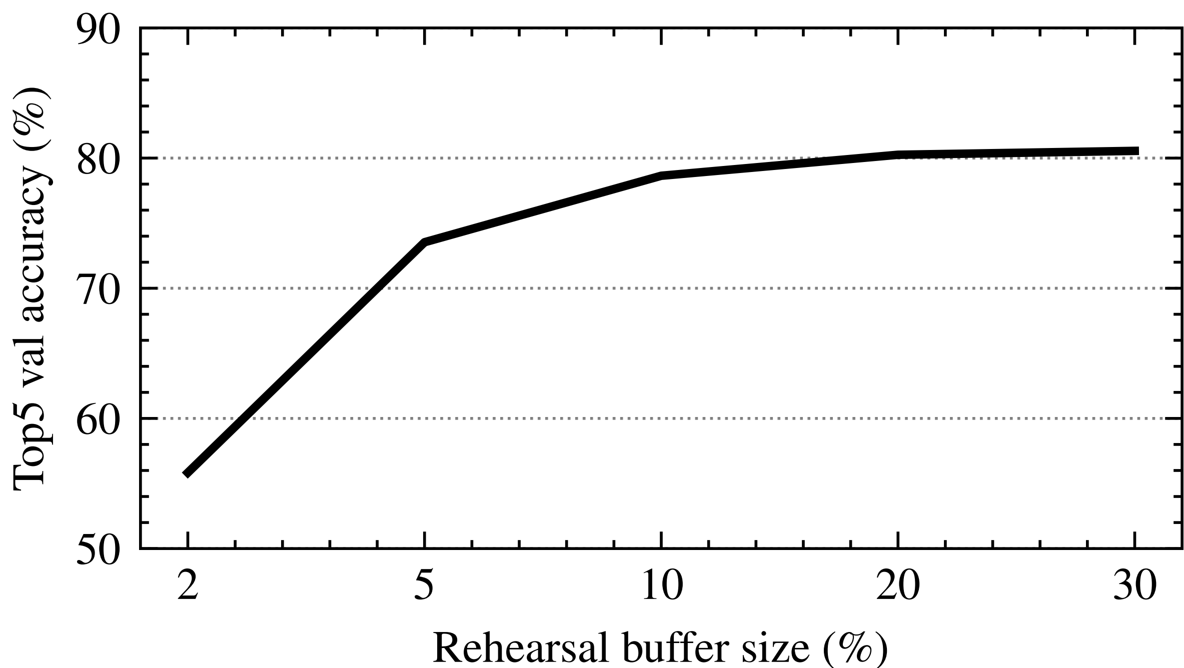


Figure 5.1 – Accuracy w.r.t. different rehearsal buffer sizes $|\mathcal{B}|$ (percentage of the input dataset). Each data point is the average of the top-5 accuracy obtained on all previous tasks.

We measure the performance of our approach with different rehearsal buffer sizes by applying Equation 5.1 once at epoch 120 (conclusion of the training), in order to evaluate the DL model on all previous tasks i.e., on all the classes seen until then. We only consider ResNet-

50 for this study, and run these experiments on 2 nodes (16 GPUs). We report the results in Figure 5.1. As expected, the larger the rehearsal buffer size $|\mathcal{B}|$, the better the diversity among stored representatives. As a result, the model forgets less knowledge acquired in previous tasks, resulting in a higher final accuracy. In our setting, storing 30% of the input data samples as representatives yields to a final top-5 accuracy of 80.55%, which is significantly better than the accuracy achieved with $|\mathcal{B}| = 2.5\%$ (55.83%). We emphasize that storing 30% of ImageNet samples amounts to storing 1.45 GB of raw data per process (with $p = 16$), which is only a fraction of the memory available on typical HPC systems (512 GB of host memory per compute node). We set $|\mathcal{B}| = 30\%$ in the remainder of this chapter.

5.2.2 Impact of Other Rehearsal-related Hyperparameters on Accuracy

The number of training samples used to populate the rehearsal buffer at each iteration (parameter c introduced in Section 3.1.3) is less relevant in class-incremental scenarios, as: (1) classes from different tasks are disjoint, and (2) the competition to populate the buffer is done per class. As a result, representatives from previous tasks never get evicted under this setting. We set $c = 14$, which in our experimental setup only affects the renewal rate of representatives from the current task.

The number of representatives sampled from the rehearsal buffer (parameter r introduced in Section 3.2) has a direct impact on the balance between plasticity and stability, where the model should be both plastic enough to learn new concepts, and stable enough to retain knowledge. Mixing too many representatives with incoming minibatches decreases the ability of the DL model to learn the current task, resulting in a degraded accuracy. A larger value for r favors stability over plasticity. Following an exploration of the hyper-parameter space, we identified the optimal values for maximizing accuracy, namely, a minibatch size of $b = 56$ and a representative count of $r = 7$.

5.2.3 Comparison with Baseline Approaches

The following baselines instantiate models without any rehearsal of representatives ($r = 0$):

- **Incremental training**: updates the model with the training data corresponding to a single task, one at a time. No training samples of any previous tasks are revisited.
- **Training from scratch**: re-trains the model from scratch at every new task, using all accumulated training samples of both the new task and the previous tasks.

The minibatch size is set to $b = 56$, and we use ResNet-50 in this study. In Figure 5.2, the top-5 evaluation accuracy achieved by rehearsal (80.55%) greatly outperforms the incremental training baseline. We apply Equation 5.2 for each task ($T = 4$) with $n = 30$ as 30 epochs are performed on task t. Incremental training suffers from catastrophic forgetting and is regarded

as the lower bound accuracy-wise (23.3%). On the opposite, training from scratch as new data arrives is regarded as the upper bound (91%), only about 10.5 percentage points above the accuracy achieved with our rehearsal-based approach. This represents a clear improvement over incremental training, and comes very close to from-scratch training.

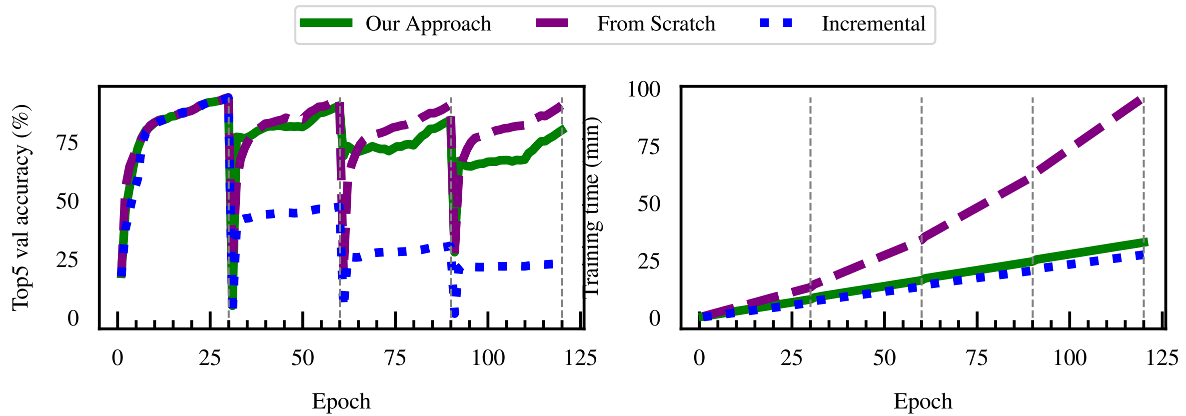


Figure 5.2 – $|\mathcal{B}| = 30\%$ and $r = 7$. Left: accuracy w.r.t. epoch number. Our rehearsal-based approach achieves a final accuracy of 80.55%. Right: training time w.r.t. epoch number. Our approach induces a small runtime increase compared with incremental training, which stays linear.

Figure 5.3 – Top-5 accuracy for ResNet-50, 16 GPUs, ImageNet (4 tasks).

In Figure 5.2, we observe that incremental training has the shortest runtime as no task gets revisited (lower bound). On the other hand, the duration of training from scratch increases cubically with the number of tasks to learn T (sum of the first T triangular numbers). This is noticeable as a large gap between the two approaches as the number of tasks increases. Just like incremental training, our rehearsal-based approach exhibits a linear runtime with just a slight increase proportional to the r additional samples added to the minibatch. We demonstrate in the next section that this overhead is not introduced by the rehearsal buffer management itself. Thus, we conclude that our approach represents a substantial improvement over incremental training and approaches the performance of from-scratch training, demonstrating the effectiveness of our method in balancing between the two baselines. This allows to overcome the limitations of the baseline approaches, which sacrifices accuracy for scalability or vice versa.

5.2.4 Rehearsal Buffer Management Breakdown

In Figure 5.4 we examine the time taken for the individual operations within a training iteration. This study allows us to understand how well our approach overlaps the rehearsal buffer management with the actual training procedure.

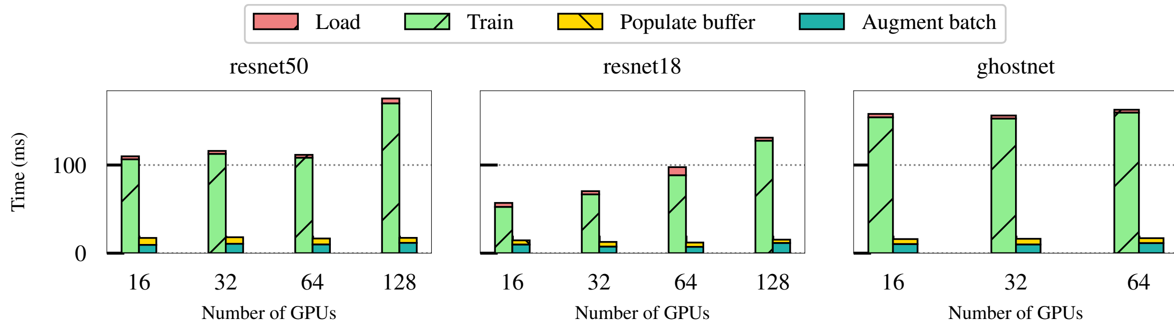
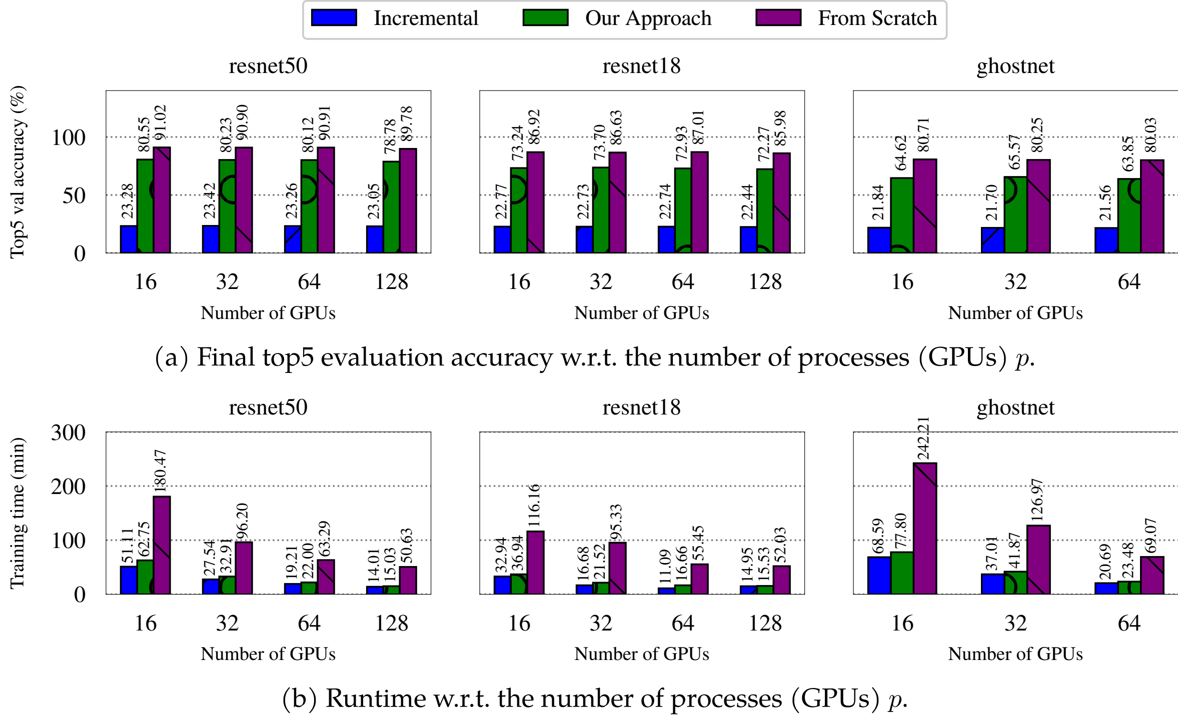


Figure 5.4 – Time breakdown for the training loop and rehearsal buffer management, for each of the three models and for different number of GPUs, averaged across 35 minibatches. Training iterations sometimes take longer as the time required for communication and synchronization between GPUs increases with their number. Non-linearity can be explained by the topology of GPU interconnects, as well as other factors such as communication overhead and resource contention.

Specifically, we measure the time taken to obtain a new minibatch from DALI (denoted *Load*), which itself uses an asynchronous data pipeline that prefetches and shuffles the training data. Then, we measure the duration of the forward and backward passes as reported by PyTorch (denoted *Train*). The time taken for *Load* followed by *Train* is the lowest possible overhead perceived by the application; this time is represented by the stacked bars on the left of each of the 11 pairs of data bars in Figure 5.4. In the background, our approach handles updates to the individual rehearsal buffers (denoted *Populate buffer*), the distributed sampling of the remote rehearsal buffers, and the minibatch augmentation (denoted *Augment batch*); this time is represented by the right-hand stacked bars in the figure. As long as the stacked bars on the right are lower than those on the left, our approach will not cause the training iterations to wait for the augmented minibatches. This means there is a full overlap and the rehearsal buffer management is completely hidden in the background thanks to our asynchronous techniques.

Indeed, we observe that this condition holds for all models and all scales used in our experiments i.e., the duration of managing the rehearsal buffer is constant in all settings. Furthermore, the total overhead of our approach is just a fraction of the *Load* and *Train* overheads. Since the *Train* overhead dominates (thanks to DALI's asynchronous data pipeline), we conclude that there is a large margin left to optimize the forward and backward passes without reducing the effectiveness of our approach.

Another interesting effect is visible: we cannot simply reduce the duration of the forward pass and backward pass at scale by optimizing the computations. When we switch from ResNet-50 to ResNet-18, which is significantly less computationally expensive to train, the duration of *Train* increases because `allreduce` gradient reductions are expensive and begin to stall the



(a) Final top5 evaluation accuracy w.r.t. the number of processes (GPUs) p.

(b) Runtime w.r.t. the number of processes (GPUs) p.

Figure 5.5 – Accuracy and runtime, $|\mathcal{B}| = 30\%$, $b = 56$ and $r = 7$ for all 3 models. For ResNet-50, colors match those in Fig. 5.2.

computations. This observation illustrates that the overall training time does not decrease proportionally with the number of GPUs. However, in a data-parallel setting, the amount of data processed at every iteration is proportional to the number of GPUs. This means that as the number of GPUs increases, the amount of data processed per iteration also increases, which leads to a decrease in the number of iterations required to achieve the same level of accuracy. As long as the decrease in the number of iterations is greater than the increase in the duration per iteration, there will be a speed-up in the training time. Thus, based on the observed trends, we hypothesize that our approach remains effective at scale even in extreme cases of computationally trivial models.

Please note that the durations obtained in this experiment are not indicative of the entire training procedure but rather reflect a subset of minibatches observed during the first epoch. While this allows us to conclude that buffer management does not impede training, one can not predict total training time from this observation made at very low level.

5.2.5 Scalability Study

We study the scalability of our approach for all three models compared with the two baselines for an increasing the number of data-parallel processes (GPUs). Specifically, we measure

the final evaluation top-5 accuracy for an increasing number of processes in Figure 5.5a, where Equation 5.1 is applied once at epoch 120. We also measure the overall runtime to train all tasks and depict it in Figure 5.5b.

All three approaches retain similar accuracy for an increasing number of processes. Since incremental training and training from scratch make direct use of data parallelism, this finding is not surprising. On the other hand, the same trend is visible for our approach, which demonstrates that it applies global sampling correctly at scale and therefore avoids potential biases. Furthermore, all approaches exhibit shorter runtimes for increasing numbers of data-parallel processes. What is interesting to observe though is that the gap between our approach and incremental training does not increase with the number of processes. Instead, the gap is decreasing, which shows that our approach is scalable and can successfully overlap the asynchronous updates of the rehearsal buffer and the global sampling with the training iterations, even when the all-to-all communication complexity increases, as we noted in Section 5.2.4.

Visually on Figure 5.5b, one can notice that the *scaling efficiency* exhibited by ResNet-18 is inferior to that of ResNet-50. Specifically, the decrease in training time expected when using more GPUs is less pronounced for ResNet-18. In fact, training ResNet-18 for 120 epochs on 128 GPUs requires more time than training ResNet-50 under identical conditions. This phenomenon can be attributed to several factors, independent of rehearsal:

— As illustrated in Figure 5.4, the duration of individual training iterations increases with the number of processes due to longer communication times required by the synchronization of a greater number of model replicas. The communication overhead incurred by gradient exchange becomes a significant bottleneck at scale, and is detrimental to scaling efficiency. For smaller models like ResNet-18, the ratio of communication to computation time is higher. Although Figure 5.4 depicts only minibatches of the first epoch and is not representative of the entire training procedure, it can be observed that the average duration of an individual iteration of ResNet-18 approaches that of ResNet-50 when using 128 GPUs.

— When communication time dominates the duration of an iteration, it can induce a stalling effect on computation. In such cases, the suboptimal overlap of communication and computation results in extended training times.

