Efficient Distributed Continual Learning for Steering Experiments in Real-Time

Thomas Bouvier, Bogdan Nicolae, Alexandru Costan, Tekin Bicer, Ian Foster, Gabriel Antoniu

▶ **To cite this version:**

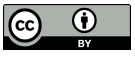
Thomas Bouvier, Bogdan Nicolae, Alexandru Costan, Tekin Bicer, Ian Foster, et al.. Efficient Distributed Continual Learning for Steering Experiments in Real-Time. Future Generation Computer Systems, 2024, pp.1-19. 10.1016/j.future.2024.07.016 . hal-04664176v2

HAL Id: hal-04664176
https://inria.hal.science/hal-04664176v2

Submitted on 23 Aug 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Efficient Distributed Continual Learning for Steering Experiments in Real-Time

Thomas Bouvier[a,*], Bogdan Nicolae[b], Alexandru Costan[a], Tekin Bicer[b], Ian Foster[b], Gabriel Antoniu[a]

[a] *Univ Rennes, CNRS, Inria, IRISA, Rennes, France*
[b] *Argonne National Laboratory, Lemont, Illinois, United States of America*

Abstract

Deep learning has emerged as a powerful method for extracting valuable information from large volumes of data. However, when new training data arrives continuously (i.e., is not fully available from the beginning), incremental training suffers from catastrophic forgetting (i.e., new patterns are reinforced at the expense of previously acquired knowledge). Training from scratch each time new training data becomes available would result in extremely long training times and massive data accumulation. Rehearsal-based continual learning has shown promise for addressing the catastrophic forgetting challenge, but research to date has not addressed 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. We further propose a generalization of rehearsal buffers to support both classification and generative learning tasks, as well as more advanced rehearsal strategies (notably Dark Experience Replay, leveraging knowledge distillation). We illustrate this approach with a real-life HPC streaming application from the domain of ptychographic image reconstruction. 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). Results show that rehearsal-based continual learning achieves a top-5 validation accuracy close to the upper bound, while simultaneously exhibiting a runtime close to the lower bound.

Keywords: continual learning, data-parallel training, experience replay, distributed rehearsal buffers, asynchronous data management, scalability, streaming, generative AI

1. Introduction

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 HPC (High-Performance Computing) [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 every 3.4 months since 2012 [9].

DL models are typically trained on large, many-GPU (HPC) 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 (GPU hours, storage space), leading to both prohibitive runtimes as well as inefficient resource usage.

One approach to this 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 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* [10]. Specifically, the training introduces a bias in favor of new samples, effectively 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

*Corresponding author
Email address: thomas.bouvier@inria.fr (Thomas Bouvier)

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, scalability, and low resource utilization just like incremental training. To address this trade-off, *continual learning* (CL) is gaining popularity in the machine learning community [11, 12]. In a broad sense, CL mitigates catastrophic forgetting by complementing incremental training with a strategy to reinforce patterns seen earlier.

Proposed CL strategies include various methods: *rehearsing* historic training samples, co-training a generative DL model that can mimic old patterns by generating new samples on demand, and regularization (i.e., rules that constrain DL model parameter updates to prevent catastrophic forgetting), among others. We focus here 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 incoming 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 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 procedure. In contrast, other CL approaches require different hyperparameters, additional code to implement regularization, and/or additional generative DL models.

Prior work on rehearsal-based CL [13, 14, 15] has employed a single rehearsal buffer specialized for the learning task (e.g., classification). However, 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, reusable abstraction that implements its own complementary optimizations. Furthermore, *data parallel* training is widely used to reduce the training time. It involves DL model replicas that are trained in parallel with a different data shard, while the gradients are averaged during the backpropagation to keep the replicas in sync. Under such circumstances, it becomes important to enable high-performance, scalable, and resource-efficient rehearsal based CL on **multiple GPUs** that is well adapted to support data-parallel training.

Addressing these two limitations of state of art approaches is challenging for several reasons: (1) the rehearsal buffer needs to store, sample and replace heterogeneous data that is dependent on both the learning task (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), and the rehearsal strategy (e.g., Dark Experience Replay [16] stores additional state information such as activations together with the training samples); (2) 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; (3) it is not enough to simply instantiate independent rehearsal buffers associated with each DL model replica to enable data parallelism: instead, scalable distributed techniques are needed to enable the rehearsal buffers to collaborate at global level in order to avoid biases introduced by localized sampling.

This article describes a novel *distributed* rehearsal buffer abstraction that aims to solve the challenges mentioned above. To this end, it relies on a generic asynchronous engine capable of storing, removing and exposing annotated tuples of tensors efficiently at scale using RDMA (Remote Direct Access Memory [17]) operations for efficiency. This engine can be easily adapted to both classification and generative learning tasks, as well as different rehearsal strategies in order to transparently augment the minibatches assembled by the data pipelines of data-parallel CL training instances. Early results of this work presented in [18] have laid the foundation for the design and implementation of such a distributed rehearsal buffer that is specialized for classification tasks. Specifically, this work focused on realizing a seamless asynchronous integration with the data pipeline, providing scalable support for global sampling under data parallelism, and demonstrating the benefits using classification learning tasks commonly used in the image classification ML community (ecosystem around ImageNet and ResNet-50). This article extends the previous paper with several new contributions: (1) support for heterogeneous data in the rehearsal buffer served in the form of annotated tuples, which extends its applicability to generative CL learning tasks and more advanced rehearsal strategies (notably Dark Experience Replay, leveraging knowledge distillation); (2) integration with a real-life scientific application (PtychoNN [19]) that benefits from generative CL; (3) experimental evaluation of the benefits of rehearsal for PtychoNN using our approach in comparison of several other state-of-art baselines. Overall, we summarize our contributions as follows:

- We motivate the benefits of continual learning for streaming based scientific applications, illustrating them in the context of ptychographic reconstruction that makes use of generative DL models (Section 2).
- We define the concept of rehearsal buffers to address continual learning, and introduce **extensions to leverage them for data-parallel training** (Section 5).
- 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 5.4).
- We propose a generalization of rehearsal buffers capable of storing tensors holding representative training samples and their associated states, which addresses the need to support both classification and generative learning tasks, as well as different rehearsal strategies (Section 6).
- We implement a **distributed rehearsal buffer prototype**

that we integrated with PyTorch, a popular AI runtime (Section 7).

- We report on extensive experiments using 128 GPUs of the ANL's ThetaGPU supercomputer for the case of classification learning 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 (Section 8).
- We report on experiments for the case of generative learning tasks illustrated through a real-life HPC streaming application: ptychographic image reconstruction [20]. To this end, we showcase results obtained using the PtychoNN [19] model in the context of CL for two different types of rehearsal (simple random selection and Dark Experience Replay), which compare favorably to CL based on incremental training and to conventional reconstruction based on algorithms that are computationally intensive (Section 9).

2. Motivating Scenario: Continual Learning in Support of Streaming Applications

As an increasing number of real-world applications produce data continuously, this opens a challenge: how to adapt and update models over time, to effectively capture evolving patterns and trends as they occur? Under this streaming paradigm, we are interested in minimizing the latency between the data acquisition and the production of subsequent insights.

This is particularly true for modern scientific HPC applications that need to **simultaneously** train and run inferences on DL models that need to adapt to highly dynamic patterns. For these applications, the ability to quickly update the DL model in response to new training data is critical. 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 training data over time makes each training procedure (using the current experimental data and all accumulated data so far) take longer (following a cubic increase), consuming more resources. Thus, there is a need to train and update DL models directly from streaming data, without accumulating it. In this section, we discuss a general recurring pattern in modern HPC applications that showcases these challenges. Then we illustrate its real-life application in the case of ptychographic image reconstruction.

2.1. HPC Producer-Consumer Stream Processing

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. However, conventional algorithms and simulations used for data analytics in these workflows are computationally intensive, posing a significant challenge to process massive data volumes with low latency. 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 might not even 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, partial discarding, or feature detection via light computational stages. Unfortunately, such data reduction techniques imposed by limited computational capabilities of the edge infrastructure reduce the quality of end results.

To address these challenges, 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 executing a classic, computationally expensive processing task using 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 simultaneously. As the experiment progresses, if the DL model learns to predict ground truth data with high fidelity, it can be replicated at the edge to use its predictions instead of the main HPC workflow. Although this approach incurs a higher initial overhead due to the need for additional HPC resources to train the model, it ultimately enables to **reduce the latency in generating insights**, with methods based on DL models achieving up to 1000-fold [22] speedups in X-ray imaging.

Furthermore, the DL training procedure might proceed 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. In the most favorable cases, this approach can eventually achieve the quality obtained with the replaced algorithm. Finally, such updates can be used to adjust the course of the experiment, by **steering it in a different direction**, or calibrating the instrument through a feedback loop.

2.2. Zoom on Ptychographic Image Reconstruction

Building on the concept of X-ray imaging, **ptychographic image reconstruction** is a specific application that produces massive streams of data at the edge and processes them on HPC clusters. The goal of this procedure is to reconstruct a high-resolution, two-dimensional representation of a *specimen* (the "sample" under analysis). Specifically, a high-intensity beam produced by a light source is moved along a specimen. 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 cluster where *phase retrieval* is performed using an iterative algorithm such as Tike [23]. For a given diffraction pattern, the result of such an inversion algorithm is a pair of small *(structure, phase)* images corresponding to a local area of the specimen at microscopic level. By stitching these intermediate

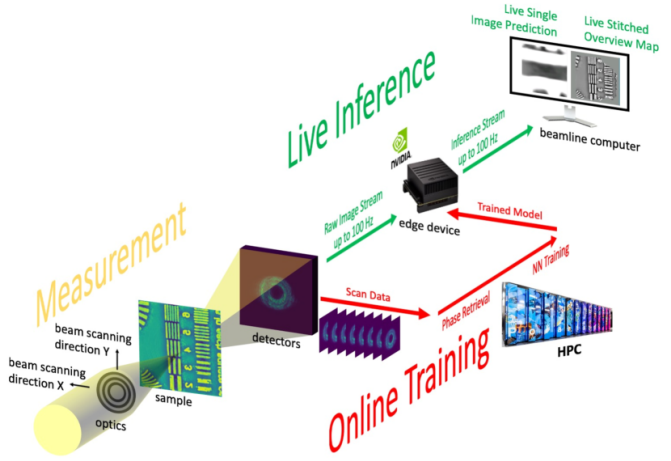


Figure 1: Illustration of a DL-enabled workflow for real-time streaming ptychography imaging, featuring the high-intensity beam produced by the light source, the specimen ("sample") under analysis, the HPC cluster performing the continual training of the DNN, and the edge device for live inference. Figure borrowed from [21].

results together, it becomes possible to reconstruct comprehensive structure and phase images that correspond to the current *perspective* — a specific angle of the specimen under analysis. This procedure is repeated until the entire perspective is covered. The specimen is then rotated to capture new diffraction patterns from a different perspective. These diffractions are sent to the HPC cluster to reconstruct new images, which can be further processed using tomographic reconstruction to generate 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 [24], strain imaging of nanowires [25], and the examination of semiconductor structures through Bragg ptychography [26]. Similarly, electron ptychography has yielded significant breakthroughs, achieving sub-angstrom resolution and nanoscale 3D imaging [27].

2.3. Real-Time Ptychographic Reconstruction using Continual Learning

Iterative algorithms used for phase retrieval are compute-intensive [28], a limitation exacerbated by state-of-the-art ptychography instruments which significantly increase the data acquisition rate. To mitigate this bottleneck, a possible alternative is to rely on faster generative DL models for data analytics [29] (as discussed in Section 2.1). Such models can directly infer the pair of small *(structure, phase)* images corresponding to a diffraction pattern with much lower computational complexity. DL models suitable for this generative learning task have been explored before, with PtychoNN [19] as a prominent example. However, since each studied specimen is different, it is not feasible to pre-train a universal DL model that can extrapolate all possible variations of diffraction patterns. Instead, one can train a DL model on-the-fly (i.e., in an online fashion) 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 conventional phase retrieval (as detailed in Section 2.2), 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 proceeds for the DNN's parameters at the edge to be updated periodically. Continual learning plays a key role to adapt to changing experimental conditions. Besides, the continuous stream of diffraction patterns and their corresponding *(structure, phase)* reconstructions makes 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 keep the DNN up-to-date in real-time. This workflow is illustrated in Figure 1 and is further detailed in [30].

3. Background and Problem Statement

In this section we revisit several key DL concepts to set the context for our work, which we will apply to both a classification benchmark and the ptychography use-case we just described.

3.1. Basics of Deep Learning

Given a probability distribution of the training data $\mathcal{D}$ and a random variable z that takes on values in the space of all possible input-output pairs (x, y), training the DL model defined by parameters w is an iterative process that progressively updates w in order to predict ground truth value y (e.g., a class id) given an input x (e.g., an image). More formally, it can be cast as an optimization problem: we want to find w^* that minimizes the loss function $\mathcal{L}$:

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

where $\mathcal{H}$ is the hypothesis set containing all possible combinations of parameters w, $f_w(x)$ 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 (x, y).

To solve this optimization problem, the most commonly used technique is gradient descent (GD). The DL model is constructed as a composition of functions (parameterized by w) for which derivatives are easily computed. Starting with an initial w chosen randomly, a *forward pass* computes the intermediate predictions for each training sample in the chain of composed functions, after which a *backward pass* computes the intermediate gradients, which can be used to adjust w in order to come

closer to a minimum of $\mathbb{E}$. This process is repeated iteratively until some termination criteria are satisfied. Since computing the gradients for all training samples at each iteration is expensive, a typical optimization is to compute the gradients for small subsets of training samples called *minibatches* (each minibatch contains b samples), assuming that these cheaper gradients still roughly point in the same direction. To increase the likelihood of the cheaper gradients pointing in the same direction, the training data is revisited repeatedly over multiple *epochs*. Each of these 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. This process is called stochastic gradient descent (SGD).