— ResNet-50 is a deeper and more complex model compared to ResNet-18. It has more layers and parameters, which means it has a higher computational load per minibatch. This higher computational load can better leverage the parallel processing capabilities of multiple GPUs, leading to better GPU utilization and scaling efficiency.

Thanks to the asynchronous rehearsal management, the average training time of our approach is only determined by the r additional representatives assembled into augmented minibatches at every iteration, as shown in Section 5.2.4. Thus, the runtime of rehearsal-based CL is

roughly proportional to r.

5.3 Discussion

In this section, we draw conclusions from our evaluation and highlight the strengths of the rehearsal-based approach to continual learning.

5.3.1 Experience Replay is a Strong Approach

Only by replaying a subset of training samples assembled into the augmented minibatches ($r = 7$), our rehearsal-based approach significantly surpasses the incremental training baseline by achieving a top-5 evaluation accuracy of 80.55%, approaching the upper bound of 91% attained by training from scratch as new data arrives. Rehearsal seems to prevent the DL model from leaving the first found low-loss region, effectively mitigating catastrophic forgetting. Despite its conceptual simplicity, we conclude that Experience Replay is a very promising direction for large scale CL. Similar findings were reported by [72] but on small-scale CL datasets like split-CIFAR and mini-ImageNet, which are less realistic and less diverse. Authors in [73] draw the same conclusion on the larger Taskonomy dataset [153]. The benefits of Experience Replay on the accuracy are tangible even on very small rehearsal buffers [72], when they are populated with a single representative per class.

5.3.2 The Rehearsal Buffer Size Matters

The two studies cited above [72, 73], among others, already made the unsurprising observation that a larger rehearsal buffer leads to a better achieved accuracy. Our results confirm this observation. The study varied the rehearsal buffer size from 2.5% to 30% of the total number of ImageNet data samples and found that storing 30% of the input data samples as representatives yielded a final top-5 accuracy of 80.55%, which is significantly better than the accuracy achieved with a smaller buffer size. A larger buffer size $|\mathcal{B}|$ enables a more diverse and extensive representation of previously encountered tasks.

Storage of a significant part of a large dataset like ImageNet is made possible by our distributed buffer. The accumulation of representatives may grow to very large sizes, but our approach aggregates the free memory on the compute nodes in a scalable fashion. Specifically, given only a fraction of the host memory on each compute node (1 GB in our experiments), our approach was capable of storing 30% of the ImageNet dataset even at medium scale (128 GPUs). Furthermore, this amount of free memory can be calculated in advance as it is bounded w.r.t. the number of classes K and many additional data reduction techniques can be applied if necessary (e.g., compression). Authors in [82] suspect that training repeatedly over a limited

number of representatives would lead to overfitting the rehearsal buffer, which may be an inherent limitation of CL by being harmful for generalization. Indeed, large DL models are capable of memorizing small amounts of data, such as those in rehearsal buffers retaining a limited number of representatives, without developing any generalization capabilities [129]. In this regard, our approach of using a distributed rehearsal buffer allows the aggregated size to grow proportionally with the number of training processes. While this approach does not eliminate the problem of overfitting per se, it mitigates the issue by enabling retention of a larger and more diverse set of representatives. This increases the quality of continual learning in combination with data-parallel training, pushing the limits acknowledged by other state-of-the-art approaches.

5.3.3 Rehearsal as a Trade-off Between Incremental and From-scratch Training

Rehearsal-based continual learning strikes a balance between incremental training and training from scratch, with hyperparameters $|\mathcal{B}|$ and r playing a crucial role in setting this trade-off. Both the size of the rehearsal buffer and the number of representatives assembled into augmented minibatches determine the degree to which the model relies on previous knowledge versus adapting to new information. In the extreme case where the buffer size $|\mathcal{B}|$ is set to infinity and minibatches are augmented with all the buffer content throughout a given epoch, the rehearsal-based approach would tend towards the performance of training from scratch. However, even under these conditions, rehearsal does not fully replicate training from scratch. In rehearsal, the diversity of representatives replayed increases as the buffer populates during training, whereas in training from scratch, the model has access to the entire dataset from the outset. Consequently, rehearsal alters the learning dynamics, with a higher probability of sampling the same representatives frequently early in training when the buffer contains fewer elements.

5.3.4 Choice of the Performance Metrics

We chose the top-5 evaluation accuracy averaged over all past tasks to assess the performance of our approach. As noted in [67], comparing continual learning methods solely based on such average performance can be misleading, as it fails to capture how well catastrophic forgetting is mitigated. For instance, a rigid model that achieves 100% accuracy on the first task and 0% accuracy on the second task has the same average performance as a plastic model with 0% accuracy on the first task and 100% accuracy on the second task. To gain deeper insights into the dynamics of continual learning, other performance metrics have been proposed. We plan to report them in future work.

One such metric is the *performance drop* [73] on task t, which aims at measuring plasticity of the model by quantifying its performance gap with the model obtained with from-scratch

training on all tasks, and then evaluated on task t. Another widely-used metric is *backward transfer* [19], which measures how the performance on previously learned tasks changes when training on new ones. In particular, a positive backward transfer is achieved when the model improves the performance on a previous task j when learning task t, with $t \geq j$. This metric provides insights into the degree of catastrophic forgetting and the stability of the model: a large negative backward transfer yields to more catastrophic forgetting. This measure is sometimes referred to as *forgetting* [73], which represents the average loss increase of the model when evaluated on previous tasks, when compared to the performance that was obtained at the time of learning such tasks. Additionally, *forward transfer* [19] is a metric that quantifies how much training on previous tasks improves the performance of a model on future tasks. In particular, a positive forward transfer is achieved when the model is able to perform *zero-shot* learning [154].

5.3.5 The Need for Hyperparameter Optimization

To achieve optimal performance, it is essential to perform an hyperparameter search to find the best combination of parameters such as minibatch size b, LR schedule, or weight decay. The cost of computation required for this procedure is an important factor that should be reported. We carried out this procedure using 8 GPUs of the Grid'5000 testbed [155]. Once model convergence has been ensured in this small-scale setting, we simulated a gradual transition to larger scales using *gradient accumulation* [156]. This technique allows the model to perform multiple backward passes before updating its parameters all at once for multiple minibatches. This is unlike the conventional manner where the model parameters are updated once every minibatch, simulating the larger effective batch size that would result from training on more processes. To simulate training on $p = 128$ processes from 8 physical GPUs, 16 accumulation steps are required. This stage took about 1500 GPU-hours using NVIDIA V100s.

Our study found that the rehearsal-related hyperparameter $|\mathcal{B}|$ had a significant impact on the achieved accuracy. However, 8 GPUs do not allow to accumulate more than 1% of the ImageNet dataset. We therefore launched a new hyperparameter search for b and r with the buffer size $|\mathcal{B}|$ set to 1%. This stage took about 400 GPU-hours. Then, we were able to run our experiments on ThetaGPU (a machine with more memory, benefiting our distributed buffer) to demonstrate the advantages of a larger buffer as illustrated in Figure 5.1. The hyperparameters resulting from this search procedure have been used in Figures 5.5a and 5.5b.

The discovered hyperparameters generalize to ResNet and GhostNet architectures, showing that our approach is applicable to different DL models. Although we have not run any experiments to confirm this, we suspect that the optimal value of rehearsal-related hyperparameters depends strongly on the number of tasks T to be learned.

Conclusion

In our proposal, we aim to address the challenges of continual learning in the context of data-parallel training by designing and implementing a distributed rehearsal buffer. Our approach handles the selection of representative samples and the augmentation of minibatches asynchronously in the background, enabling efficient and scalable training on HPC systems. To evaluate the performance and scalability of our proposal, we conduct a comprehensive series of experiments focusing on a classification problem. We utilize up to 128 GPUs of the ThetaGPU supercomputer to compare our approach with two baselines: from-scratch training, which serves as the upper bound in terms of accuracy, and incremental training, which represents the lower bound. As a notable result, in the best case with ResNet-50, our method improves the top-5 classification accuracy from 23.1% to 80.55%, with just a small runtime increase—an ideal trade-off that combines the best of both baselines used in the comparison. In addition, we ensure that buffer management does not impede training iterations. We conclude that Experience Replay is a strong approach to mitigating catastrophic forgetting, with applicability across various model architectures and training scales. Furthermore, while there is a positive correlation between the distributed buffer's aggregated capacity and achievable accuracy, the latter plateaus after reaching a certain memory threshold.

CHAPTER 6

TOWARDS A GENERIC DISTRIBUTED REHEARSAL BUFFER

Contents

6.1 Decoupling Buffer Management from the Learning Task 93

6.1.1 Supporting More Rehearsal Strategies and Learning Tasks 94

6.1.2 Extending Rehearsal with Additional States 94

6.2 Illustration Leveraging Knowledge Distillation 95

6.2.1 Integrating Dark Experience Replay 95

6.2.2 Integrating Dark Experience Replay++ 97

In this chapter, we explore the concept of generic rehearsal buffers that can store, sample, and replace heterogeneous data independent from both the learning task and the rehearsal strategy. We aim to create a buffer that seamlessly integrates with the data pipeline and can exchange fine-grain data (as tensors) a global level using scalable distributed techniques to avoid biases introduced by localized sampling. The chapter is organized into two main sections:

— We discuss the importance of decoupling the rehearsal buffer from the learning task and present an extension, enabling it to store **heterogeneous data** and serve it in the form of **annotated tuples** of tensors (Section 6.1).

— Then, we demonstrate the generality of our approach by integrating functional regularization algorithms based on **knowledge distillation**. As such, we implement Dark Experience Replay and Dark Experience Replay++ [25] and discuss the improvements offered over vanilla Experience Replay (Section 6.2).

6.1 Decoupling Buffer Management from the Learning Task

In this section, we show how enabling the rehearsal buffer to store heterogeneous data, and serve it in the form of annotated tuples of tensors, allows for flexible integration with different rehearsal and functional regularization strategies.

6.1.1 Supporting More Rehearsal Strategies and Learning Tasks

Thus far in this dissertation, we have designed the rehearsal buffer to store representatives organized by class, in the form of input-label (x, y) tuples. Besides, the validation of our approach in Chapter 5 was carried out on a classification problem, leveraging Experience Replay as the rehearsal strategy. We have identified two main levers to make our proposal support more deep learning use cases and continual learning approaches: (1) allow the enrichment of tuples (x, y) exposed to the AI runtime with additional state information in order to accommodate more rehearsal strategies, and (2) make the per-class management of representatives more flexible to support more learning tasks like generative workloads, in which the output space is flat i.e., all outputs belong to a single class.

In the light of these new elements, we redefine the main properties needed for the distributed rehearsal buffer: (1) the rehearsal buffer needs to store, sample and replace heterogeneous data that is independent from the rehearsal strategy (e.g., storing additional state information together with the training samples), and serve it conveniently to the training loop using annotated tuples; (2) the buffer's internal data layout should be flexible enough to accommodate different learning tasks (e.g., separate per-class management of training samples in the case of classification tasks vs. unified management of training samples in the case of generative tasks). Performance optimizations introduced to date should be leveraged too: (3) the rehearsal buffer needs to seamlessly integrate with the data pipeline responsible for asynchronously reading the training data and feeding it to the training iterations, which implies the need to transparently overlap the management of the rehearsal buffer with both the data pipeline and the training iterations; and (4) it is not enough to simply instantiate independent rehearsal buffers associated with each DL model replica to enable data parallelism, as extensively discussed in Section 3.2.1. Scalable distributed techniques are needed to enable rehearsal buffers to collaborate at global level in order to avoid biases introduced by localized sampling.

6.1.2 Extending Rehearsal with Additional States

With the growing diversity of rehearsal techniques discussed in Section 2.2.4, it becomes important to decouple the rehearsal buffer from the learning task, such that it becomes a generic abstraction that can store additional state information as needed by more advanced continual learning algorithms. Besides, as discussed in Section 2.2.4, the line between rehearsal and functional regularization algorithms is blur, as the latter can sometimes be seen as a variant of the former. Instead of replaying raw inputs alongside the corresponding label, functional regularization leverages past anchor points labeled with the corresponding predictions as made by a previous version of the model. Such predictions can be stored in the rehearsal buffer as additional states. For instance, the Hindsight Anchor Learning (HAL) [88] algorithm complements

Experience Replay with regularization to align the model responses with representatives encoding classes encountered in previous tasks.

Additional information is stored in separate local buffers, using the same structure as for representative training samples. Conceptually, this amounts to having two (or more) rehearsal buffers per processing unit n. Let $\mathcal{B}_{n,i}$ denote the i-th rehearsal buffer for processing unit n, where $i = 1, 2, ..., k$. This design provides efficient access to representatives and associated states under concurrency, both locally and remotely, reusing all optimizations developed in Chapter 4. We extend the rehearsal buffer presented in Section 3.1 to serve heterogeneous data in the form of **annotated tuples** of tensors i.e., additional state data in buffers $\mathcal{B}_{n,i}, i \geq 2$ enrich tuples $(x, y, \ldots)$ exposed to the AI runtime. Such tensors to be served in an annotated tuple are transferred in separate bulks, albeit all attached to the same RPC. This optimization limits the number of RPCs issued by each training process to r, independently from the number of components in the annotated tuple. In addition, the asynchronous CUDA copies produced for each additional state are carried out in dedicated CUDA streams.

6.2 Illustration Leveraging Knowledge Distillation

In this section, we present an enhanced distributed rehearsal buffer and integrate an Experience Replay strategy leveraging knowledge distillation, Dark Experience Replay [25].

6.2.1 Integrating Dark Experience Replay

Instead of replaying raw training data samples as representatives, some variants of rehearsal leveraging *knowledge distillation* have been explored in the CL literature. The authors in [25] propose Dark Experience Replay (DER), an algorithm applying the teacher-student approach [157] to encourage the DL model to mimic the *neural activations* triggered when learning previous tasks. Activations refer to the raw, unnormalized output values produced by the neural network's final layer, typically a fully connected layer. These activations (also referred to as *logits*), denoted a, are then passed through an activation function, such as the `softmax` function, to obtain the probabilities for each class in a classification task.

This state information a is stored in the rehearsal buffer alongside the associated representative sample x. Activations cannot be dissociated from the sample that produced them. During the training procedure, differently from vanilla Experience Replay, DER does not sample input-label (x, y) tuples from the rehearsal buffer but instead input-activation (x, a) tuples corresponding to previous tasks. In that sense, DER still benefits from the transparent global sampling of representatives detailed in Section 3.2. However, past data samples are not reinjected into the training process through augmented minibatches; past and current activations are simply compared when calculating the loss to promote their consistency, thus benefiting from the

knowledge encapsulated by the former (which acts as a teacher). Using the same notation as in [25], we indicate the neural activations with $h_w(x)$ and $f_w(x) \triangleq \text{softmax}(h_w(x))$. The loss function is extended with an additional distillation term minimizing the Euclidian distance between past activations a and current activations $h_w(x)$ for the associated representative sample x:

$$\mathcal{L}_{t_c} + \alpha \cdot \mathbb{E}_{(x,a)\sim\mathcal{B}} \left[\|a - h_w(x)\|_2^2 \right] \tag{6.1}$$

where α is a hyper-parameter introduced by DER setting the importance of the distillation term. Note that no rehearsal is performed with DER i.e., no backward pass is performed on past representatives. However, sampling activations associated to a past representative x from the distributed rehearsal buffer is still required, as they are compared with activations computed for x using the current model w. As such, the training procedure is not modified beyond a slight adjustment to the loss function, which is complemented by the regularization term from Equation 6.1.

Intuitively, activations a corresponding to a past representative x become outdated while learning new tasks, as (1) they are potentially different from those observed at the task's local minimum, and (2) backward transfer could impact past tasks positively in some cases (positive and negative backward transfers were discussed in Section 5.1.4). However, the authors of [25] observed that this selection strategy occurring along the optimization trajectory i.e., during the training procedure does not degrade the achieved accuracy. When using the selection policy of representatives introduced in Section 3.1.3, c candidate input-activation tuples are pushed into the rehearsal buffer at every training iteration, with the competition still being done per class. In CL scenarios where some classes are observed across multiple tasks, this allows to update activations gradually over time.

Algorithm 5: Training loop implementing Dark Experience Replay (adapted from the original implementation [25]).

1 $\mathcal{B} \leftarrow \{\}$
2 **for** $k = 0$ ***to*** $\frac{|D|}{b} \cdot epochs$ **do**
3 $(x, y) \leftarrow \vec{m} \leftarrow$ `sample(D,b)` // Obtain b samples from dataset D
4 $a \leftarrow h_w(x)$ // Compute output activations
5 $(x', y', a') \leftarrow$ `get_representatives(`$\mathcal{B}, r$`)` // Obtain r samples from $\mathcal{B}$
6 $dis \leftarrow \alpha \cdot \|a' - h_w(x')\|_2^2$ // Compute distillation term
7 $g_m \leftarrow \nabla[\ell(f_{w^{(k)}}(x), y) + dis]$ // Compute gradient using backpropagation
8 $w^{(k+1)} \leftarrow w^{(k)} + u_{GD}(g_m)$ // Weight update rule
9 $\mathcal{B} \leftarrow$ `update_buffer(`$(x, y, a), c$`)` // Update buffer using current activations

Algorithm 5 describes the training procedure as proposed by DER. Variable dis denotes

the regularization term added by DER. After sampling a minibatch m from dataset D (line 3), the corresponding neural activations a are computed (line 4). Then, an input-activation tuple is sampled from the distributed rehearsal buffer (line 5), and the distillation term dis is computed using the current model parameters w (line 6). An augmented minibatch m' is also prepared for rehearsal as detailed in Section 3.2. Finally, model parameters are updated considering the loss obtained from m' and dis (line 8) and the distributed rehearsal buffer is updated with new representatives passed alongside their associated activations (line 9).