3.2. Data Parallelism

The iterative nature of SGD makes it difficult to parallelize efficiently. A typical optimization used in practice is to create multiple DL model replicas, each of which is trained at the same time on a different shard of the training data [31]. The forward and backward passes can then proceed independently, except that after each backward pass, the gradients computed by all replicas are averaged (by using a collective communication pattern such as `allreduce`) before adjusting 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).

3.3. Catastrophic Forgetting

Although efficient on static training data, SGD does not perform well when training data arrives over time. For example, if a new training dataset $\mathcal{D}'$ is available in addition to $\mathcal{D}$, but we sample new minibatches only from $\mathcal{D}'$, then our DL model will drift in the direction of minimizing $\mathbb{E}'$ (the overall error corresponding to $\mathcal{D}'$), which may no longer be representative of $\mathbb{E} + \mathbb{E}'$. This phenomenon is known as *catastrophic forgetting*. We call each new training dataset $\mathcal{D}'$ a new *task* (sometimes referred to as *data increments*). This is where continual learning (CL) comes in — a paradigm that aims to mitigate catastrophic forgetting when training the model on a sequence of T tasks, denoted as $\{1, \ldots, t_T\}$ [32]. Given T tasks and their probability distributions of data $\mathcal{D}_t$, the optimization problem becomes:

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

Catastrophic forgetting exemplifies the more general plasticity-stability dilemma [33], where (1) plasticity refers to the ability of the model to learn concepts from the current task, and (2) stability refers to its ability to preserve knowledge acquired in previous tasks.

3.4. Domain-incremental CL vs. Class-incremental CL

Two popular CL settings are domain-incremental ("Domain-IL") and class-incremental ("Class-IL") learning. In Domain-IL, the underlying problem structure remains the same, but the context varies over time, resulting in domain shifts. The output space $\mathcal{Y}$ is fixed across tasks. In contrast, Class-IL involves incrementally learning to discriminate between an increasing number of classes, where the output space $\mathcal{T} \times \mathcal{Y}$ changes from one task to another. Note that Class-IL is a more general and challenging setting, as it requires the model to discriminate between classes that were not observed together. For this reason, in this article we aim to support class-incremental CL. Besides, generative learning tasks can be considered as a special case of domain-incremental learning, as the output space is flat (i.e., all outputs belong to a single class).

3.5. Streaming CL vs. Batched CL

In the streaming CL formulation, the model is not allowed to visit a training sample more than once. The input data is fed as a stream and training samples are lost if not accumulated on persistent storage. This is sometimes referred to as the *single epoch* [34] setting. In contrast, batched CL allows unrestricted access to the training samples of the current task (but not those of previous tasks). In this case, the CL training involves *multiple epochs* per task, just like a regular training, which means training samples from the current task are revisited at each epoch. Thus, the assumption is that there is enough space to store the training samples of a single task. In this work, we target batched CL as the multi-epochs setting leads to easier-to-read results by mitigating the underfitting caused by a single or few iterations [16].

3.6. Problem Statement

We recall that retraining the model from scratch on all previously accumulated data is not feasible, because each epoch would yield an increasing number of minibatches as more tasks are being learnt. This would cause long delays until the DL model is ready for inferences after each task. Furthermore, this would cause an explosion of storage space needed to retain the history of all training samples. Our goal is to devise scalable CL techniques that retain an accuracy close to the train-from-scratch approach (which is the upper bound), while simultaneously achieving a runtime close to incremental training (which is the lower bound). The main research question we aim to answer is **how to combine rehearsal-based continual learning with data-parallel training** in order to achieve this goal.

4. Related Work

Experience Replay (that we refer to as *rehearsal*) is a simple continual learning technique in which the model knowledge is reinforced by replaying samples from previous tasks [35, 36]. These methods selectively store 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. The augmentation involves appending a fixed number of representatives to each minibatch corresponding to the new training data, in order to obtain a large minibatch that mixes new and old training samples. When

learning the current task t_c, we seek to minimize the following objective to preserve the knowledge acquired on previous tasks $\{1, \ldots, t_c - 1\}$, $t_c \leq T$:

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

The advantage of this approach is that it can mitigate catastrophic forgetting transparently [37], without the need to change existing training methods. This claim is supported by works that not only emphasize its efficacy compared to alternative methods [38], but also propose diverse extensions to enhance its performance [39]. Researchers devised optimized strategies for selecting and storing representatives based on their gradients [40] or training loss [39], as well as alternative sampling policies from the buffer [41]. Hindsight Anchor Learning (HAL) [42] complements Experience Replay with regularization to align the model responses with data points encoding classes encountered in previous tasks. This approach bears similarity with meta-learning approaches [43], which aim to maximize transfer learning capabilities. Dark Experience Replay (DER and DER++) [16] demonstrate that replaying model activations instead of ground truth labels (or doing both) yields to a better achieved accuracy than Experience Replay alone. eXtended-DER [44] takes an extra step over previous methods by preparing future classification heads to accommodate future classes. Other researchers discussed some limitations of rehearsal too [45].

Data Management Techniques for Training Data. Reading the training data directly from a shared repository (such as a parallel file system) has been shown to introduce significant bottlenecks [46]. This happens even when using asynchronous data pipelines that assemble new minibatches in the background (such as NVIDIA DALI [47]), while overlapping with the training iterations. Quiver [48] uses SSDs to cache training samples so as to avoid slow data access from remote storage. Cerebro [49] partitions training data across nodes in a cluster for data-parallel training without shuffling, instead moving the DL models between compute nodes. However, this does not accelerate single DL training instances and introduces high overheads for large DL models. DeepIO [50] uses a partitioned caching technique for data-parallel training, relying heavily on RDMA for high performance I/O. DIESEL [51] deploys a distributed cache across compute nodes to handle multiple DL training instances sharing the same training data. MinIO [52] focuses on eviction-free caching of training data, which has low overheads and is easy to implement but may lead to higher miss rate. NoPFS [53] introduces a performance model that can leverage multi-level node-local storage for distributed caching of training samples. Lobster [54] further refines this approach by optimizing cache evictions and by enabling load balancing in the data pipeline. Such approaches optimize the data pipeline and complement our proposal.

Positioning. In this work, 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. To our best knowledge, we are the first to explore this direction.

Table 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
c	number of candidates per minibatch
b	minibatch size (number of samples per minibatch)
r	number of representatives added to augmented minibatches

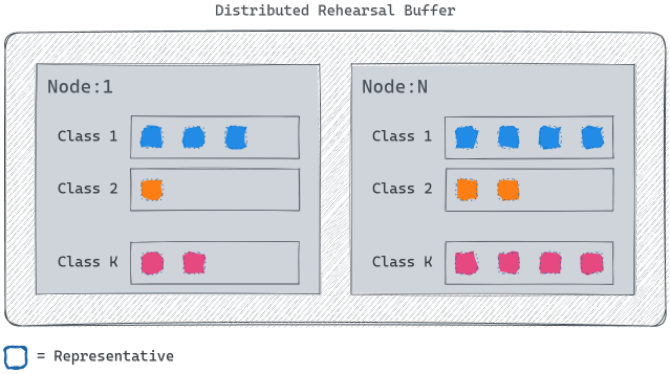


Figure 2: For every process n, a rehearsal buffer $\mathcal{B}_n$ contains representatives from the classes seen so far. The distributed rehearsal buffer $\mathcal{B}$ contains representatives from the K classes.

5. Distributed Rehearsal Buffers

In this section, we introduce the key design principles behind our main contribution: a distributed rehearsal buffer that enables scalable data-parallel training for continual learning.

5.1. Aggregated Memory Space of Rehearsal Buffers

In a basic version of rehearsal, a buffer $\mathcal{B}_n$ stores *representative* training samples from previous tasks. Every class i observed so far is attached to its own memory space $R_n^i \in \mathcal{B}_n$. 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. This new minibatch is an *augmented minibatch*. After training with the 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 R_n^i memories 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 1 provides a summary of the notation used to characterize rehearsal buffers.

Starting from this basic version, we propose to design a *distributed rehearsal buffer* that can be used with data-parallel training. In our case, the training uses N distributed processes (each attached to a dedicated GPU). Each process maintains its own local rehearsal buffer $\mathcal{B}_n$. Thus, we can leverage the aggregated spare memory provided by a large number of compute nodes to store more representatives compared with a single centralized buffer. 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 2.

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

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 $|R^i|_{max} = N \times S_{max}/K$, which increases the number of representatives stored per class and therefore the diversity and quality of the minibatch augmentation. This complements data-parallel training well, since data-parallel training improves performance and scalability, not the quality of the results.

5.2. Selection and Eviction Policies

Since the rehearsal buffer $\mathcal{B}$ is smaller than the dataset $\mathcal{D}$, we are concerned about 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 1, 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 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 [55] and demonstrates low computational complexity.

Since representatives are distributed among R_n^i according to their class labels, the competition between new candidates and stored representatives is done by class. Thus, candidate samples belonging to a specific class compete against the existing representatives of the same class. As depicted in Figure 3, a candidate sample of class i replaces a random representative in R_n^i if the latter is full. 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. Thus, we *independently* obtain a good mix of old and new training samples in each memory space R_n^i. This approach both increases the quality of the augmentations

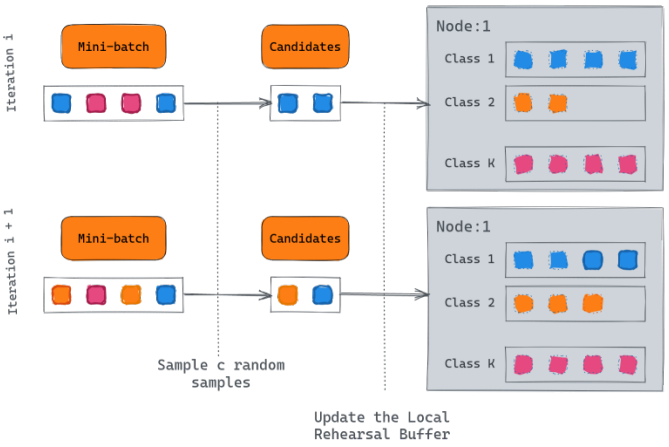


Figure 3: For a given process n, c candidates from the incoming minibatch are sampled and used to populate $\mathcal{B}_n$. If the buffer for class i is full, representatives from R_n^i are replaced at random. The figure depicts the rehearsal buffer $\mathcal{B}_n$ state for two subsequent iterations for $c = 2$.

Algorithm 1: Rehearsal buffer updates with new candidates for each process n

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

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

More sophisticated policies than random sampling might further improve the achieved accuracy. In that sense, the distributed rehearsal buffer could leverage gradient-based selection [40] or loss-based selection [39] of representatives to be stored. Eviction policies could leverage herding selection to maintain their distribution [56], reservoir sampling [43, 57], or FIFO/FIRO strategies (First In, First Out/First In, Random Out) [58]. In this work, we chose to manage representatives per class for simplicity, all these strategies could be applied to the entire buffer, regardless of class. We leave such studies for future work.

5.3. Global Minibatch Augmentation using RDMA-enabled Distributed Sampling of Representatives

Experience Replay consists in interleaving representatives with the current minibatch m to build a new augmented minibatch m'. As depicted in Figure 4, 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 [55, 37],

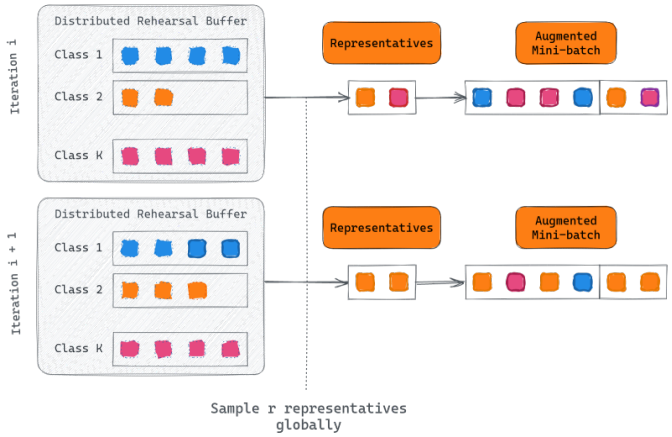


Figure 4: On a given process n, every incoming minibatch is augmented with r representatives sampled randomly and without replacement from the distributed rehearsal buffer $\mathcal{B}$. Here, $r = 2$ on two subsequent iterations. Sampling from $\mathcal{B}$ introduces communication between the N distributed processes.

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, we 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 is a difficult challenge for several reasons: (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.

To address this challenge, 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. Second, since the requests of the processes to sample remote rehearsal buffers are not synchronized (independent servers must react to requests that are not necessarily synchronized), we cannot simply rely on existing patterns such as MPI all-to-all collectives, as this would introduce unnecessary delays. One-sided MPI communication can potentially address this limitation, but it is non-trivial to adopt in a consistent fashion (e.g., ensuring that one-sided *get* operations from the rehearsal buffer do not occur during updates). Therefore, we propose a point-to-point communication pattern atop Mochi [59], an HPC-oriented set of services that provides low-overhead RDMA-enabled remote procedure calls (RPCs). Specifically, we introduce several key concepts: (1) progressive assembly of augmented minibatches using concurrent asynchronous RPCs, which hide the remote access latency; (2) RPC consolidation to transfer the training samples in bulk from the same remote rehearsal buffer, reducing the number of RPCs; (3) 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.

5.4. 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.

To this end, we revisit the major steps of CL based on rehearsal and data-parallel training: ① prepare the augmented minibatches, which involves global sampling from the distributed rehearsal buffer; ② update the distributed rehearsal buffer using the new samples of the original minibatches; ③ perform a forward pass with the augmented minibatch as input data; ④ perform a backward pass that averages the gradients and updates the parameters w of each DL model replica. One key observation is that we start with an empty rehearsal buffer. Hence, for the first training iteration we do not need to perform an augmentation. However, after step ②, we can prepare an augmented minibatch in advance for the next training step. This applies for all subsequent iterations.