With this approach of storing states with representatives, each parallel process n can sample remote activations thanks to global sampling. Each copy of the model can therefore reuse neural activations generated in the past by any replica of the model. This is valid because, in a data-parallel context, each copy of the model is synchronized after each training iteration.

6.2.2 Integrating Dark Experience Replay++

A natural way to improve over vanilla Experience Replay is to extend it with the main idea of DER i.e., knowledge distillation. Building on this idea, the authors in [25] propose a variant, named DER++, leveraging both rehearsal of representatives stored in the buffer and neural activations in the loss calculation. As such, DER++ optimizes the following objective:

$$\mathcal{L}_{t_c} + \alpha \cdot \mathbb{E}_{(x',a')\sim\mathcal{B}} \left[\|a' - h_w(x')\|_2^2 \right] + \beta \cdot \mathbb{E}_{(x'',y'')\sim\mathcal{B}}[\ell(f_w(x''), y'')]. \tag{6.2}$$

where β is a hyper-parameter introduced by DER++ setting the importance of the rehearsal term, and (x'', y'') is the input-label pair used for rehearsal. Note that DER++ collapses to DER when $\beta = 0$.

Algorithm 6: Training loop implementing Dark Experience Replay++ (adapted from the original implementation [25]).

1 $\mathcal{B} \leftarrow \{\}$
2 **for** $k = 0$ ***to*** $\frac{|D|}{b} \cdot$ *epochs* **do**
3 $\quad (x, y) \leftarrow \vec{m} \leftarrow$ `sample(D,b)` `// Obtain` b `samples from dataset D`
4 $\quad a \leftarrow h_w(x)$ `// Compute output activations`
5 $\quad (x', y', a') \leftarrow$ `get_representatives(`$\mathcal{B}, r$`)` `// Obtain` r `samples from` $\mathcal{B}$
6 $\quad dis \leftarrow \alpha \cdot \|a' - h_w(x')\|_2^2$ `// Compute distillation term`
7 $\quad (x'', y'', a'') \leftarrow m' \leftarrow$ `get_representatives(`$\mathcal{B}, r$`)` `// Obtain` r `samples from` $\mathcal{B}$
8 $\quad loss \leftarrow \beta \cdot \ell(f_w(x''), y'')$ `// Forward pass on representatives`
9 $\quad g_m \leftarrow \nabla[\ell(f_{w^{(k)}}(x), y) + dis + loss]$ `// Compute gradient`
10 $\quad w^{(k+1)} \leftarrow w^{(k)} + u_{GD}(g_m)$ `// Weight update rule`
11 $\quad \mathcal{B} \leftarrow$ `update_buffer(`$(x, y, a), c$`)` `// Update buffer using current activations`

Algorithm 6 describes the training procedure as proposed by DER++: after sampling a mini-

batch m from dataset D (line 3), the corresponding neural activations a are computed (line 4) using the current model parameters w. Then, similarly to DER, a tuple containing r input-activation pairs is sampled from the distributed rehearsal buffer (line 5), and the distillation term dis is computed using the current model w (line 6). From the rehearsal buffer standpoint, the additional state information brought by activations is held alongside its associated representative training sample, and returned as annotated tuples of tensors (x', y', a'). Next, a second tuple containing r input-output pairs is sampled (line 7) in preparation for rehearsal. A forward pass using these representatives is performed (line 8), whose loss result is weighted by β. Finally, model parameters are updated considering the loss $loss$ obtained from m' and dis and the distributed rehearsal buffer is updated with new representatives passed alongside their associated activations (line 11).

Algorithm 7: DER++ integrated into the distributed rehearsal buffer

```
1  B ← {}
2  for k = 0 to |D|/b · epochs do
3      (x, y) ← m ← sample(D,b)                        // Obtain b samples from dataset D
4      a ← h_w(x)                                       // Compute output activations
5      (aug_x, aug_y) ← m' ← augment(m)                 // Prepare augmented minibatch
6      (x', y', a') ← get_representatives(B, r)         // Obtain r samples from B
7      dis ← α · ||a' − h_w(x')||_2^2                   // Compute distillation term
8      g_m ← ∇[ℓ(f_{w^(k)}(aug_x), aug_y) + dis]        // Compute gradient
9      w^(k+1) ← w^(k) + u_GD(g_m)                      // Weight update rule
10     B ← update_buffer((x, y, a), c)  // Update buffer using current activations
```

A limitation of Algorithm 6, implemented as such, is that two separate forward passes are performed sequentially for rehearsal (the first using representatives at line 8, and the second using the original minibatch at line 9), hindering the parallelization capabilities of the GPUs. For this reason, we assemble the original minibatch and representatives into a single augmented minibatch (we set $\beta = 1$ so that representatives to rehearse do not have a higher weight than original training samples). Algorithm 7 details the enhanced integration of the DER++ algorithm into our proposal, with the explicit rehearsal term in Equation 6.2 being replaced with a single forward pass using an augmented minibatch. As such, the augmented minibatch m' is prepared for rehearsal on line 5. This operation allows to perform a single forward pass on m' containing r past representatives (line 8), while still harnessing knowledge distillation as done with DER when calculating the loss.

We illustrate the simplicity of our proposal for the end-user in Listing 6.1. We reuse the `update()` primitive devised in Section 4.1.2, benefiting rehearsal with all asynchronous optimizations introduced in Chapter 3. Not illustrated in this listing, one should also manage swapping pre-allocated augmented minibatches as done in Listing 4.2.

```
1  aug_minibatch = preallocate_augmented_minibatch()
2
3  # Get the first minibatch of the dataset (does not advance the read cursor)
4  (x, y) = DataPipeline.get_first_minibatch()
5  current_activations = Model.forward(x)
6
7  for i in range(num_steps):
8    (x, y) = DataPipeline.get_next_minibatch()
9
10   # Get an annotated tuple of tensors containing representatives and
       associated activations from the buffer
11   (x_, y_, a_) = RehearsalBuffer.get_representatives()
12   output = Model.forward(x_)
13   dis = alpha * mse_loss(output.activations, a_)
14
15   minibatch = (x, y, current_activations)
16
17   # Reuse the update() primitive to (1) pass the current minibatch, and (2)
       prepare the next augmented minibatch asynchronously
18   r = RehearsalBuffer.update(minibatch, aug_minibatch)
19
20
21   current_activations = Model.train(aug_minibatch, dis)
```

Listing 6.1 – Example of a training loop integrating both rehearsal and knowledge distillation, a form of functional regularization.

By returning annotated tuples allowing arbitrary state information to be served, the distributed rehearsal buffer offers a high degree of flexibility, making the implementation of more-advanced CL algorithms straightforward. The training loop is kept short and concise, while the `update()` primitive encapsulates the complexity of high-performance techniques like multi-threaded concurrency, non-blocking RPCs, asynchronous CUDA copies, and I/O optimizations in data movements thanks to RDMA-enabled bulk transfers.

Conclusion

In this chapter, we explored the concept of a generic rehearsal buffers for storing, sampling, and replacing heterogeneous data to accommodate different rehearsal strategies. We presented an extension to the buffer enabling it to serve heterogeneous data in the form of annotated tuples of tensors. We illustrated the generality of our approach by integrating learning algorithms leveraging knowledge distillation, specifically Dark Experience Replay and Dark Experience Replay++, demonstrating a flexible integration with both rehearsal and functional regularization strategies. Our proposed buffer seamlessly integrates with the data pipeline and can collaborate at a global level using scalable distributed techniques. This design provides efficient access to representatives and associated states under concurrency, both locally and remotely, reusing all asynchronous optimizations presented in previous chapters.

Chapter 7

Application to a Real-Life Use-case: the Ptychography Application

Contents

7.1 Motivating Scenario: Continual Learning in Support of Streaming Applications . . . 102

7.1.1 The Need for *ML out HPC* . . . 102

7.1.2 Ptychographic Image Reconstruction . . . 103

7.1.3 DL-enabled Real-Time Ptychographic Reconstruction . . . 103

7.1.4 Generative Models and Rehearsal-based Continual Learning . . . 106

7.2 Experimental Setup and Methodology . . . 107

7.2.1 Training Dataset . . . 108

7.2.2 Continual Learning Scenario . . . 109

7.2.3 Learning Model . . . 110

7.2.4 Performance Metrics . . . 111

7.2.5 Computing Environment . . . 112

7.3 Experiments and Results . . . 112

7.3.1 Comparison with Baseline Approaches . . . 112

7.3.2 Comparison of Tike vs. PtychoNN Stitched Reconstructions . . . 114

As an increasing number of real-world applications produce data continuously, this opens a challenge: how to adapt and update DL models over time, to effectively capture evolving patterns and trends as they occur? Within this paradigm, minimizing the latency between data acquisition and the generation of subsequent insights is paramount. In this chapter, we demonstrate the versatility of our proposed rehearsal buffer by applying it to an Edge-to-HPC streaming use-case involving a generative DL model.

— First, we motivate the need for Continual Learning to address the recurring pattern in modern HPC applications in which a DL model is trained simultaneously with a

resource-intensive analysis techniques, with the goal of eventually replacing the latter to decrease the latency between data acquisition and generation of insights. We illustrate this workflow with a real-life ptychography imaging application (Section 7.1).

— Next, we define the continual learning scenario corresponding to such a streaming setting, and we introduce the performance metrics used in our study (Section 7.2).

— Finally, we apply rehearsal to enable continual learning for ptychography image reconstruction, addressing questions about the benefits of CL over vanilla incremental training to train DL models on the fly, the impact of knowledge distillation, the speed of convergence, and the comparison with conventional analysis techniques (Section 7.3).

We chose to work on this real-life use-case in the context of the UNIFY associate team, a collaboration between Inria and Argonne National Laboratory (ANL). UNIFY aims to explore innovative approaches to workflow optimization, adaptive data management and processing through hybrid techniques leveraging the strengths of the three aforementioned ecosystems. The Advanced Photon Source (APS) at ANL is acquiring data at the edge, which is then streamed to on-site HPC clusters like ThetaGPU. This experimental setting is a natural fit for continual learning.

7.1 Motivating Scenario: Continual Learning in Support of Streaming Applications

In this section, we discuss a common recurring pattern in modern HPC applications involving the need to train and update DL models concurrently with resource-intensive applications. This scenario is exemplified by ptychographic image reconstruction, where data is acquired in real-time at the edge and streamed to an HPC machine for processing.

7.1.1 The Need for *ML out HPC*

Conventional HPC analysis techniques are highly resource-intensive, whereas running inferences using a surrogate DL model is typically much more cost-effective [36]. However, accurate DL models are not always available, especially in cases where the input data is acquired in real-time. One solution is to initiate the experiment with the conventional analysis technique and use its output data as the ground truth for a simultaneously trained DL model. As soon as the latter becomes sufficiently accurate, one can switch regimes and run inferences only. In this sense, the model is used to steer the experiment. This approach allows to save some computation resources if the cost of training the model and that of running inferences is below the cost of conventional computations. We refer to this paradigm as *ML out HPC* [37] in Section 2.1.1.

Continual learning is required in this context because it enables the DL model to adapt and improve over time as new data becomes available, accommodating distribution shifts by miti-

gating catastrophic forgetting. The ability to quickly update the DL model in response to highly dynamic patterns is critical in such applications. In this context, retraining from scratch each time new training data becomes available is not feasible, not only because full training is slow and resource-intensive, but also because the accumulation of all previous training data over time makes each training on a new task take longer (following a cubic increase). Thus, there is a need to train and update DL models directly from streaming data, without accumulating it.

7.1.2 Ptychographic Image Reconstruction

An example of a scientific HPC application that produces massive streams of data at the edge and processes them on HPC clusters is ptychographic image reconstruction. Specifically, a high-intensity beam produced by a light source is moved along a *specimen* (the "sample" under analysis). As the beam passes through the specimen, it scatters and creates overlapping *diffraction patterns* that are captured by a sensor in the far-field. Then, these diffraction patterns are sent to an HPC machine where *phase retrieval* is performed using an iterative algorithm such as Tike [158]. For a given diffraction pattern, the result of such an inversion algorithm is a pair of small (*structure, phase*) images corresponding to a localized area of the specimen at microscopic level. It then becomes feasible to stitch these results together to reconstruct full structure and phase images corresponding to the current position of the specimen to be analyzed. This procedure is repeated until the current perspective is completely covered. At the end of the scan of the current *perspective*, the specimen is rotated to obtain new diffraction patterns from a different position, which again are sent to the HPC machine to reconstruct new images corresponding to the new perspective. As more images get reconstructed on the HPC machine, they can be further subjected to a tomographic reconstruction to obtain a 3D representation of the specimen.

Ptychographic image reconstruction is applicable in many scientific domains, such as cell biology, materials science, and electronics, utilizing optical, X-ray, and electron sources. Notably, **X-ray ptychography** is widely used with dedicated beamlines at synchrotron light sources. By offering the capacity for high-resolution imaging of samples with minimal preparation, this approach has delivered unprecedented insights into various material and biological specimens. Examples include detailed imaging of biological cells [159], strain imaging of nanowires [160], and the examination of semiconductor structures through Bragg ptychography [161]. Similarly, electron ptychography has yielded significant breakthroughs, achieving sub-angstrom resolution and nanoscale 3D imaging [162].

7.1.3 DL-enabled Real-Time Ptychographic Reconstruction

Even when using HPC machines for performing the phase retrieval procedure, conventional iterative algorithms used for this task are compute-intensive [163]. This limitation is exacer-

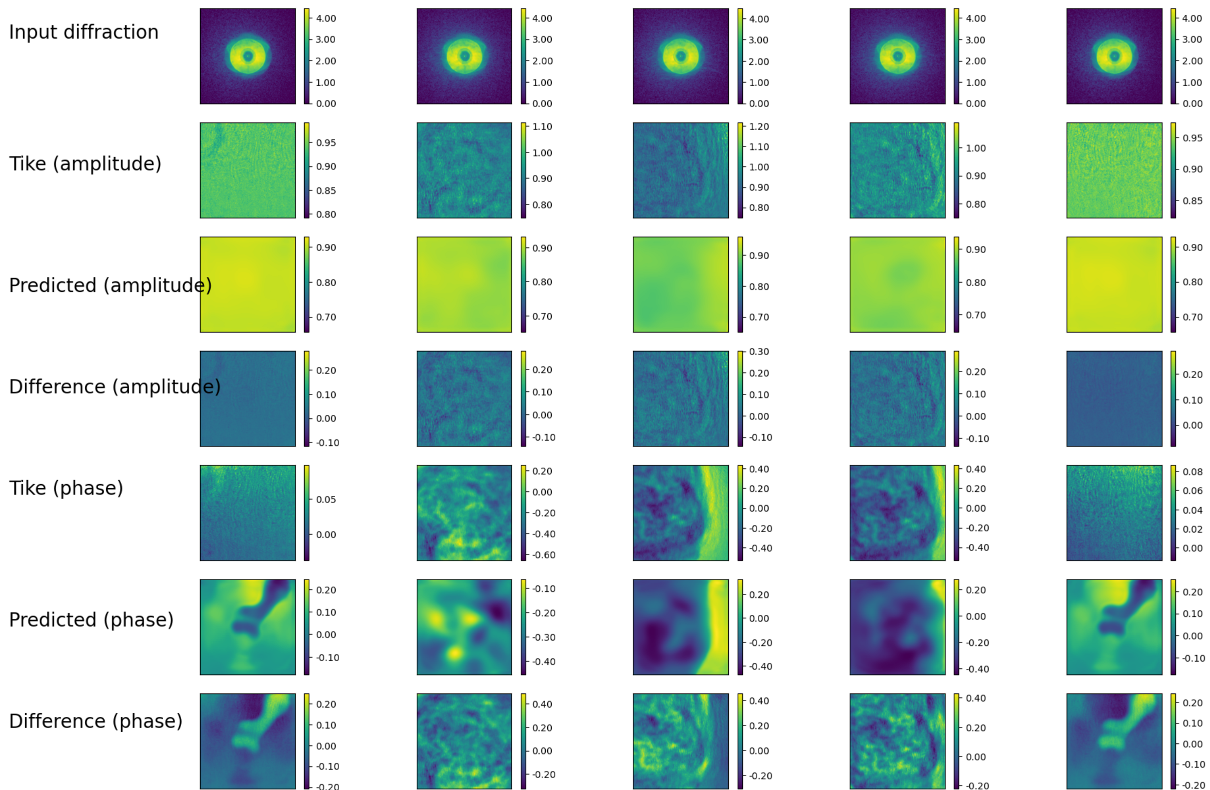


Figure 7.1 – The first row shows 5 different diffraction patterns obtained from the same specimen perspective. For each of these 5 inputs, the following lines show in order: the structure ("amplitude") computed by Tike [158] (conventional iterative algorithm), the amplitude predicted by PtychoNN, the difference between the amplitude computed by Tike and that predicted by PtychoNN, the phase computed by Tike, the phase predicted by PtychoNN, the difference between the phase computed by Tike and that predicted by PtychoNN.

bated by state-of-the-art ptychography instruments which significantly increase the data rate. To mitigate this bottleneck, a possible alternative is to rely on faster generative DL models for data analytics [164, 46]. Such models can directly infer the pair of small (*structure*, *phase*) images corresponding to a diffraction pattern with much lower computational complexity. DNNs employed for ptychography present two other advantages: (1) in contrast to iterative algorithms, DL models can produce live results before acquiring hundreds of diffraction patterns, and (2) because inferences are performed independently on each diffraction pattern, no spatial overlap is required to yield good results [21]. DL models suitable for this generative learning task have been explored before, with PtychoNN [24] as a prominent example. Figure 7.1 shows, for each diffraction acquired by the APS, the pair of images (*structure*, *phase*) reconstructed with the conventional iterative method (Tike) and using model inferences (PtychoNN). Inferred images have a dimension of 128x128 pixels.

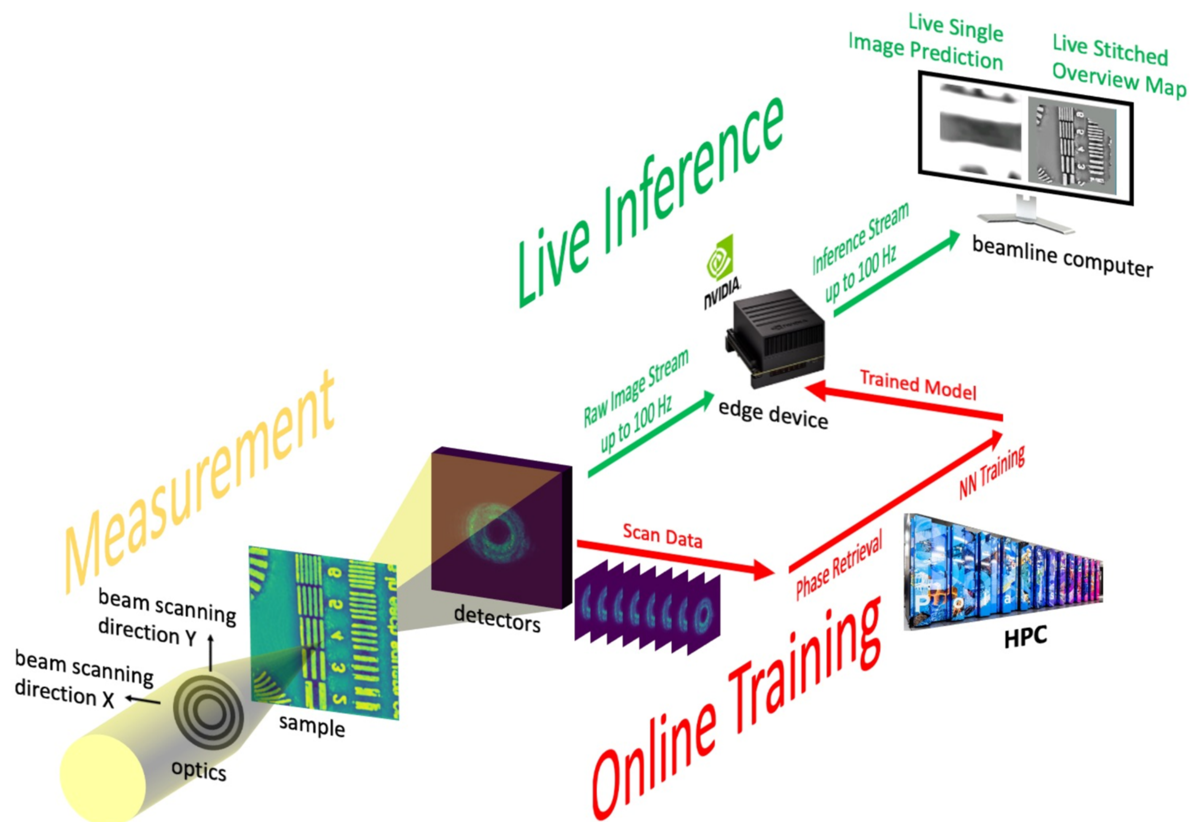


Figure 7.2 – Illustration of DL-enabled workflow for real-time streaming ptychography imaging, featuring the high-intensity beam produced by the light source, the specimen ("sample") under analysis, the HPC cluster performing the continual training of the DNN, and the edge device for live inference. Figure borrowed from [21].

However, since each studied specimen is different i.e., specimens have typically never been observed before, it is not feasible to pre-train a universal DL model that can extrapolate all possible variations of diffraction patterns. Indeed, phase retrieval is highly sensitive to material and biological properties. Instead, one can train a DL model on-the-fly on the HPC cluster, by sending some of the original diffraction patterns in addition to the (*structure*, *phase*) ground truth images reconstructed by the iterative algorithm. Then, once the quality of the inferred results using PtychoNN is acceptable compared with phase retrieval, the workflow can fallback to running inferences only, reducing the computational bottleneck. A low-cost, embedded GPU system at the edge is sufficient for real-time phase retrieval [21] using live inferences. Simultaneously, the DL model training procedure may continue for the DNN's parameters at the edge to be updated periodically. Continual learning plays a key role here, as the diffraction patterns and their corresponding (*structure*, *phase*) reconstructions are streamed continuously from the sensor to the HPC cluster, making it unfeasible to constantly retrain the DNN from scratch. At

high streaming rates, continual learning needs to be coupled with data-parallel training in order to be able to keep the DL model up-to-date in real-time. This workflow is illustrated in Figure 7.2 and is further detailed in [165]. A DL-enabled, continual workflow for ptychographic imaging consists of three concurrent stages: (1) measurement (data acquisition), (2) online training, and (3) live inference using the DL model. Authors in [21] demonstrate that such a workflow can keep up with the current maximum data acquisition rate of 2 kHz, enabling coherent imaging in real-time.

Every time that a pair of (*structure, phase*) images is inferred from a diffraction pattern at the edge, such DL inferences are stitched together to form the corresponding (*structure, phase*) full images gradually at the beamline computer. The stitching procedure consists of interpolating all inference results onto a regular grid. This is accomplished using data such as the beam position and the pixel size of the inference image.

7.1.4 Generative Models and Rehearsal-based Continual Learning

Since this dissertation has only focused on classification problems so far, we discuss how ptychography imaging integrates naturally with the distributed rehearsal buffer. The rehearsal buffer is aware of classes when storing representative samples, which is pertinent to classification models. As briefly discussed in Section 6.1.1, the number of classes can also be limited to one, in which case our approach can be applied to generative models. In the context of ptychography imaging, this class corresponds to a "*diffraction pattern*". However, in practice, users do not have to choose between a single class for generative workloads and K classes for classification workloads: the buffer's generality allows for the definition of an arbitrary number of classes. In the case of ptychography, one could imagine defining two classes as well, depending on whether the diffraction contains an area of the specimen with or without a void. This distinction could make it possible to adapt the way minibatches are subsequently augmented, by favoring the selection of diffractions corresponding to full zones (without voids), or according to any other characteristic enabling to distinguish between diffractions. Buffer management and the distribution of stored representatives can be adjusted to the problem at hand in order to improve the achieved accuracy. For simplicity reasons, we limit the number of classes here to a single class. Experiments about the buffer data layout would be useful to study how this would affect the accuracy of different tasks.

Another key difference from classification problems is the ground truth data type. In the case of a classification problem, the ground truth is a simple label y corresponding to the class number. However, with generative workloads, the shape of the ground truth data is typically more complex as it corresponds to the expected reconstruction of the DNN model (e.g., an image, a text, a video, etc) for a given input sample. For ptychographic imaging, the output of the DNN is a pair of small (*structure, phase*) images reconstructed from a given diffraction pattern.

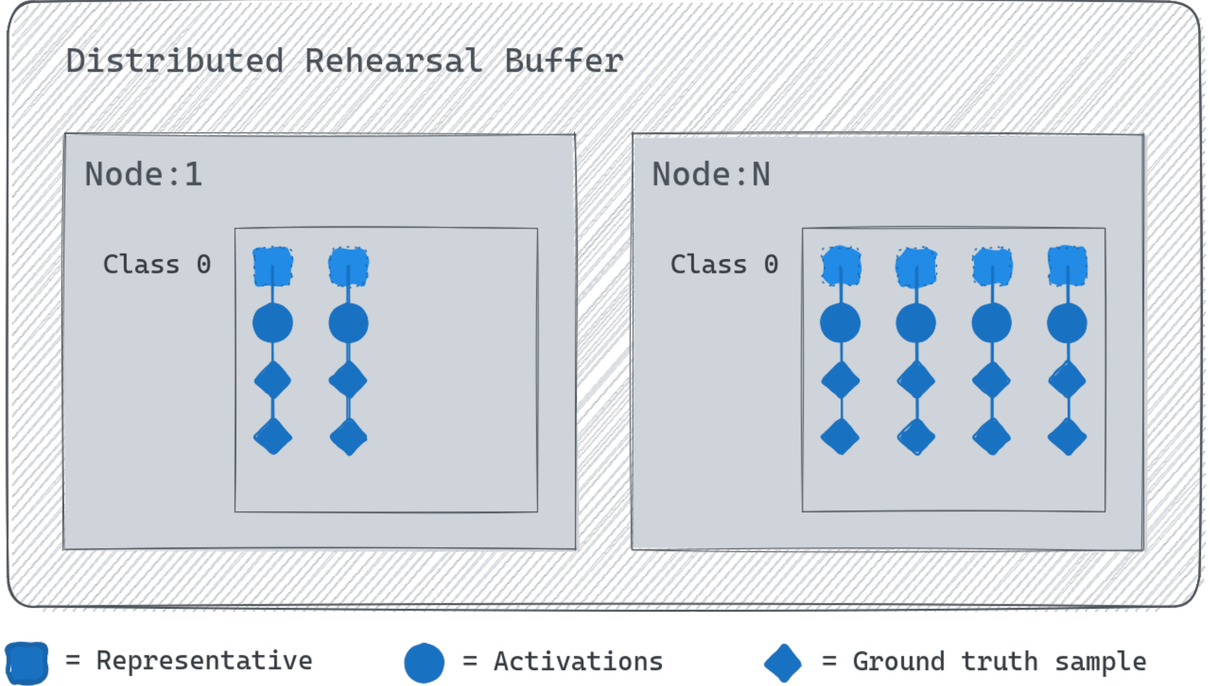


Figure 7.3 – With generative workloads, representatives of a single class are stored in the rehearsal buffer $\mathcal{B}_n$. This includes training samples as well as associated structure and phase reconstructions. Our generic design allows to hold additional information associated to these representatives: this example holds activations. This data is served using global sampling to be made available to the training procedure alongside representatives.

Since these images are ground truth, they are an intrinsic part of a representative training sample. For this reason, they are stored in the buffer $\mathcal{B}_n$ alongside diffraction patterns. The flexible design of the distributed rehearsal buffer allows to handle all these scenarios by storing data intrinsic of representative training samples contiguously in memory. A possible buffer configuration is depicted in Figure 7.3, illustrating the flexibility of our approach to accommodate a large range of deep learning workloads. In this case, all local buffers $\mathcal{B}_{n,1}$ contain representatives of shape $(diffraction, structure, phase)$, all belonging to the same class *diffraction pattern*. Corresponding activation states are stored in a separate buffer $\mathcal{B}_{n,2}$, as discussed in Section 6.1.2.

7.2 Experimental Setup and Methodology

In this section, we define the continual learning scenario and we introduce the performance metrics used in our study. We make links between the concepts developed in this dissertation and the real-life application of ptychography imaging in real-time.

7.2.1 Training Dataset

We use a sequence of 157 perspectives in the order in which they are captured by Argonne's Advanced Photon Source (APS). Each perspective corresponds to a specific rotation of the specimen to be analyzed. The X-ray beam follows a commonly used pattern to scan a perspective, as illustrated in Figure 7.4, which depicts a single perspective. This procedure results in a large number of intensity *diffraction patterns* to be acquired in the far-field. Specifically, about 950 diffraction patterns are produced from every perspective. A pair of structure and phase reconstructions is computed for every individual diffraction using Tike [158], a commonly used phase retrieval algorithm based on traditional computations. When stitched together, all these local reconstructions form a full image of the perspective at hand, that we refer to as the final *stitched reconstruction*.

The objective of the learning model is to supplant the resource-demanding Tike algorithm, with the same purpose of reconstructing the corresponding phase and structure from a given diffraction pattern. In that sense, the (*structure, phase*) pair serves as the ground truth for the learning model, and each training sample is a (*diffraction, structure, phase*) tuple. Unlike the previous classification problem introduced in Chapter 5, we treat all training samples as belonging to a single class (as detailed in Section 7.1.4). The dataset preparation procedure is as follows: for each of the first 156 perspectives, first we filter out diffraction patterns associated with a "blank" phase reconstruction (i.e., containing a void area), that do not capture a meaningful pattern and are too sparse to be informative for training the model. Such blank *Fields of View* (FOVs) could even bias the training in an undesirable direction. Specifically, after about twenty training cycles with all other hyperparameters fixed, we exclude diffraction patterns whose square of the phase image is below a threshold of 0.02. This reduces the number of meaningful training samples from 950 to about 600 on the average. Then, out of the meaningful training samples, we choose 10% randomly to be used as validation data, while the rest is used for the actual training procedure.

Out of the sequence of 157 perspectives, the first 156 are used to train the model, while the last one is used to test the quality of the final stitched reconstruction against the baseline (Tike). Besides, to simulate a realistic CL setup, we do not shuffle the positions of the training samples within each perspective. Instead, we assume they are fed to the training in a streaming fashion, in the same order they were acquired from the sensor, as per Figure 7.4. This allows us to study how the natural order impacts the CL process. Since all training samples belong to the same class, the model is unaware of the specific perspective they originate from.

In Figure 7.4, the axes are not labeled with specific units. This is because the experimental data we obtained was lacking metadata such as the nature of the specimen under analysis, or the precise scaling of the axes. While this absence of metadata might typically be considered a limitation, it is not critical for the purposes of our study. The figure is intended to illustrate

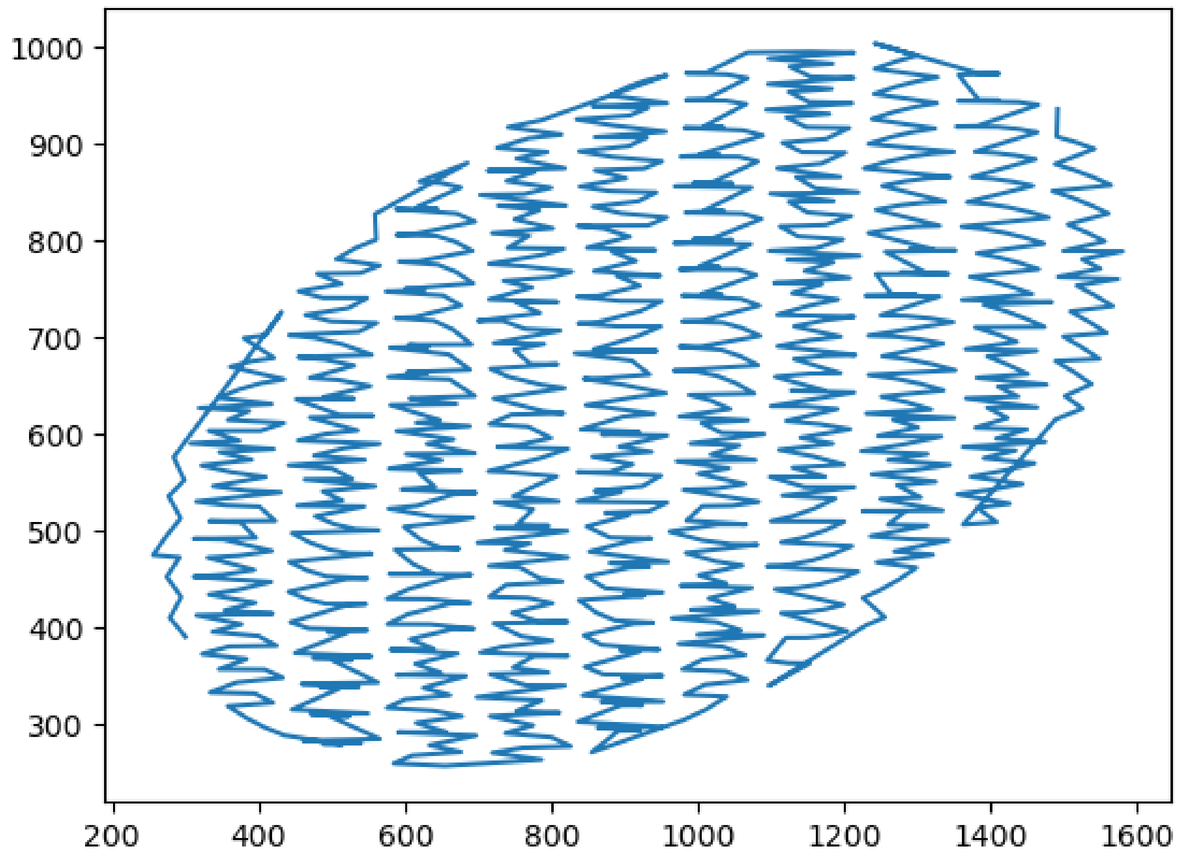


Figure 7.4 – Illustration of the spatial path (left-right, top-bottom) taken by the X-ray beam to scan a single perspective of a specimen. This procedure generates 950 diffraction patterns in the far field, which are then used to compute structure and phase reconstructions using Tike.

the spatial path taken by the X-ray beam during the imaging procedure, which is sufficient for understanding the principles and outcomes discussed. The axes could represent nanometers, a common unit in ptychography imaging techniques, but the primary focus here is on the qualitative behavior of the data acquisition. Specifically, diffraction patterns are measured in a specific order, and fed as such to the analytics workflow (or DL model).