Therefore, we can use the following strategy: ① 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; ② 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 ③ and ④ as above. This process is illustrated in Figure 5.

Using this approach, the communication and synchronization overheads related to the management of the rehearsal buffer can be overlapped with the training steps. The training iterations only need to wait if the updates of the buffer and the global sampling cannot keep up with it and introduce a delay. It is important to note though that even in cases where the rehearsal buffer overhead can be fully absorbed asynchronously (i.e., no wait at step ①), 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. 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.

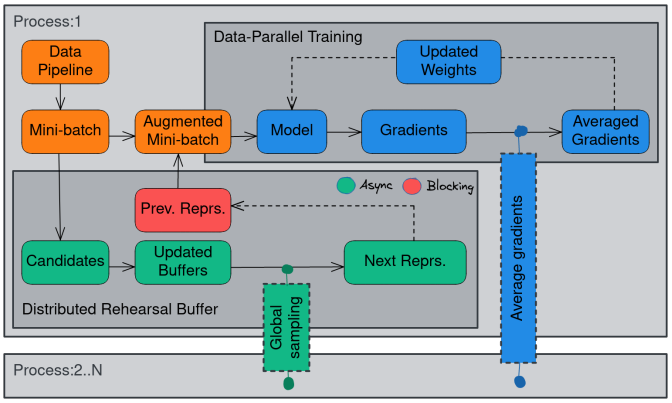


Figure 5: Asynchronous updates of the rehearsal buffers and global augmentations: r representatives sampled globally beginning with the previous iteration are used by the training loop to assemble an augmented minibatch on each process n. Meanwhile, the distributed rehearsal buffer extracts candidates from the current minibatch to update each B_n locally, then collects the next r representatives using global sampling.

6. Towards Generic Distributed Rehearsal Buffers

With a growing diversity of rehearsal techniques, it becomes important to decouple the buffer from the learning task, such that it becomes a generic abstraction that can store additional state information as needed by more advanced rehearsal-based CL algorithms. To this end, we extend the rehearsal buffer presented in Section 5 to store heterogeneous data alongside representatives, and serve it efficiently using annotated tuples. Furthermore, the rehearsal buffer interacts with the data pipeline directly, independently from the training procedure itself (beyond slight modifications to the loss function). This enables not only transparent augmentation of minibatches, but also a flexible integration with different experience replay strategies, as discussed next.

6.1. Extending Experience Replay using Knowledge Distillation

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 [16] propose *Dark Experience Replay* (DER), an algorithm applying the teacher-student approach to encourage the DNN model to mimic the *neural activations* denoted a triggered when learning previous tasks. Specifically, these activations are stored in the rehearsal buffer alongside the associated representative training samples x. Differently from vanilla Experience Replay, DER does not sample input-label (x, y) tuples from the buffer but 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 5.3. However, past data samples are not reinjected into the training procedure 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. No rehearsal of past training samples is occuring with DER alone. Using the same notation as in [16], we indicate the neural activations with $h_w(x)$ and the model predictor is defined as $f_w(x) \triangleq \text{softmax}(h_w(x))$. 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 a representative sample x:

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

where α is a hyperparameter introduced by DER setting the importance of the distillation term. Leveraging the selection policy introduced in Section 5.2, c candidate input-activation tuples are pushed into the buffer at every training iteration, with the competition still being done per class. Activations a corresponding to a past representative x could become outdated while learning new tasks. However, the authors observed that this selection strategy occurring along the optimization trajectory does not degrade the achieved accuracy.

Algorithm 2: Dark Experience Replay++ (borrowed from the original implementation [16]) with $\beta = 1$.

1 $\mathcal{B} \leftarrow \{\}$
2 **for** $i = 0$ ***to*** $\frac{|\mathcal{D}|}{b} \cdot epochs$ **do**
3 $\quad (x, y) \leftarrow m \leftarrow \texttt{sample}(\mathcal{D}, b)$
4 $\quad a \leftarrow h_w(x)$
5 $\quad (x'', a'') \leftarrow \texttt{get_representatives}(\mathcal{B}, r)$
6 $\quad dis \leftarrow \alpha \cdot \|a'' - h_w(x'')\|_2^2$
7 $\quad (x', y') \leftarrow m' \leftarrow \texttt{augment}(m)$
8 $\quad w \leftarrow w - \eta \cdot \nabla_w [\ell(f_w(x'), y') + dis]$
9 $\quad \mathcal{B} \leftarrow \texttt{update_buffer}((x, y, a), c)$

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 [16] propose a variant named DER++, combining the rehearsal of representatives stored in the buffer with neural activations in the loss calculation. As such, DER++ optimizes the following objective:

$$\mathcal{L}_{t_c} + \beta \cdot \mathbb{E}_{(x'',y'')\sim\mathcal{B}} \left[\ell(f_w(x''), y'') \right] + dis$$

where dis denotes the distillation term added by DER, and β denotes an hyperparameter introduced by DER++ setting the importance of the rehearsal term. We decide to set this parameter to $\beta = 1$, so that representatives to be rehearsed do not have a higher weight than original samples from minibatch m when training. This choice allows to perform a single forward pass on an augmented minibatch m'', reusing all concepts introduced in Section 5.1. Algorithm 2 describes the training procedure: after sampling a minibatch m from dataset $\mathcal{D}$ (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 w (line 6). An augmented minibatch m' is also prepared for rehearsal as detailed in Section 5.3. Finally, model parameters are updated considering the loss obtained from m' and dis. Variable η denotes the learning rate.

6.2. Annotated Tuples

The additional state information brought by activations is held alongside its associated representative training samples in the distributed rehearsal buffer. This generic design makes the implementation of more advanced rehearsal-based CL algorithms straightforward, while taking advantage of the high-performance global sampling introduced in Section 5.3. From the training loop standpoint, such states are served as annotated tuples of tensors of arbitrary shapes. For instance, line 5 serves a tuple (x, a) holding representatives and associated activations. Such tensors are transferred from the buffer 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.

6.3. Application to Generative DL Workloads

The proposed rehearsal buffer is aware of classes when storing representative samples, which is pertinent to classification models. 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 Large Language Models (LLMs) for instance, this only class id would correspond to representative historical "*embeddings*". Similarly, this class corresponds to "*diffraction patterns*" in the ptychography imaging use-case described in Section 2.3. 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 genericity 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.

Another key difference of generative workloads 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 id. With generative workloads however, the shape of the ground truth is typically more complex as it corresponds to the expected prediction of the DNN model, whether it is text, an image, or otherwise. For LLMs, this could correspond to a set of human-generated textual annotations representing an acceptable output. For ptychography imaging, the output of the DNN is a pair of *(structure, phase)* images, produced from a diffraction pattern as input. Alongside diffraction patterns stored as representatives, two tensors retain the corresponding ground truth data (structure and phase reconstructions), which are then served to the AI runtime's training loop using an annotated tuple. The generic design of our distributed rehearsal buffer allows to handle all these scenarios by storing additional data of any shape in the buffer. A possible complex buffer configuration is depicted in Figure 6, illustrating the flexibility of our approach to accommodate a large range of deep learning workloads. In this case, all local buffers $\mathcal{B}_n$ contain representatives of the same class.