7.2.2 Continual Learning Scenario

Since the goal of the training is to produce an accurate generative DL model, we have a single flat output space. Each new perspective of the specimen represents a new learning task that is used to update the DL model through continual learning, which fits the the domain-incremental ("Domain-IL") scenario, as discussed in Section 2.2.3. This setting characterizes task changes by introducing a shift in the input distribution, corresponding to different perspectives of the spec-

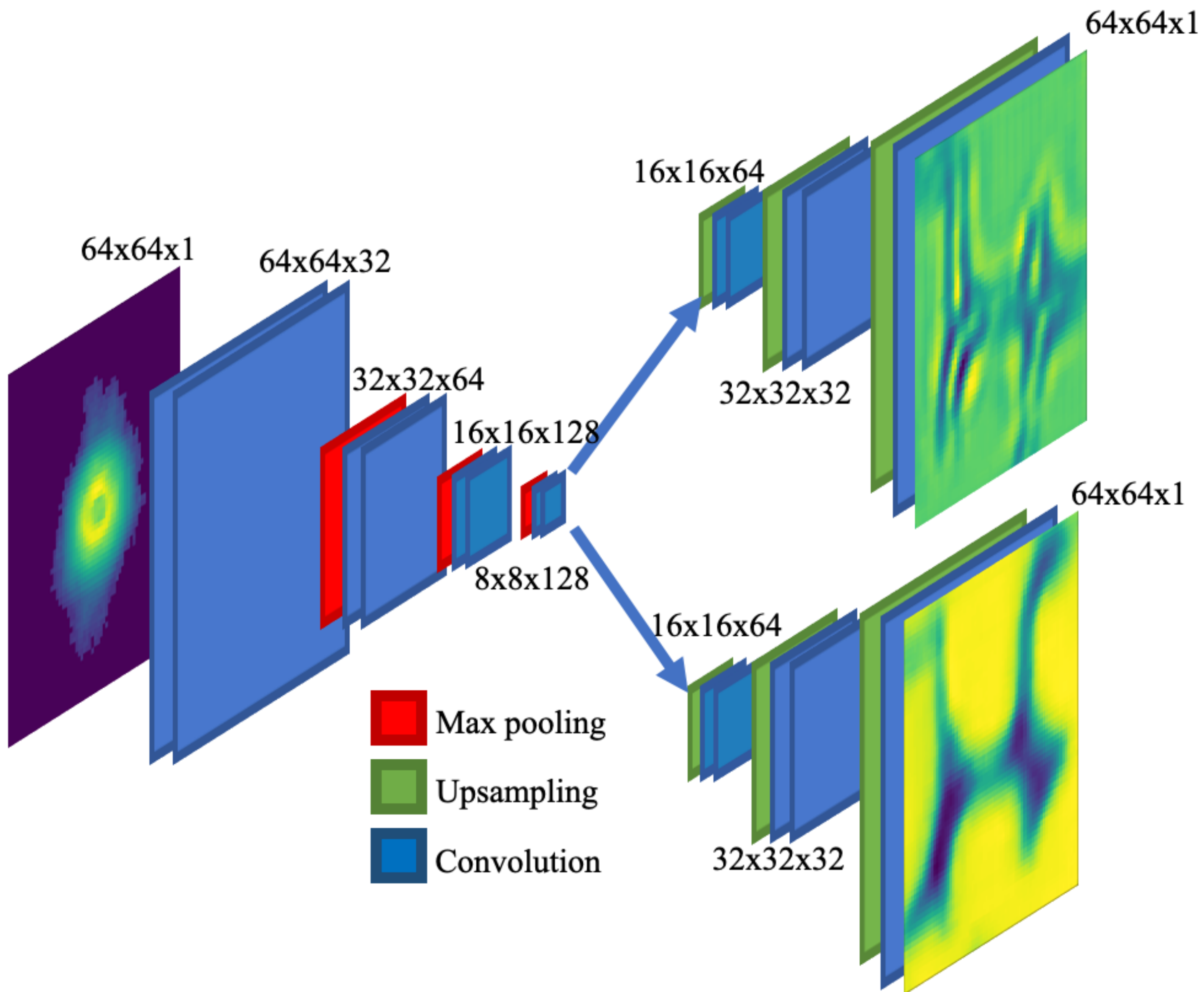


Figure 7.5 – PtychoNN architecture

imen to analyze in our case. This is complementary to the experiments presented in Chapter 5, where we focused on the class-incremental learning scenario.

7.2.3 Learning Model

We use **PtychoNN** [24] to reconstruct ground-truth images from the diffraction patterns data. Specifically, this autoencoder DNN takes as input a diffraction pattern and outputs a pair of structure and phase images (reconstructions). The DL model architecture (depicted in Figure 7.5) consists of three parts: an encoder arm that contains convolutional and max pooling layers designed to learn a representation of the input data (diffraction patterns), and two decoder arms that contain convolutional and upsampling layers designed to reconstruct real-space images (structure and phase). The phase model uses `tanh()` as the activation function, whose

output is restored in the $[\pi; -\pi]$ range.

Since the position of the X-ray beam is known for each diffraction pattern, one can stitch the local reconstructions together to form the final image corresponding to one perspective. We use the Adam optimizer with an initial learning rate $\eta = 0.00088$. We then follow an *exp_range* learning rate schedule as suggested by [166], with a gamma coefficient set to 0.996 and a step size set to 184 training iterations. We enable Automated Mixed Precision (AMP) to speed up the training.

In contrast to the evaluations presented in Section 5.2.2 where we used a minibatch size of $b = 56$ and a representative count of $r = 7$, we now integrate a larger number of representatives into the training procedure, specifically, a minibatch size of $b = 16$ and a representative count of $r = 24$. The proportion of old samples is increased in our experimental context as the input data is not revisited; instead, input diffraction patterns are streamed to the model continuously.

7.2.4 Performance Metrics

The DL model is trained using an absolute square loss calculated between its predictions (i.e., structure and phase images) and the ground-truth images computed by Tike. Furthemore, to evaluate the quality of the final reconstructed image, we use the Peak-Signal-to-Noise Ratio(PSNR) and the Structural Similarity (SSIM) [167] metrics, which are computed against the baseline reconstruction using Tike. PSNR is most easily defined via the mean squared error (MSE). Given a $W \times H$ ground-truth image computed by Tike I and its corresponding PtychoNN reconstruction R, MSE is defined as:

$$MSE(I,R) = \frac{1}{WH} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} [I(i,j) - R(i,j)]^2 \tag{7.1}$$

The PSNR (in dB) is defined as:

$$PSNR(I,R) = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE(I,R)) \tag{7.2}$$

where MAX_I is the maximum pixel value of the ground-truth image. The SSIM measure between I and R is:

$$SSIM(I,R) = \frac{(2\mu_I\mu_R + C_1)(2\sigma_{IR} + C_2)}{(\mu_I^2 + \mu_R^2 + C_1)(\sigma_I^2 + \sigma_R^2 + C_2)} \tag{7.3}$$

where μ_I and μ_R are the average (mean) intensities of I and R, σ_I^2 and σ_R^2 are the variances of I and R, σ_{IR} is their covariance, and C_1 and C_2 are small constants to avoid instability when the denominator is very close to zero.

7.2.5 Computing Environment

We run our experiments on a single node of Argonne National Laboratory's Polaris HPC testbed, with the same software environment as detailed in Chapter 5. It consists of 560 nodes, each equipped with 512 GB of DDR4 memory (aggregated from four NUMA domains), a 32-core AMD Zen 3 (Milan) and four Nvidia A100 GPUs aggregating to a total of 160 GB HBM memory. Four A100 GPUs are connected with each other using four NVLinks and with the host memory through a PCIe Gen 4 interface.

7.3 Experiments and Results

Our experiments seek to illustrate the benefits of a distributed rehearsal buffer for generative DL models. They focus on the generality of the rehearsal buffers, as underlined in Chapter 6, rather than the scalability on multiple GPUs. To this end, we apply rehearsal to enable continual learning for ptychographic image reconstruction, as described in Section 7.1.3. In this case, we aim to answer the following questions:

— What benefits does continual learning based on rehearsal bring over incremental training that ignores catastrophic forgetting?
— Does knowledge distillation help improving the achieved accuracy?
— Does rehearsal speed up convergence?
— What is the impact on the end-result if we replace a traditional reconstruction algorithm (Tike) with a generative DL model?

7.3.1 Comparison with Baseline Approaches

Our first series of experiments compares the evolution of the validation loss (absolute square loss between the DL model predictions and ground truth for the 10% of the training samples reserved as validation data) for an increasing number of tasks (perspectives). This enables us to gain two important insights: (1) how fast CL converges; (2) at what validation loss level does the training stabilize after convergence. We compare the following three approaches:

— **Incremental training**: updates the model directly with the new training samples corresponding to a new perspective. No training samples of any previous task (perspective) are revisited, which may lead to catastrophic forgetting.
— **Rehearsal using Simple Replay (ER)**: augments each minibatch during the training of a new task (perspective) with randomly selected training samples from the rehearsal buffer, which stores representatives from the previous tasks (perspectives).
— **Rehearsal using Dark Experience Replay (DER++)**: follows the same approach as ER, but, additionally, it uses the knowledge distillation technique introduced in Section 6.1.2,

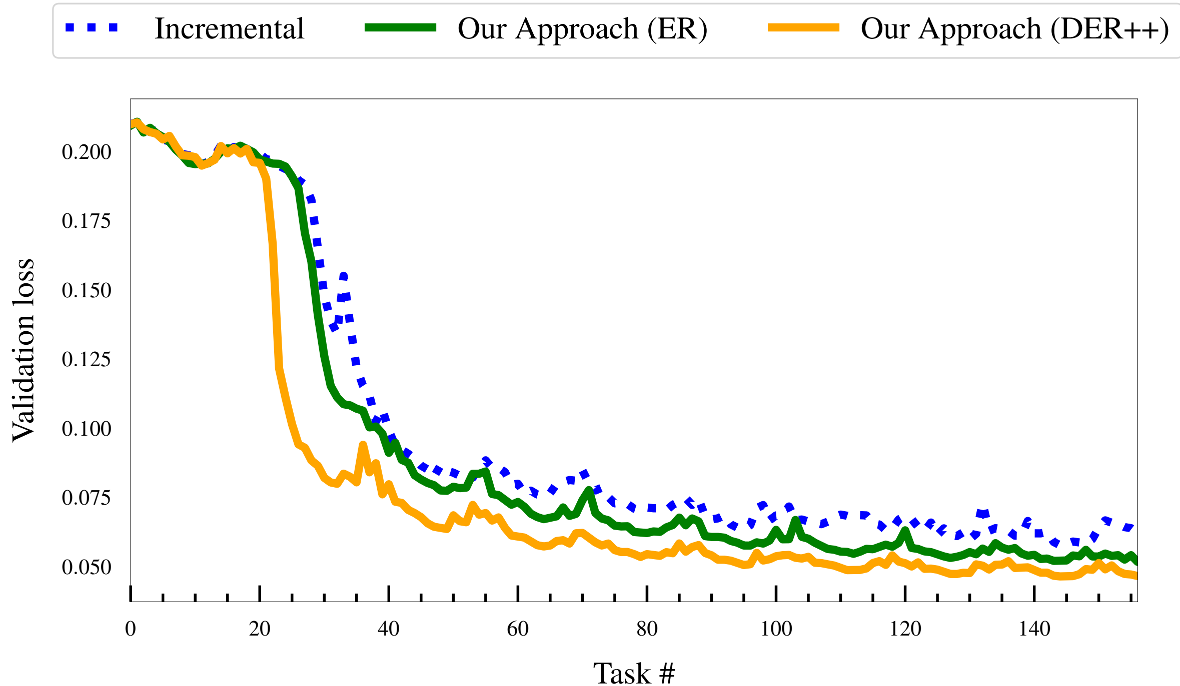


Figure 7.6 – Evolution of the validation loss for an increasing number of tasks.

which leverages the rehearsal buffer to store input activation tuples for the representatives. PtychoNN uses `tanh()` as the activation function for the phase decoder. We ran a hyper-parameter search setting the parameters α and β to 0.8 (knowledge distillation) and 1 (rehearsal), respectively.

The results are depicted in Figure 7.6, where the validation loss on all previous tasks is plotted after training on a new task (perspective). As can be observed, unlike the results obtained for the ImageNet benchmark, the negative effect of catastrophic forgetting is less pronounced in this case: with increasing number of tasks, the incremental training converges relatively fast towards a low validation loss. However, ER and DER++ stabilize after convergence at significantly lower validation loss: 25% and, respectively, 30% lower. Interesting to observe is the convergence speed of the rehearsal approaches. While ER experiences a drop in validation loss that almost overlaps with the incremental training, DER++ experiences a sharper drop significantly sooner. For example, after 30 tasks, DER++ has the same validation loss as ER after 40 tasks. This observation is important because it hints at the advantages of using more advanced forms of rehearsal to train an accurate DL model sooner, which means the switch from traditional computations to a DL model can be triggered earlier, thus improving the overall benefits.

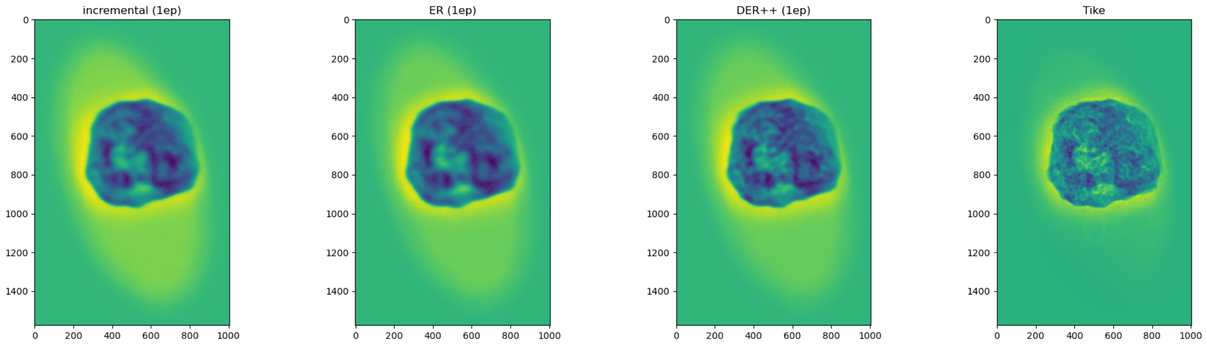


Figure 7.7 – Stitched reconstructions obtained from four different approaches. No task gets revisited during training (one epoch per task). From left to right: incremental training, vanilla Experience Replay, Experience Replay + knowledge distillation (DER++), and the ground-truth reconstruction computed by Tike. One can notice a gradual improvement in reconstruction quality.

7.3.2 Comparison of Tike vs. PtychoNN Stitched Reconstructions

Our next series of experiments focuses on evaluating the end-impact of our proposal in the final results. Specifically, in the case of ptychographic image reconstruction, this involves a full reconstruction of the final image corresponding to a perspective using the DL model predictions (structure and phase), which is then compared with the equivalent final image obtained using Tike. To study the adaptability of the DL model to completely new perspectives (as encountered in a real-life scenario when we delegate retrieval to the DL model by running live inferences), we study the 157-th perspective, which was excluded from the training to form a separate testing set.

The results are visually depicted in Figure 7.7. We generate four final images corresponding to four approaches: incremental training, ER, DER++, and Tike. For each approach, we focus on the phases predicted (or computed in the case of Tike) from the diffraction patterns, which are then stitched together following the path taken by the beam, depicted in Figure 7.4. As can be observed, all three DL models can accurately reconstruct the final image, albeit with a slightly different level of sharpness compared to Tike. In particular, DER++ produces the sharpest image, followed by ER and finally incremental training, for which the studied object is slightly blurry. This may be attributed to an intrinsic limitation of autoencoder models [168].

To quantify the differences between the final images produced by all four approaches, we use the PSNR and SSIM metrics introduced in Section 7.2.4. The results are listed in Table 7.1. As expected, DER++ has the highest quality (higer PSNR and SSIM), followed by ER and finally by incremental training. Interesting to observe is that these metrics are close for all three approaches in absolute terms. However, as mentioned previously, they have an impact in the image quality that is visible in the sharpness level. Furthermore, the reconstruction was done

Learning setting	PSNR	SSIM
Incremental Training	66.31	0.99784
Rehearsal using ER	67.86	0.99853
Rehearsal using DER++	68.24	0.999634

Table 7.1 – PSRN and SSIM of incremental, ER and DER++ relative to *Tike* which is used as the baseline.

for the 157-th perspective after training with 156 perspectives. At this point, all three approaches have converged. If done at a lower task number (earlier perspective), we expect that the metrics may emphasize a larger difference between DER++ and the other two DL models, since DER++ converges faster (as per Section 7.3.1).

Our experiment with ptychographic image reconstruction shows that advanced experience replay strategies for continual learning such as DER++ can outperform simple experience replay (ER) in terms of convergence speed i.e., the faster convergence of the model means it can be reached more quickly, enabling faster analysis of specimens. This improvement helps replacing conventional analysis techniques, which are typically resource-intensive, with faster equivalent DL models. Finally, compared with incremental training, rehearsal-based continual learning enhances accuracy of the reconstruction procedure after convergence.

During the experiment, if the DL model's performance is deemed satisfactory according to the criteria required by the steaming application, the ground truth data generation using conventional analysis techniques can be stopped, or slowed down. This has the effect of lowering the overall resource utilization by moving the workload at the edge entirely. Hence, relying on faster surrogate DL models open the door to real-time imaging, and thus automated steering of experiments. While we did not demonstrate this aspect experimentally, we plan to investigate it in future work and to quantify the benefits.

Conclusion

In this chapter, we have demonstrated the versatility and effectiveness of our proposed rehearsal buffer by applying it to a different learning task involving a generative DL model. We discussed the necessary adaptations and highlighted the flexibility of our buffer design in accommodating a wide range of deep learning workloads, including generative models. By applying our approach to the real-life use-case of ptychographic image reconstruction, we showcased the potential of our method in enabling continual learning in support of streaming applications. Our results indicate that our proposed rehearsal buffer can speed up the convergence of generative DL models in handling streaming data, making it a valuable tool for modern HPC applications that need to simultaneously train and run inferences on DL models that need to adapt to highly dynamic patterns. Notably, rehearsal coupled with knowledge distillation (DER++) achieves the best reconstruction quality, with a 1.93 point improvement in the PSNR (Peak Signal-To-Noise Ratio) metric over incremental training. As a result, the reconstruction quality obtained with incremental training can be achieved more quickly when using the distributed rehearsal buffer.