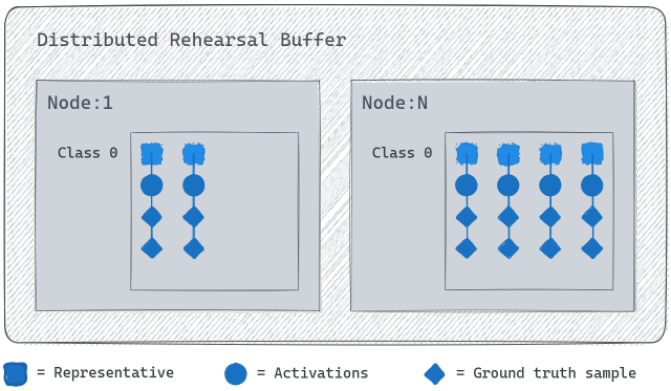


Figure 6: With generative workloads, representatives of a single class are stored in the rehearsal buffer $\mathcal{B}_n$. This buffer configuration then replaces the one shown in Figure 2. Our generic design allows to hold additional information associated to these training samples: in this example, activations and two ground truth tensors are retained for each of them. This data are served using global sampling to be made available to the training process alongside representatives.

7. Implementation Details

We implemented our approach as a high-performance C++ library that offers convenient bindings for Python using *pybind11*. There are multiple reasons for this choice: (1) our approach requires low-overhead access to system-level resources, notably RDMA-enabled remote procedure calls (RPCs), which is not available for Python; (2) even if bindings existed, the overheads of interpreted languages are unacceptable in our case given the need to provide consistency and manage multiple connections under concurrency; (3) Python has limited support for multi-threaded concurrency due to the global interpreter lock that allows only a single thread to run interpreted code. Nevertheless, the complexity of our proposal is completely hidden from end-users: the distributed rehearsal buffer integrates seamlessly with the training loop using a convenient *update* primitive encapsulating all our contributions, illustrated in Listing 1.

For the purpose of this work, we integrate our proposal with PyTorch [60] and rely on Horovod [61] to enable data parallelism. We rely on NVIDIA DALI [47] 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 [62]), data-parallel implementations or data pipelines.

```
1 for i in range(no_minibatches):
2   m = DataPipeline.get_next_minibatch()
3   r = RehearsalBuffer.update(m)
4   m_a = concat(m, r)
5   Model.train(m_a)
```

Listing 1: Example of a training loop integrating our proposal. The *update* primitive (highlighted) waits until r representatives were collected by the asynchronous global sampling, then updates the rehearsal buffer, and starts the next global sampling.

To take advantage of high-performance, fine-grain parallelism, all operations are executed in a separate system pthreads and CUDA operations involving copies between GPUs and host memory are executed in a dedicated CUDA stream isolated from the Python frontend. We used Argobots (part of the Mochi framework [63]) for implementing a low-overhead, userspace thread pool that serves concurrent requests to update and read the training samples from rehearsal buffers. We used Thallium (also part of Mochi) to implement global sampling leveraging non-blocking RDMA-enabled RPCs.

Our implementation is publicly available as an **open-source project** [64].

8. Evaluation: Rehearsal for Classification Models illustrated with ImageNet Benchmarks

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

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

8.1. Methodology

Training Dataset: we use the ImageNet-1K [65] dataset, which is widely used in the image classification community. We specifically use the variant with face-blurred images [66], containing 1.2M training images split among 1000 object classes. 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.

Continual Learning Scenario: we recall that we focus on the class-incremental ("Class-IL") scenario, in which there are clear and well-defined boundaries between the tasks to be learned (i.e., there is no overlap between classes of different tasks). We design a sequence of 4 disjoint tasks, each containing 250 classes from ImageNet. Each of them gets revisited 30 times (i.e., the model is trained for 30 epochs on every task), which corresponds to a total of 120 training epochs. The model can not revisit previous tasks.

Learning Models: to show that our rehearsal-based approach to CL is model-agnostic, we use the 3 following convolutional networks and their corresponding configurations:

- **ResNet-50** [67] is the standard with ImageNet. We use the SGD optimizer with a learning rate of 0.0125; a per-task learning rate increase on 5 warmup epochs as in [68]; 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.
- **ResNet-18** [67] has roughly half the number of parameters of ResNet-50 and is thus faster to train (i.e., its minibatch processing time is shorter). We use the same hyperparameters as ResNet-50.
- **GhostNet-50** [69] implements a different architecture to minimize inference time on resource-constrained devices. We use the SGD optimizer with a learning rate of 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 [70] to speed up the training.

Scale: we apply the linear scaling rule [68] by multiplying the learning rate with the number of processes N. However, with an augmented minibatch size set to $b + r = 63$ training samples, which corresponds to a global batch size of $N \times 63$ in our data-parallel setting, the latter becomes greater that 8K with $N = 128$. This requires further consideration to mitigate the instability introduced by such large batches [68, 71]. We do so by setting a maximum rate independent of the minibatch size equal to 64, as suggested theoretically in [72].

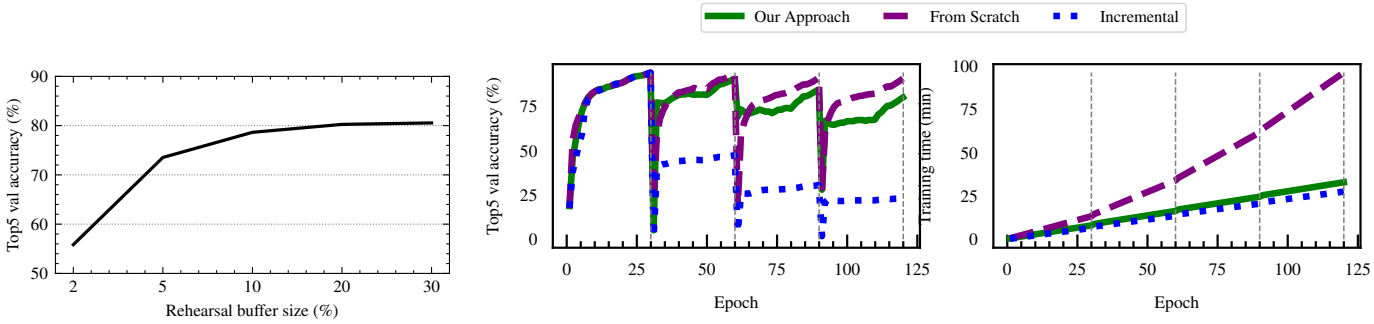
Performance Metrics: we report the top-5 accuracy achieved on the validation set to measure the model performance. Top-5 accuracy means any of the model's top 5 highest probability predictions is considered as correct. Let $a_{i,j}$ denote the top-5 evaluation accuracy on task j using the model snapshot obtained at the end of task i. The accuracy (fraction of correct classifications) assessing the DL model performance on all previous tasks is defined as follows:

$$accuracy_T = \frac{1}{T} \sum_{j=1}^{T} a_{T,j} \tag{1}$$

Computing Environment: we run our experiments on up to 16 nodes of ANL's ThetaGPU supercomputer (128 GPUs). Each node comprises eight NVIDIA A100 GPUs (40 GiB) and two AMD Rome CPUs. We use the following software environment: 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 as well as libfabric 1.16 compiled with CUDA support.