Chapter 8

Conclusion and Prospects

Contents

8.1 Achievements **118**
8.1.1 Achieving Much-Improved Accuracy in Class-Incremental CL Scenarios 118
8.1.2 Enabling the Support of More Advanced CL Algorithms 119
8.1.3 Enabling the Steering of Experiments in Real-Time, by Learning from a Stream of Data 119
8.2 Prospects **120**
8.2.1 More Advanced Selection and Eviction Policies 120
8.2.2 Application to Numerical Simulations 121
8.2.3 Application to Large Language Models 122

Existing research typically addresses distributed deep learning and continual learning separately. In this dissertation, we were interested in how CL methods can take advantage of data parallelization across nodes, which is one of the main techniques to achieve training scalability on HPC systems. The aggregated memory could benefit the accuracy achieved by such algorithms by instantiating distributed rehearsal buffers. The main research goals of this dissertation were the (1) design and implementation of a rehearsal buffer leveraging distributed systems effectively and the (2) study of trade-offs introduced by large-scale CL in terms of training time, accuracy and memory usage. We summarize some of the research questions that we addressed in this dissertation:

1. A first step towards enabled CL at scale was to understand the role of **distributed rehearsal buffer** by answering questions like: *What are the extensions needed to leverage rehearsal buffers for data-parallel training, and how do they impact the training performance? How can asynchronous techniques be used to hide the overhead of managing rehearsal buffers?*
2. A natural next step was to **generalize the rehearsal buffers** to enable more advanced continual learning settings. With a growing diversity of rehearsal techniques, it becomes important to decouple the rehearsal buffer from the learning task, such that it becomes a generic abstraction that can store additional state information. For instance, we answer questions like *How can we design a rehearsal buffer that do not impose any particular constraints*

on the data layout of representatives retained in the rehearsal buffer? How can we serve additional state information conveniently to make it available in the training loop?

3. Finally, we assessed the impact of our proposal on a real-life use-case involving **streaming data**, with the aim of illustrating how continual learning can benefit real-time applications. One relevant challenge is *How can the integration of DL models in HPC streaming applications be practically applied to improve the efficiency and accuracy of real-life experiments?*

The specific contributions of this dissertation toward answering the above questions are detailed in the next section. We then discuss the prospects opened for future work.

8.1 Achievements

We structure the list of achievements of this dissertation to mirror the research objectives listed in Section 1.2.

8.1.1 Achieving Much-Improved Accuracy in Class-Incremental CL Scenarios

Corresponding research objective: A Distributed Rehearsal Buffer to Enable Continual Learning at Scale.

We propose a novel distributed rehearsal buffer abstraction that aims to leverage distributed systems effectively. Specifically, we (1) define the concept of rehearsal buffers to address continual learning, and introduce **extensions to leverage them for data-parallel training** in Chapter 3. Data parallelism allows to achieve short runtime and scalability while retaining high accuracy. Next, we (2) introduce key design principles such as **asynchronous techniques to hide the overhead of managing rehearsal buffers** in Chapters 3, 4. Such optimizations are necessary to prepare augmented minibatches anticipating the next training iteration. Besides, **global sampling** enables a full spectrum of representative combinations for **minibatch augmentations**. The novelty of our work lies in the data management techniques to make rehearsal scalable on many GPUs. To our best knowledge, such HPC-oriented aspects aimed at improving training performance were not explored before.

We demonstrate in Chapter 5 the benefits of the distributed rehearsal buffer by studying the trade-offs introduced by large-scale CL using a synthetic benchmark in terms of training time, accuracy and memory usage. We run extensive experiments on up to 128 GPUs of the ThetaGPU supercomputer to compare our approach with baselines representative of from-scratch training (the upper bound in terms of accuracy) and incremental training (the lower bound in terms of training time). In our class-incremental scenario using the ImageNet-1K dataset, results show that rehearsal-based continual learning achieves a top-5 classification accuracy of 80.55%, close to the upper bound (91%). The achieved **accuracy is much improved compared with incremental training** (23.3%). Furthermore, the overhead of managing the rehearsal buffer is absorbed

asynchronously in our experiments, enabling rehearsal-based CL to achieve the same training performance than incremental training. Finally, we show that storing more representatives in the buffer improves accuracy, but that the effect is no longer apparent after a certain threshold (corresponding to storing 30% of the ImageNet dataset). We use three different convolutional networks to demonstrate the model-agnostic nature of our proposal.

8.1.2 Enabling the Support of More Advanced CL Algorithms

Corresponding research objective: A Generic Distributed Buffer Generic in Support of More Advanced Rehearsal Techniques.

A natural evolution of our proposal is to make the buffer generic in order to **support a larger range of continual learning techniques**. This amounts to decoupling the rehearsal buffer from the learning task while transparently augmenting the minibatches with the data needed to learn the task at hand. To achieve this, we identified three main levers in Chapter 6. Specifically, we (1) relax constraints on the data layout of representatives retained in the rehearsal buffer, allowing for the **unified management of representatives** (useful for generative tasks), or for the **per-class management of representatives**. Next, we (2) propose to store **additional heterogeneous state information** as needed by more advanced rehearsal-based CL algorithms in separate local buffers, reusing all optimizations developed in Chapter 4 for efficient access under concurrency. Finally, we (3) propose the concept of **annotated tuples** of tensors to serve representative training samples and their associated states, enabling to enrich tuples $(x, y, \ldots)$ exposed to the AI runtime with heterogeneous data.

8.1.3 Enabling the Steering of Experiments in Real-Time, by Learning from a Stream of Data

Corresponding research objective: Integration with a Real-life HPC Streaming Application Benefiting from Generative Continual Learning.

In order to demonstrate once again that the distributed rehearsal buffer can improve the performance of very different learning tasks (as discussed in Chapter 6), we study its adaptation to a **real-life use-case leveraging a generative DL model**. Our focus is on a scientific HPC application that produces massive **streams of data at the edge** and processes them on HPC machines, with the aim of generating reconstructions using **ptychography imaging**. We explain in Chapter 7 how the distributed rehearsal buffer stores more complex-shaped representatives as required by reconstruction tasks.

We report on experiments showcasing the potential of our proposal in **enabling continual learning in support of streaming applications**. We use the PtychoNN model in a domain-incremental CL setting for two different types of rehearsal (vanilla Experience Replay and Dark

Experience Replay++ leveraging knowledge distillation), which both compare favorably to incremental training in terms of final achieved accuracy. This leads to better reconstruction quality for the specimen under analysis. Furthermore, our results show that DER++ enables the **fastest convergence speed** of the model required for the **real-time visualization** of specimens at the edge. Finally, if the DL model's performance is deemed satisfactory during the experiment, the ground truth data generated by computationally intensive analysis techniques can be stopped to rely on DL inferences only. This regime helps **lowering the overall resource utilization** by moving the workload at the edge entirely.

8.2 Prospects

This dissertation opens several prospects. We list the most promising ones in this section. We divide these prospects into two sections.

8.2.1 More Advanced Selection and Eviction Policies

The novelty of our work lies in the techniques to make rehearsal scalable in the context of data-parallel training. For this reason, we focused on the *Naive Incremental Learning* (NIL) algorithm [20] as a simple selection approach, which is sufficient to illustrate the scalability of our data management techniques. This algorithm simply assigns a probability of c/b to each sample of the incoming minibatch m to be pushed into the local rehearsal buffer R_n^i (i denotes the class id). As such, c acts like an update rate: i.e., the higher the value of c, the more often representatives are renewed in rehearsal buffer $\mathcal{B}_n$. Please note that we have not studied the effect of parameter c on accuracy, as the competition to populate the buffer is done per class. In a class-incremental setting, where classes do not reappear in different tasks, representatives of class i are replaced only when the task in which training samples of class i is learned, and the content of a buffer R_n^i remains unchanged afterwards.

A research question that has received a lot of attention recently is how to best sample representatives to store in the distributed rehearsal buffer. As demonstrated in this dissertation, rehearsal is very effective and led to many variants depending on: (1) *how* the rehearsal buffer content is managed, (2) *which* training samples are selected to populate the buffer, and (3) *what* kind of regularization is applied on representatives.

Regarding (1), *how* is the rehearsal buffer populated, some authors propose a *reservoir sampling* strategy [169] to manages representatives [89, 72, 170]. Reservoir sampling guarantees that the content of the buffer constitute a uniform sample from the input data stream of unknown length. Each item in the stream is included in the rehearsal buffer with a probability of $|\mathcal{B}_n|/N$, where N is the total number of samples observed thus far. This approach would be particularly advantageous in scenarios where representatives are not organized by class id

within the rehearsal buffer, such as when the number of classes T is unknown prior to the training procedure. Other authors propose a *ring buffer* strategy [19, 72], where buffers R_n^i follow a FIFO pattern. As the competition is done per class (like implemented by our proposal), representatives from older tasks do not change throughout training in a class-incremental scenario. These methods are similar to our approach, and trivial to implement in the distributed rehearsal buffer. The required changes are to be made in Neomem, independently of the calling Python code.

Regarding (2), *which* representatives are selected to be stored in the buffer, some authors propose *mean of exemplars* or *mean of features* [18, 72] inspired by *incremental clustering* [171], a selection strategy that keeps track of the average feature vector for each class, and stores candidates whose feature representation is closest to the average feature vector. A complementary approach to evict representatives is *herding* [172] to preserve the data distribution associated with representatives of a given class. Another eviction policy proposed in [76] is to target representatives based on their gradient so as to keep diverse representatives in the buffer, formulating this as a constraint reduction problem. This approach proves more efficient than reservoir sampling on imbalanced data. The Loss-Aware Reservoir Strategy (LARS) proposed in [75] adopts an approach in which representatives are stored alongside their original loss score to evict low values first. These approaches can be integrated into the distributed rehearsal buffer, but in most cases more substantial additions will have to be made as the computational complexity increases.

Finally, regarding (3), *what* kind of regularization is applied on representatives [19, 89]. Similarly to LARS, authors in [173] leverage the loss score of representatives to devise a priority function called Error-Prioritized Replay (EPR). This approach modifies the loss calculation accordingly with importance weights. Built upon the seminal work of [174] providing new insights about knowledge distillation, authors in [92] revisit the DER algorithm. Specifically, they propose to rewrite neural activations associated with past representatives as they are sampled to leverage new insights regarding previous tasks. Furthermore, inspired by *contrastive learning* [175], X-DER implements a *future preparation* strategy which prepares classification heads (neurons) associated to incoming tasks yet to be observed. This mitigates large parameter updates, lowering the risk of forgetting as a result. The performance of such regularization methods will benefit from asynchronous augmentations required for operations needed for knowledge distillation, storing additional states in the distributed rehearsal buffer, and serving them to the AI runtime using annotated tuples. Rewriting states could be achieved with a slight variant of `accumulate()` that could update existing data.

8.2.2 Application to Numerical Simulations

Numerical simulations are a critical tool for scientific discovery and engineering applications, but their computational intensity often hinders their efficiency [176]. Recent advances in

deep learning have shown promise in accelerating these simulations through the development of deep surrogate models. For instance, Physics-Informed Neural Networks [177] (PINNs) are typically trained in a supervised fashion using simulation data [178, 179]. Building on this idea, authors in [130] present the Melissa framework designed to train DL models on synthetic data generated on-the-fly by conventional simulation codes. By generating rich datasets simultaneously with the DL training procedure, this approach avoids the need for cumbersome data storage and costly I/O operations. Instead, the generated data is directly streamed to a rehearsal buffer where reservoir sampling is used to mitigate the inherent bias of sequential samples, resulting in increased generalization capabilities and lower validation loss. Another benefit of this approach is to maximize GPU throughput by accumulating multiple training samples waiting to be ingested. Experiments have demonstrated the effectiveness of this approach, with training times reduced from days to hours. The framework has also been shown to enable training on large datasets up to 8TB in size, in a fraction of the time required by conventional simulations. This is accomplished thanks to a a multi-level parallelism (parallel solver execution, data parallel training, ensemble run execution).

Many concepts introduced by our proposal could be reused to achieve the same goal of training deep surrogate models from simulation data. First, the generic design of the distributed rehearsal buffer allows to store representatives of any shape, accommodating any training sample generated by clients running the simulation code. Then, the distribution of samples from clients to GPUs in a round-robin fashion can be delegated to the buffers themselves, which use global sampling to enforce the balance of data for data parallel training. Next, the 1:1 synchronization between training iterations and calls to `accumulate()` (as discussed in Section 4.1.1) should be relaxed to enable clients to populate the rehearsal buffer at a higher throughput than the model ingestion rate, enabling GPUs to remain active at all times. All in all, the biggest change for Neomem is the implementation of a reservoir sampling strategy as discussed in Section 8.2.1.

8.2.3 Application to Large Language Models

Recent advances in Large Language Models [180] (LLMs) have centered around larger datasets, larger DL models, and larger context lengths. Learning in context, where the model improves based on the surrounding user input (also referred to as *prompt*), benefits from longer context lengths to enhance accuracy[181]. However, there are certain limitations to relying solely on *in-context learning* during inference, as the model's ability to adapt and acquire new knowledge is constrained. For this reason, continual learning may be required to further enhance its performance. In this section, we discuss how our proposal could benefit both in-context learning and continual fine-tuning of LLMs.

In-context learning Large Language Models are trained on vast amounts of text data to generate human-like responses by predicting the likelihood of a word given its preceding words in a sentence. At inference time, if the training data of an LLM is unrelated to the user prompt, the model's response will likely be unsatisfactory, as it may not possess the necessary knowledge or context to provide a relevant and accurate answer. *In-context learning* involves providing a model with relevant context before prompting it with a task, allowing the model to extract information from the context and apply it to the task at hand. As such, the user prompt is *augmented* by incorporating additional contextual information. This approach, which does not require additional training or backpropagation, is called Retrieval Augmented Generation [182] (RAG), where relevant context is retrieved and used to generate valuable output. Recent advancements in LLMs allowed to increase the maximum context length to 10 million tokens [183], underscoring the rationale to resolve these issues with such methods relying on RAG and extended context lengths. Models generate hallucinations [184, 185] due to their probabilistic nature. However, the likelihood of a model producing hallucinations substantially decreases when the DL model is provided with the correct via context retrieval [186]. Using this idea as a starting point, some authors advocate for *retrieval-augmented LLMs* [187], where a new dimension of scaling LLMs is introduced by focusing on the size of the datastore used for RAG at inference time [188]. This study demonstrates that a larger datastore improves language modeling and downstream task performance, underscoring the importance of considering datastore size as an integral part of LLM efficiency and performance trade-offs.

In this context, our proposal of a distributed rehearsal buffer could be leveraged to inject samples into the context window when running inferences. This would require the ability to discriminate between representatives (stored as embeddings), in order to sample those most relevant to the current inference. This procedure would be carried out by a *retriever* [189] that would integrate with the rehearsal buffer. Such a component should adjust its search behavior depending on the task at hand, and would represent a significant and complementary research effort.

Continual Fine-tuning Despite the simplicity and effectiveness of in-context learning, this approach lacks close interactions with LLM components, posing at least three significant challenges. A first limitation pertains to inference costs, which increase exponentially with larger context lengths [190]. Besides, in some scenarios, the relevant context simply may not exist, leading to issues like unsupported generations [191]. For instance, if a user wants to be assisted when working with specialized libraries that lack documentation, RAG or retrieval methods cannot empower models to address the unforeseen training data. Finally, the last critical shortcoming pertains specifically to in-context learning, as a model's in-context learning capacity is constrained by its pre-training data. If a given LLM has been trained on code and documenta-

tion, one would anticipate its proficiency in concepts related to programming. However, its performance on unrelated tasks would be suboptimal. In these scenarios where in-context learning falls short, the training procedure should be resumed using data samples relevant to the current and future tasks. However, incremental training applied to LLMs, also referred to as *fine-tuning*, suffers from catastrophic forgetting [192] too. Rehearsal on representative training samples has been discussed in the literature lately [193, 194, 195, 196] to mitigate this issue. Therefore, all the arguments developed in this dissertation apply to this new field of research, and our proposal for a distributed rehearsal buffer will also benefit these large models.

The preparation of augmented minibatches for training LLMs affects the data pipeline that supplies samples to the model, rather than the training procedure itself, as extensively discussed in this dissertation. From a technical standpoint, the integration of rehearsal buffers with DeepSpeed [119], the primary runtime to perform model parallelism, is crucial. This integration enables the exploitation of 3D parallelism, which synergistically combines data, tensor, and pipeline parallelism to optimize computational efficiency. Furthermore, to maximize memory efficiency, the utilization of ZeRO optimizations [122, 123] allows for the offloading of parameters, gradients, and optimizer states to CPUs during periods of inactivity in the computation, thereby minimizing memory usage and enhancing overall system performance. To our best knowledge, such a system combining 3D parallelism and specialized data management techniques to make rehearsal scalable in this context has not been proposed yet.

BIBLIOGRAPHY

[1] M. Alam et al., « Survey on Deep Neural Networks in Speech and Vision Systems », *in*: *Neurocomputing* 417 (2020), pp. 302–321.

[2] Stephan Rasp, Michael S Pritchard, and Pierre Gentine, « Deep learning to represent subgrid processes in climate models », *in*: *Proceedings of the National Academy of Sciences* 115.*39* (2018), pp. 9684–9689.

[3] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang, « Predicting disruptive instabilities in controlled fusion plasmas through deep learning », *in*: *Nature* 568.*7753* (Apr. 2019).