8.2. Impact of the Rehearsal Buffer Size on Accuracy

As detailed in Section 5.1, distributing the training across N processes allows to leverage the aggregated memory to store more representatives in the rehearsal buffer $|\mathcal{B}| = N \times |\mathcal{B}_n|$. 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 $10\%/N$ of the data per process). To showcase the effect of different rehearsal buffer sizes on the accuracy, we vary $|\mathcal{B}|$ from 2.5%, 5%, 10%, 20%, to



(a) 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.

(b) $|\mathcal{B}| = 30\%$ 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 7: Top-5 accuracy for ResNet-50, 16 GPUs, ImageNet (4 tasks).

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.

We measure the performance of our approach with different rehearsal buffer sizes by applying Equation 1 once at epoch 120 (end of the training), in order to evaluate the DL model on all previous tasks i.e., on all the classes seen until then. We consider only ResNet-50 for this study, and run these experiments on 2 nodes (16 GPUs). We report the results in Figure 7a. 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 $N = 16$), which is only a fraction of the memory available on typical HPC systems (512 GB of host memory per compute node).

8.3. Impact of Other Rehearsal-related Hyperparameters

Parameter c (introduced in Section 5.2) 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 impacts the renewal rate of representatives from the current task.

Parameter r (introduced in Section 5.3) 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. Authors in [55] set r to 15% of the minibatch size b: we adopt a similar ratio, setting $b = 56$ and $r = 7$.

8.4. Comparison with Baseline Approaches

We apply the insights obtained in the experiments detailed in Sections 8.2 and 8.3, and we set $|\mathcal{B}| = 30\%$ and $r = 7$ to achieve high accuracy in the remainder of this paper. The following baselines instantiate models without any regularization or rehearsal:

- **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.

We consider ResNet-50 for this study. In Figure 7b, the top-5 evaluation accuracy achieved by rehearsal (80.55%) greatly outperforms the incremental training baseline. The latter 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% above the accuracy achieved with our rehearsal-based approach.

In Figure 7b, 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 combines the best of both baselines: in terms of accuracy, it is close to the from-scratch training approach, while simultaneously nearing incremental training in terms of training time. This allows to overcome the limitations of the baseline approaches, which sacrifices accuracy for scalability or vice versa.

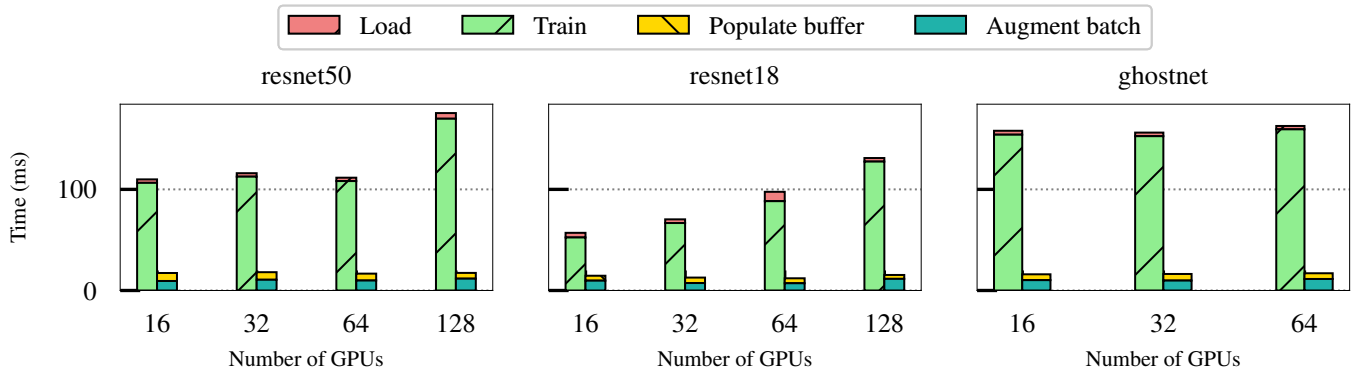


Figure 8: Time breakdown for the training loop and rehearsal buffer management, for each of the three models and for increasing numbers of GPUs, averaged across 35 minibatches. Training iterations sometimes take longer as the time required for communication and synchronization between GPUs increases 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.

8.5. Rehearsal Buffer Management Breakdown

In Figure 8 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.

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 8. 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. 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.

Moreover, 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 all-reduce gradient reductions are expensive and begin to stall the 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.