[4] Prasanna Balaprakash et al., « Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research », *in*: *SC'19: The 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2019.

[5] Sushen Zhang et al., « Learning for personalized medicine: a comprehensive review from a deep learning perspective », *in*: *IEEE reviews in biomedical engineering* 12 (2018), pp. 194–208.

[6] Connor Shorten, Taghi M. Khoshgoftaar, and Borko Furht, « Deep Learning applications for COVID-19 », *in*: *Journal of Big Data* 8.*1* (2021), pp. 1–54.

[7] E. A. Huerta et al., « Convergence of artificial intelligence and high performance computing on NSF-supported cyberinfrastructure », *in*: *Journal of Big Data* 7.*1* (2020), p. 88.

[8] Tal Ben-Nun and Torsten Hoefler, « Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis », *in*: *ACM Comput. Surv.* 52.*4* (2019).

[9] Girish Sastry et al., « Computing Power and the Governance of Artificial Intelligence », *in*: *arXiv preprint arXiv:2402.08797* (2024).

[10] Mengmeng Wang et al., « Recent progress on reliability analysis of offshore wind turbine support structures considering digital twin solutions », *in*: *Ocean Engineering* 232 (2021), p. 109168.

[11] Xin Li et al., « Big Data in Earth system science and progress towards a digital twin », *in*: *Nature Reviews Earth & Environment* 4.*5* (2023), pp. 319–332.

[12] Sebastian Thrun, « A lifelong learning perspective for mobile robot control », *in*: *Intelligent Robots and Systems*, Elsevier, 1995, pp. 201–214.

[13] Raia Hadsell et al., « Embracing change: Continual learning in deep neural networks », *in*: *Trends in Cognitive Sciences* 24.12 (2020), pp. 1028–1040.

[14] Michael McCloskey and Neal J Cohen, « Catastrophic interference in connectionist networks: The sequential learning problem », *in*: *Psychology of Learning and Motivation*, vol. 24, Elsevier, 1989, pp. 109–165.

[15] Michalis K Titsias et al., « Functional regularisation for continual learning with gaussian processes », *in*: *arXiv preprint arXiv:1901.11356* (2019).

[16] Pingbo Pan et al., « Continual deep learning by functional regularisation of memorable past », *in*: *Advances in Neural Information Processing Systems* 33 (2020), pp. 4453–4464.

[17] Seyed Iman Mirzadeh et al., « Understanding the role of training regimes in continual learning », *in*: *Advances in Neural Information Processing Systems* 33 (2020), pp. 7308–7320.

[18] Sylvestre-Alvise Rebuffi et al., « icarl: Incremental classifier and representation learning », *in*: *IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[19] David Lopez-Paz and Marc'Aurelio Ranzato, « Gradient episodic memory for continual learning », *in*: *Advances in Neural Information Processing Systems* 30 (2017).

[20] David Munoz et al., « Incremental learning model inspired in Rehearsal for deep convolutional networks », *in*: *Knowledge-Based Systems* 208 (2020), p. 106460.

[21] Anakha V Babu et al., « Deep learning at the edge enables real-time streaming ptychographic imaging », *in*: *Nature Communications* 14.1 (2023), p. 7059.

[22] Thomas Bouvier et al., « Efficient Data-Parallel Continual Learning with Asynchronous Distributed Rehearsal Buffers », *in*: *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Philadelphia (PA), United States, May 2024, DOI: 10.1109/CCGrid59990.2024.00036, URL: https://inria.hal.science/hal-04600107.

[23] Franz Pfeiffer, « X-ray ptychography », *in*: *Nature Photonics* 12.1 (2018), pp. 9–17.

[24] Mathew J Cherukara et al., « AI-enabled high-resolution scanning coherent diffraction imaging », *in*: *Applied Physics Letters* 117.4 (2020).

[25] Pietro Buzzega et al., « Dark experience for general continual learning: a strong, simple baseline », *in*: *Advances in Neural Information Processing Systems* 33 (2020), pp. 15920–15930.

[26] Thomas Bouvier et al., « Efficient distributed continual learning for steering experiments in real-time », *in*: *Future Generation Computer Systems* (2024), ISSN: 0167-739X, DOI: https://doi.org/10.1016/j.future.2024.07.016, URL: https://www.sciencedirect.com/science/article/pii/S0167739X24003820.

[27] Thomas Bouvier, Alexandru Costan, and Gabriel Antoniu, *Heterogeneity-aware Deep Learning Workload Deployments on the Computing Continuum*, IPDPS 2021 - 35th IEEE International Parallel and Distributed Processing Symposium, Poster, May 2021, URL: `https://hal.science/hal-03270129`.

[28] Thomas Bouvier, Alexandru Costan, and Gabriel Antoniu, « Deploying Heterogeneity-aware Deep Learning Workloads on the Computing Continuum », *in*: *Proceedings of the BDA 2021 conference*, Proceedings of the BDA 2021 conference, Paris, France, Oct. 2021, URL: `https://hal.science/hal-03338520`.

[29] Thomas Bouvier, *Neomem*, `https://github.com/thomas-bouvier/neomem`, 2023, (visited on 12/31/2023).

[30] Thomas Bouvier, *Distributed Continual Learning*, `https://github.com/thomas-bouvier/distributed-continual-learning`, 2023, (visited on 12/31/2023).

[31] Todd Gamblin et al., « The Spack package manager: bringing order to HPC software chaos », *in*: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[32] Alexander Sergeev and Mike Del Balso, « Horovod: Fast and easy distributed deep learning in TensorFlow », *in*: *arXiv preprint arXiv:1802.05799* (2018).

[33] Daniel Rosendo et al., « E2clab: Exploring the computing continuum through repeatable, replicable and reproducible edge-to-cloud experiments », *in*: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2020, pp. 176–186.

[34] Anthony Robins, « Catastrophic forgetting, rehearsal and pseudorehearsal », *in*: *Connection Science* 7.2 (1995), pp. 123–146.

[35] Olivier Terzo and Jan Martinovič, *HPC, Big Data, and AI Convergence Towards Exascale: Challenge and Vision*, CRC Press, 2022.

[36] Geoffrey Fox et al., « Learning everywhere: Pervasive machine learning for effective high-performance computation », *in*: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 422–429.

[37] Rosa M Badia et al., « Integrating HPC, AI, and Workflows for Scientific Data Analysis (Dagstuhl Seminar 23352) », *in*: (2024).

[38] João Gama et al., « A survey on concept drift adaptation », *in*: *ACM computing surveys (CSUR)* 46.4 (2014), pp. 1–37.

[39] Masashi Sugiyama, Matthias Krauledat, and Klaus-Robert Müller, « Covariate shift adaptation by importance weighted cross validation. », *in*: *Journal of Machine Learning Research* 8.5 (2007).

[40] Daniel Rosendo et al., « Distributed intelligence on the Edge-to-Cloud Continuum: A systematic literature review », *in*: *Journal of Parallel and Distributed Computing* 166 (2022), pp. 71–94.

[41] Rafael Vescovi et al., « Linking scientific instruments and computation: Patterns, technologies, and experiences », *in*: *Patterns* 3.*10* (2022).

[42] Patricia Daukantas, « Synchrotron light sources for the 21st century », *in*: *Optics and Photonics News* 32.*9* (2021), pp. 32–39.

[43] Amanda Weltman et al., « Fundamental physics with the square kilometre array », *in*: *Publications of the Astronomical Society of Australia* 37 (2020), e002.

[44] Anna Lena Eberle and Dirk Zeidler, « Multi-beam scanning electron microscopy for high-throughput imaging in connectomics research », *in*: *Frontiers in neuroanatomy* 12 (2018), p. 112.

[45] Dany Vohl et al., « Enabling near real-time remote search for fast transient events with lossy data compression », *in*: *Publications of the Astronomical Society of Australia* 34 (2017), e038.

[46] Zhengchun Liu et al., « BraggNN: fast X-ray Bragg peak analysis using deep learning », *in*: *IUCrJ* 9.*1* (2022), pp. 104–113.

[47] Mathew J Cherukara, Youssef SG Nashed, and Ross J Harder, « Real-time coherent diffraction inversion using deep generative networks », *in*: *Scientific reports* 8.*1* (2018), p. 16520.

[48] Kaiming He et al., « Delving deep into rectifiers: Surpassing human-level performance on imagenet classification », *in*: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[49] Herbert Robbins and Sutton Monro, « A stochastic approximation method », *in*: *The annals of mathematical statistics* (1951), pp. 400–407.

[50] Jack Kiefer and Jacob Wolfowitz, « Stochastic estimation of the maximum of a regression function », *in*: *The Annals of Mathematical Statistics* (1952), pp. 462–466.

[51] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, « Learning representations by back-propagating errors », *in*: *nature* 323.*6088* (1986), pp. 533–536.

[52] Boris T Polyak, « Some methods of speeding up the convergence of iteration methods », *in*: *Ussr computational mathematics and mathematical physics* 4.*5* (1964), pp. 1–17.

[53] Ilya Sutskever et al., « On the importance of initialization and momentum in deep learning », *in*: *International conference on machine learning*, PMLR, 2013, pp. 1139–1147.

[54] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky, « Neural networks for machine learning lecture 6a overview of mini-batch gradient descent », *in*: *Cited on* 14.*8* (2012), p. 2.

[55] John Duchi, Elad Hazan, and Yoram Singer, « Adaptive subgradient methods for online learning and stochastic optimization. », *in*: *Journal of machine learning research* 12.*7* (2011).

[56] Diederik P Kingma and Jimmy Ba, « Adam: A method for stochastic optimization », *in*: *arXiv preprint arXiv:1412.6980* (2014).

[57] Kaichao You et al., « How does learning rate decay help modern neural networks? », *in*: *arXiv preprint arXiv:1908.01878* (2019).

[58] Sepp Hochreiter, « Untersuchungen zu dynamischen neuronalen Netzen », *in*: *Diploma, Technische Universität München* 91.*1* (1991), p. 31.

[59] Sunitha Basodi et al., « Gradient amplification: An efficient way to train deep neural networks », *in*: *Big Data Mining and Analytics* 3.*3* (2020), pp. 196–207.

[60] Albert Njoroge Kahira et al., « An oracle for guiding large-scale model/hybrid parallel training of convolutional neural networks », *in*: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 161–173.

[61] Mu Li et al., « Efficient mini-batch training for stochastic optimization », *in*: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 661–670.

[62] Ohad Shamir, « Without-replacement sampling for stochastic gradient methods », *in*: *Advances in neural information processing systems* 29 (2016).

[63] Ragav Venkatesan and Baoxin Li, *Convolutional neural networks in visual computing: a concise guide*, CRC Press, 2017.

[64] Martial Mermillod, Aurélia Bugaiska, and Patrick Bonin, « The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects », *in*: *Frontiers in Psychology* 4 (2013), p. 504.

[65] Gido M Van de Ven and Andreas S Tolias, « Three scenarios for continual learning », *in*: *arXiv preprint:1904.07734* (2019).

[66] Friedemann Zenke, Ben Poole, and Surya Ganguli, « Continual learning through synaptic intelligence », *in*: *International Conference on Machine Learning*, PMLR, 2017, pp. 3987–3995.

[67] Gido M van de Ven, Nicholas Soures, and Dhireesha Kudithipudi, « Continual Learning and Catastrophic Forgetting », *in*: *arXiv e-prints* (2024), arXiv–2403.

[68] Gido M Van de Ven, Tinne Tuytelaars, and Andreas S Tolias, « Three types of incremental learning », *in*: *Nature Machine Intelligence* 4.*12* (2022), pp. 1185–1197.

[69] Roger Ratcliff, « Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. », *in*: *Psychological Review* 97.*2* (1990), p. 285.

[70] Matthew A Wilson and Bruce L McNaughton, « Reactivation of hippocampal ensemble memories during sleep », *in*: *Science* 265.*5172* (1994), pp. 676–679.

[71] Björn Rasch and Jan Born, « Maintaining memories by reactivation », *in*: *Current opinion in neurobiology* 17.*6* (2007), pp. 698–703.

[72] Arslan Chaudhry et al., « Continual learning with tiny episodic memories », *in*: *Workshop on Multi-Task and Lifelong Reinforcement Learning*, 2019.

[73] Yogesh Balaji et al., « The effectiveness of memory replay in large scale continual learning », *in*: *arXiv preprint arXiv:2010.02418* (2020).

[74] David Rolnick et al., « Experience replay for continual learning », *in*: *Advances in Neural Information Processing Systems* 32 (2019).

[75] Pietro Buzzega et al., « Rethinking experience replay: A bag of tricks for continual learning », *in*: *25th International Conference on Pattern Recognition (ICPR)*, IEEE, 2021, pp. 2180–2187.

[76] Rahaf Aljundi et al., « Gradient based sample selection for online continual learning », *in*: *Advances in Neural Information Processing Systems* 32 (2019).

[77] Rahaf Aljundi, Klaas Kelchtermans, and Tinne Tuytelaars, « Task-free continual learning », *in*: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11254–11263.

[78] Hanul Shin et al., « Continual learning with deep generative replay », *in*: *Advances in neural information processing systems* 30 (2017).

[79] Yulai Cong et al., « Gan memory with no forgetting », *in*: *Advances in Neural Information Processing Systems* 33 (2020), pp. 16481–16494.

[80] Rahaf Aljundi et al., « Online continual learning with maximal interfered retrieval », *in*: *Advances in neural information processing systems* 32 (2019).

[81] Xialei Liu et al., « Generative feature replay for class-incremental learning », *in*: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 226–227.

[82] Eli Verwimp, Matthias De Lange, and Tinne Tuytelaars, « Rehearsal revealed: The limits and merits of revisiting samples in continual learning », *in*: *IEEE/CVF International Conference on Computer Vision*, 2021, pp. 9385–9394.

[83] Gido M Van de Ven, Hava T Siegelmann, and Andreas S Tolias, « Brain-inspired replay for continual learning with artificial neural networks », *in*: *Nature communications* 11.*1* (2020), p. 4069.

[84] James Kirkpatrick et al., « Overcoming catastrophic forgetting in neural networks », *in*: *Proceedings of the National Academy of Sciences* 114.*13* (2017), pp. 3521–3526.

[85] Hippolyt Ritter, Aleksandar Botev, and David Barber, « Online structured laplace approximations for overcoming catastrophic forgetting », *in*: *Advances in Neural Information Processing Systems* 31 (2018).

[86] Timothée Lesort, Andrei Stoian, and David Filliat, « Regularization shortcomings for continual learning », *in*: *arXiv preprint arXiv:1912.03049* (2019).

[87] Jeremias Knoblauch, Hisham Husain, and Tom Diethe, « Optimal continual learning has perfect memory and is np-hard », *in*: *International Conference on Machine Learning*, PMLR, 2020, pp. 5327–5337.

[88] Arslan Chaudhry et al., « Using hindsight to anchor past knowledge in continual learning », *in*: *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, 8, 2021, pp. 6993–7001.

[89] Matthew Riemer et al., « Learning to learn without forgetting by maximizing transfer and minimizing interference », *in*: *International Conference on Learning Representations*, International Conference on Learning Representations, ICLR, 2019.

[90] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al., « Distilling the knowledge in a neural network », *in*: *arXiv preprint arXiv:1503.02531* 2.*7* (2015).

[91] AS Benjamin, D Rolnick, and K Kording, « Measuring and regularizing networks in function space », *in*: (2019).

[92] Matteo Boschini et al., « Class-incremental continual learning into the extended derverse », *in*: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.*5* (2022), pp. 5497–5512.

[93] Mehrdad Farajtabar et al., « Orthogonal gradient descent for continual learning », *in*: *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 3762–3773.

[94] Hongjoon Ahn et al., « Uncertainty-based continual learning with adaptive regularization », *in*: *Advances in neural information processing systems* 32 (2019).

[95] Axel Laborieux et al., « Synaptic metaplasticity in binarized neural networks », *in*: *Nature communications* 12.*1* (2021), p. 2549.

[96] Simon Schug, Frederik Benzing, and Angelika Steger, « Presynaptic stochasticity improves energy efficiency and helps alleviate the stability-plasticity dilemma », *in*: *Elife* 10 (2021), e69884.

[97] Jake Snell, Kevin Swersky, and Richard Zemel, « Prototypical networks for few-shot learning », *in*: *Advances in neural information processing systems* 30 (2017).

[98] Mohammadamin Banayeeanzade et al., « Generative vs. discriminative: Rethinking the meta-continual learning », *in*: *Advances in Neural Information Processing Systems* 34 (2021), pp. 21592–21604.

[99] Shuang Li et al., « Energy-based models for continual learning », *in*: *Conference on lifelong learning agents*, PMLR, 2022, pp. 1–22.

[100] Leonardo Dagum and Ramesh Menon, « OpenMP: an industry standard API for shared-memory programming », *in*: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[101] Sharan Chetlur et al., « cudnn: Efficient primitives for deep learning », *in*: *arXiv preprint arXiv:1410.0759* (2014).

[102] Anuj Kalia, Michael Kaminsky, and David G Andersen, « Design guidelines for high performance {RDMA} systems », *in*: *2016 USENIX Annual Technical Conference* (*USENIX ATC 16*), 2016, pp. 437–450.

[103] Salman Salloum et al., « Big data analytics on Apache Spark », *in*: *International Journal of Data Science and Analytics* 1 (2016), pp. 145–164.

[104] Kaiming He et al., « Deep residual learning for image recognition », *in*: *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[105] Ofer Dekel et al., « Optimal Distributed Online Prediction Using Mini-Batches. », *in*: *Journal of Machine Learning Research* 13.1 (2012).

[106] Maurice Herlihy et al., *The art of multiprocessor programming*, Newnes, 2020.

[107] Elad Hoffer, Itay Hubara, and Daniel Soudry, « Train longer, generalize better: closing the generalization gap in large batch training of neural networks », *in*: *Advances in neural information processing systems* 30 (2017).

[108] Priya Goyal et al., « Accurate, large minibatch sgd: Training imagenet in 1 hour », *in*: *arXiv preprint arXiv:1706.02677* (2017).

[109] Yang You, Igor Gitman, and Boris Ginsburg, « Large batch training of convolutional networks », *in*: *arXiv preprint arXiv:1708.03888* (2017).

[110] Samuel L Smith et al., « Don't decay the learning rate, increase the batch size », *in*: *arXiv preprint arXiv:1711.00489* (2017).

[111] Frank Seide et al., « On parallelizability of stochastic gradient descent for speech DNNs », *in*: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing* (*ICASSP*), IEEE, 2014, pp. 235–239.

[112] Jeffrey Dean et al., « Large scale distributed deep networks », *in*: *Advances in neural information processing systems* 25 (2012).

[113] Yanping Huang et al., « Gpipe: Efficient training of giant neural networks using pipeline parallelism », *in*: *Advances in neural information processing systems* 32 (2019).

[114] Noam Shazeer et al., « Mesh-tensorflow: Deep learning for supercomputers », *in*: *Advances in neural information processing systems* 31 (2018).

[115] Deepak Narayanan et al., « PipeDream: generalized pipeline parallelism for DNN training », *in*: *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[116] Shigang Li and Torsten Hoefler, « Chimera: efficiently training large-scale neural networks with bidirectional pipelines », *in*: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[117] Penghui Qi et al., « Zero Bubble Pipeline Parallelism », *in*: *arXiv preprint arXiv:2401.10241* (2023).

[118] Mohammad Shoeybi et al., « Megatron-lm: Training multi-billion parameter language models using model parallelism », *in*: *arXiv preprint arXiv:1909.08053* (2019).

[119] Jeff Rasley et al., « Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters », *in*: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.

[120] Can Karakus et al., « Amazon sagemaker model parallelism: A general and flexible framework for large model training », *in*: *arXiv preprint arXiv:2111.05972* (2021).

[121] Amir Gholami et al., « Integrated model, batch, and domain parallelism in training neural networks », *in*: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 77–86.

[122] Samyam Rajbhandari et al., « Zero: Memory optimizations toward training trillion parameter models », *in*: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–16.

[123] Yuanzhong Xu et al., « Automatic cross-replica sharding of weight update in data-parallel training », *in*: *arXiv preprint arXiv:2004.13336* (2020).

[124] Adam Paszke et al., « PyTorch: An imperative style, high-performance deep learning library », *in*: *Advances in Neural Information Processing Systems* 32 (2019).

[125] Martín Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015, URL: `https://www.tensorflow.org/`.

[126] Ilya Loshchilov and Frank Hutter, « Decoupled weight decay regularization », *in*: *arXiv preprint arXiv:1711.05101* (2017).

[127] Shai Shalev-Shwartz and Shai Ben-David, *Understanding machine learning: From theory to algorithms*, Cambridge university press, 2014.

[128] Christopher J Shallue et al., « Measuring the effects of data parallelism on neural network training », *in*: *Journal of Machine Learning Research* 20.*112* (2019), pp. 1–49.

[129] Chiyuan Zhang et al., « Understanding deep learning (still) requires rethinking generalization », *in*: *Communications of the ACM* 64.*3* (2021), pp. 107–115.

[130] Lucas Meyer et al., « High Throughput Training of Deep Surrogates from Large Ensemble Runs », *in*: *SC 2023-The International Conference for High Performance Computing, Networking, Storage, and Analysis*, ACM, 2023, pp. 1–14.

[131] Haidong Li et al., « Research on overfitting of deep learning », *in*: *2019 15th international conference on computational intelligence and security (CIS)*, IEEE, 2019, pp. 78–81.

[132] Xue Ying, « An overview of overfitting and its solutions », *in*: *Journal of physics: Conference series*, vol. 1168, IOP Publishing, 2019, p. 022022.

[133] Tianle Zhong et al., « RINAS: Training with Dataset Shuffling Can Be General and Fast », *in*: *arXiv preprint arXiv:2312.02368* (2023).

[134] Truong Thao Nguyen et al., « Why globally re-shuffle? Revisiting data shuffling in large scale deep learning », *in*: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2022, pp. 1085–1096.

[135] Qi Meng et al., « Convergence analysis of distributed stochastic gradient descent with shuffling », *in*: *Neurocomputing* 337 (2019), pp. 46–57.

[136] Lijie Xu et al., « In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle », *in*: *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1286–1300.

[137] NVIDIA, *NVIDIA Data Loading Library*, `https://developer.nvidia.com/DALI`, 2023, (visited on 09/22/2023).

[138] Robert B Ross et al., « Mochi: Composing data services for high-performance computing environments », *in*: *Journal of Computer Science and Technology* 35 (2020), pp. 121–144.

[139] Massimiliano Fatica and Gregory Ruetsch, « CUDA Fortran for scientists and engineers », *in*: *Best Practices for Efficient CUDA Fortran Programming; Elsevier Inc./Morgan Kaufmann: Waltham, MA, USA* (2014).

[140] Oriol Vinyals et al., « Matching networks for one shot learning », *in*: *Advances in neural information processing systems* 29 (2016).

[141] Jia Deng et al., « ImageNet: A large-scale hierarchical image database », *in*: *IEEE Conference on Computer Vision and Pattern Recognition*, Ieee, 2009, pp. 248–255.

[142] Kaiyu Yang et al., « A study of face obfuscation in ImageNet », *in*: *International Conference on Machine Learning*, PMLR, 2022, pp. 25313–25330.

[143] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, « ImageNet Classification with Deep Convolutional Neural Networks », *in*: *Advances in Neural Information Processing Systems*, ed. by F. Pereira et al., vol. 25, Curran Associates, Inc., 2012, URL: `https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b Paper.pdf`.

[144] Sebastian Farquhar and Yarin Gal, « Towards Robust Evaluations of Continual Learning », *in*: (2018).

[145] Eden Belouadah, Adrian Popescu, and Ioannis Kanellos, « A comprehensive study of class incremental learning algorithms for visual tasks », *in*: *Neural Networks* 135 (2021), pp. 38–54.

[146] Irwan Bello et al., « Revisiting ResNets: Improved Training and Scaling Strategies », *in*: *Advances in Neural Information Processing Systems*, ed. by M. Ranzato et al., vol. 34, Curran Associates, Inc., 2021, pp. 22614–22627, URL: `https://proceedings.neurips.cc/paper_files/paper/2021/file/bef4d169d8bddd17d68303877a3ea945-Paper.pdf`.

[147] Zhuang Liu et al., « A ConvNet for the 2020s », *in*: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022).

[148] Norman P Jouppi et al., « In-datacenter performance analysis of a tensor processing unit », *in*: *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[149] Kai Han et al., « Ghostnet: More features from cheap operations », *in*: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1580–1589.

[150] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda, « Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes », *in*: *arXiv preprint arXiv:1711.04325* (2017).

[151] Léon Bottou, Frank E Curtis, and Jorge Nocedal, « Optimization methods for large-scale machine learning », *in*: *SIAM review* 60.2 (2018), pp. 223–311.

[152] Paulius Micikevicius et al., « Mixed Precision Training », *in*: *International Conference on Learning Representations*, 2018.

[153] Amir R Zamir et al., « Taskonomy: Disentangling task transfer learning », *in*: *IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 3712–3722.

[154] Bernardino Romera-Paredes and Philip Torr, « An embarrassingly simple approach to zero-shot learning », *in*: *Proceedings of the 32nd International Conference on Machine Learning*, ed. by Francis Bach and David Blei, vol. 37, Proceedings of Machine Learning Research, Lille, France: PMLR, 2015, pp. 2152–2161, URL: `https://proceedings.mlr.press/v37/romera-paredes15.html`.

[155] Franck Cappello et al., « Grid'5000: A large scale and highly reconfigurable grid experimental testbed », *in*: *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* IEEE, 2005, 8–pp.

[156] Joeri R Hermans, Gerasimos Spanakis, and Rico Möckel, « Accumulated gradient normalization », *in*: *Asian Conference on Machine Learning*, PMLR, 2017, pp. 439–454.

[157] Mengya Gao et al., « An embarrassingly simple approach for knowledge distillation », *in*: *arXiv preprint arXiv:1812.01819* (2018).

[158] Argonne National Laboratory, *Tike: A toolbox for tomographic reconstruction of 3D objects from ptychography data*, `https://tike.readthedocs.io`, 2023, (visited on 11/24/2023).

[159] Clément YJ Hémonnot et al., « X-rays reveal the internal structure of keratin bundles in whole cells », *in*: *ACS nano* 10.*3* (2016), pp. 3553–3561.

[160] Alexander Björling et al., « Three-dimensional coherent Bragg imaging of rotating nanoparticles », *in*: *Physical Review Letters* 125.*24* (2020), p. 246101.

[161] Martin V Holt et al., « Strain imaging of nanoscale semiconductor heterostructures with X-ray Bragg projection ptychography », *in*: *Physical review letters* 112.*16* (2014), p. 165502.

[162] Yi Jiang et al., « Electron ptychography of 2D materials to deep sub-ångström resolution », *in*: *Nature* 559.*7714* (2018), pp. 343–349.

[163] Kaushik Datta et al., « Computational requirements for real-time ptychographic image reconstruction », *in*: *Applied Optics* 58.*7* (2019), B19–B27.

[164] Zhengchun Liu et al., « TomoGAN: low-dose synchrotron x-ray tomography with generative adversarial networks: discussion », *in*: *JOSA A* 37.*3* (2020), pp. 422–434.

[165] Anakha V Babu et al., « AI-assisted automated workflow for real-time x-ray ptychography data analysis via federated resources », *in*: *arXiv preprint arXiv:2304.04297* (2023).

[166] Leslie N Smith, « Cyclical learning rates for training neural networks », *in*: *2017 IEEE winter conference on applications of computer vision (WACV)*, IEEE, 2017, pp. 464–472.

[167] Zhou Wang et al., « Image quality assessment: from error visibility to structural similarity », *in*: *IEEE transactions on image processing* 13.*4* (2004), pp. 600–612.

[168] Jiayu Wang et al., « Unregularized auto-encoder with generative adversarial networks for image generation », *in*: *Proceedings of the 26th ACM international conference on Multimedia*, 2018, pp. 709–717.

[169] Jeffrey S Vitter, « Random sampling with a reservoir », *in*: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[170] Jihwan Bang et al., « Rainbow memory: Continual learning with a memory of diverse samples », *in*: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 8218–8227.

[171] Moses Charikar et al., « Incremental clustering and dynamic information retrieval », *in*: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 626–635.

[172] Max Welling, « Herding dynamical weights to learn », *in*: *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1121–1128.

[173] Huiyi Hu et al., « One pass imagenet », *in*: *arXiv preprint arXiv:2111.01956* (2021).

[174] Aditya K Menon et al., « A statistical perspective on distillation », *in*: *Proceedings of the 38th International Conference on Machine Learning*, ed. by Marina Meila and Tong Zhang, vol. 139, Proceedings of Machine Learning Research, PMLR, 2021, pp. 7632–7642, URL: `https://proceedings.mlr.press/v139/menon21a.html`.

[175] Prannay Khosla et al., « Supervised Contrastive Learning », *in*: *Advances in Neural Information Processing Systems*, ed. by H. Larochelle et al., vol. 33, Curran Associates, Inc., 2020, pp. 18661–18673, URL: `https://proceedings.neurips.cc/paper_files/paper/2020/file/d89a66c7c80a29b1bdbab0f2a1a94af8-Paper.pdf`.

[176] Hisashi Yashiro et al., « A 1024-member ensemble data assimilation with 3.5-km mesh global weather simulations », *in*: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–10.

[177] Salvatore Cuomo et al., « Scientific machine learning through physics–informed neural networks: Where we are and what's next », *in*: *Journal of Scientific Computing* 92.3 (2022), p. 88.

[178] Aditi Krishnapriyan et al., « Characterizing possible failure modes in physics-informed neural networks », *in*: *Advances in neural information processing systems* 34 (2021), pp. 26548–26560.

[179] Didier Lucor, Atul Agrawal, and Anne Sergent, « Simple computational strategies for more effective physics-informed neural networks modeling of turbulent natural convection », *in*: *Journal of Computational Physics* 456 (2022), p. 111022.

[180] Murray Shanahan, « Talking about large language models », *in*: *Communications of the ACM* 67.2 (2024), pp. 68–79.

[181] Qingxiu Dong et al., « A survey on in-context learning », *in*: *arXiv preprint arXiv:2301.00234* (2022).

[182] Jiawei Chen et al., « Benchmarking large language models in retrieval-augmented generation », *in*: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 16, 2024, pp. 17754–17762.

[183] Machel Reid et al., « Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context », *in*: *arXiv preprint arXiv:2403.05530* (2024).

[184] Ziwei Ji et al., « Survey of hallucination in natural language generation », *in*: *ACM Computing Surveys* 55.12 (2023), pp. 1–38.

[185] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli, « Hallucination is inevitable: An innate limitation of large language models », *in*: *arXiv preprint arXiv:2401.11817* (2024).

[186] Kurt Shuster et al., « Retrieval augmentation reduces hallucination in conversation », *in*: *arXiv preprint arXiv:2104.07567* (2021).

[187] Akari Asai et al., « Reliable, adaptable, and attributable language models with retrieval », *in*: *arXiv preprint arXiv:2403.03187* (2024).

[188] Rulin Shao et al., « Scaling Retrieval-Based Language Models with a Trillion-Token Datastore », *in*: *arXiv preprint arXiv:2407.12854* (2024).

[189] Akari Asai et al., « Task-aware Retrieval with Instructions », *in*: *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 3650–3675.

[190] Fangyuan Xu, Weijia Shi, and Eunsol Choi, « RECOMP: Improving retrieval-augmented LMs with context compression and selective augmentation », *in*: *The Twelfth International Conference on Learning Representations*, 2024.

[191] Tianyu Gao et al., « Enabling Large Language Models to Generate Text with Citations », *in*: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 6465–6488.

[192] Yuexiang Zhai et al., « Investigating the Catastrophic Forgetting in Multimodal Large Language Models », *in*: *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[193] Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan, « Fine-tuned language models are continual learners », *in*: *arXiv preprint arXiv:2205.12393* (2022).

[194] Jisoo Mok et al., « Large-scale lifelong learning of in-context instructions and how to tackle it », *in*: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 12573–12589.

[195] Zihan Zhang et al., « CITB: A Benchmark for Continual Instruction Tuning », *in*: *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 9443–9455.

[196] Jianheng Huang et al., « Mitigating catastrophic forgetting in large language models with self-synthesized rehearsal », *in*: *arXiv preprint arXiv:2403.01244* (2024).

COLLEGE DOCTORAL BRETAGNE — MATHS, TELECOMS, INFORMATIQUE, SIGNAL, SYSTEMES, ELECTRONIQUE

INSA — INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Titre : Tampons de Répétition Distribués pour passer l'Apprentissage Continu à l'Échelle

Mot clés : apprentissage continu, oubli catastrophique, parallélisme de données, tampon de répétition distribué, gestion de données asynchrone, scalabilité

Résumé : L'apprentissage profond est un outil d'extraction d'informations à partir de volumes de données gigantesques. Cependant, lorsqu'ils sont entraînés sur des tâches séquentielles (sans accès au jeu de données complet au début de l'entraînement), les réseaux de neurones souffrent d'*oubli catastrophique*, un phénomène qui donne davantage d'importance aux échantillons récents au détriment des connaissances acquises plus tôt. Cette limitation est problématique pour les applications exploitant des flux de données générés au fil du temps. Il est irréalisable de ré-entraîner des modèles à partir de zéro à chaque fois que de nouveaux échantillons sont ingérés, car cela s'accompagnerait de temps d'entraînement trop élevés.

Dans cette thèse, nous présentons des techniques basées sur la répétition pour passer l'apprentissage continu à l'échelle. Les approches basées sur la répétition utilisent des échantillons représentatifs rencontrés précédemment pendant l'entraînement, afin d'augmenter les futurs minibatches avec. Notre contribution principale porte sur la façon d'allier répétition d'échantillons représentatifs et parallélisme de données, qui est l'une des principales techniques pour passer des workloads à l'échelle sur les systèmes HPC. Nous proposons ainsi un tampon de répétition distribué exploitant de nombreuses techniques de parallélisation, permettant d'améliorer les performances prédictives du modèle sans allonger le temps entraînement.

Title: Distributed Rehearsal Buffers for Continual Learning at Scale

Keywords: continual learning, catastrophic forgetting, data parallelism, experience replay, distributed rehearsal buffer, asynchronous data management, scalability

Abstract: Deep Learning (DL) emerged as a way to extract valuable information from ever-growing volumes of data. However, when trained on sequential tasks i.e., without full access to the dataset at the beginning of the training, typical Deep Neural Networks (DNNs) suffer from *catastrophic forgetting*, a phenomenon causing them to reinforce new patterns at the expense of previously acquired knowledge. This limitation prevents updating models incrementally, which is problematic in many real-life scenarios where the aforementioned datasets are replaced by data streams generated over time. It is unfeasible to train models from scratch every time new samples are being ingested either, as this would lead to prohibitive time and/or resource constraints.

In this dissertation, we present techniques based on rehearsal to achieve Continual Learning at scale. Rehearsal-based approaches utilize representative samples previously encountered during training to augment future minibatches with. The key novelty we address is how to adopt rehearsal in the context of data-parallel training, which is one of the main techniques to achieve training scalability on HPC systems. We design a distributed rehearsal buffer that leverages parallelization techniques, enabling us to improve model performance without increasing training time.