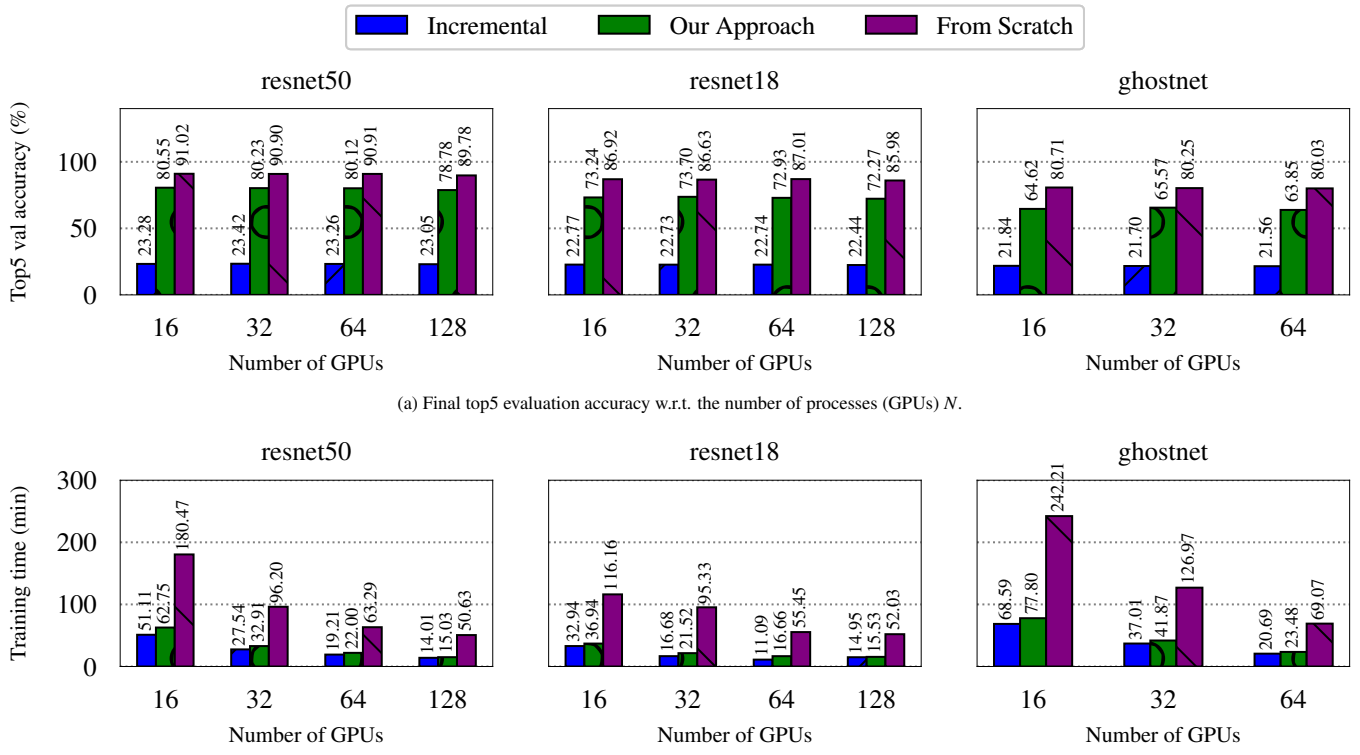
8.6. Scalability

We study the scalability of our approach for all three models compared with the two baselines for an increasing number of data-parallel processes (GPUs). Specifically, we measure the final evaluation top-5 accuracy in Figure 9a, where Equation 1 is applied once at epoch 120. We also measure the overall runtime to train all tasks and depict it in Figure 9b.

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 biases.

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 8.5.

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 8.5. Thus, the runtime of rehearsal-based CL is roughly proportional to r.



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

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

Figure 9: Accuracy and runtime, $|\mathcal{B}| = 30\%$, $b = 56$ and $r = 7$ for all 3 models. For ResNet-50, colors match those in Fig. 7b.

9. Experiments for Generative Models illustrated with Ptychographic Reconstruction

Our next series of experiments are complementary and illustrate the benefits of our proposal for generative DL models. They focus on the generality of the rehearsal buffers, as underlined in Section 6, rather than the scalability on multiple GPUs. To this end, we apply rehearsal to enable continual learning for a real-life HPC streaming application: ptychographic image reconstruction (as described in Section 2.3). In this section, we aim to answer the following questions:

- What benefits does continual learning based on rehearsal bring over incremental training that ignores catastrophic forgetting?
- What is the impact of augmenting simple rehearsal with Dark Experience Replay?
- What is the impact on the end-result if we replace a traditional reconstruction algorithm with a generative DL model?

9.1. Methodology

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 under analysis. The X-ray beam follows a commonly used pattern to scan a perspective, as illustrated in Figure 10, 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 [23], a commonly used phase retrieval algorithm based on conventional analysis techniques. 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 Section 8, we treat all training samples as belonging to a single class as detailed in 6.3. 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. 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

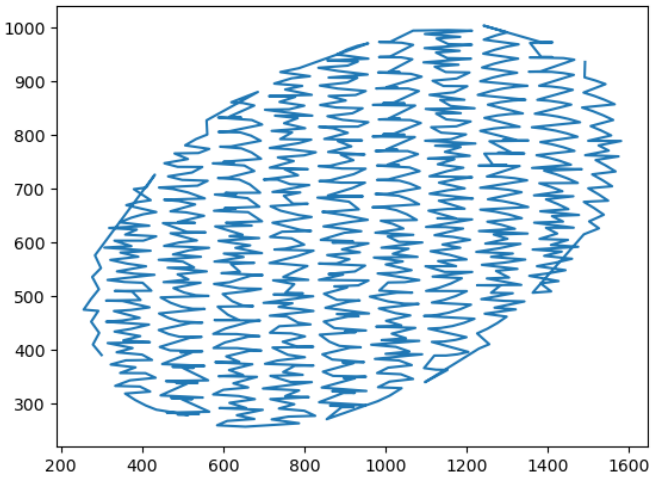


Figure 10: Illustration of the path (left-right, top-bottom) taken by the X-ray beam to scan a single perspective of a specimen. This procedure generates 950 diffraction patterns in the far field, which are then used to compute structure and phase reconstructions using Tike.

validation data, while the rest is used for the actual training.

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 10. 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.

Continual Learning Scenario: the goal of the training procedure is to produce an accurate generative DL model for the specimen under analysis. Each of its perspectives represents a new learning task, whose diffraction patterns are used to update the DL model through continual learning. As all diffraction patterns belong to the same class, this setting characterizes task changes by introducing shifts in the input distribution (corresponding to different perspectives). This domain-incremental ("Domain-IL") scenario complements the experiments presented in Section 8, where we focused on class-incremental learning ("Class-IL").

Learning Model: we use **PtychoNN** [19] 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 consists of 3 parts: an encoder arm that contains convolutional and max pooling layers designed to learn a representation of the input data (diffraction patterns), and 2 decoder arms that contain convolutional and upsampling layers designed to reconstruct real-space images (structure and phase). Since we know the position of the X-ray beam for each diffraction pattern, we 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 of 0.00088. We then follow an *exp_range* learning rate schedule as suggested by [73], 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.

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. Furthermore, to evaluate the quality of the final reconstructed image, we use the Peak-Signal-to-Noise Ratio (PSNR) and the Structural Similarity (SSIM) [74] 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:

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

The PSNR (in dB) is defined as:

$$PSNR(I,R) = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE(I,R)) \quad (3)$$

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

$$SSIM(I,R) = \frac{(2\mu_I\mu_R + C_1)(2\sigma_{IR} + C_2)}{(\mu_I^2 + \mu_R^2 + C_1)(\sigma_I^2 + \sigma_R^2 + C_2)} \quad (4)$$

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.

Computing Environment: we run our experiments on a single A100 GPU of ANL's ThetaGPU supercomputer, with the same software environment as detailed in Section 8.

9.2. 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 vanilla Experience 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).

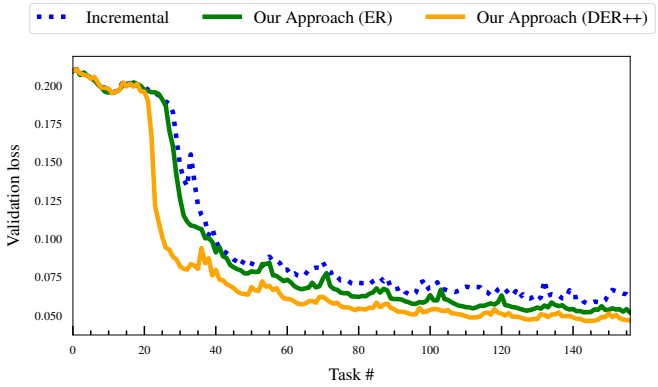


Figure 11: Evolution of the validation loss for an increasing number of tasks.

- **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, which leverages the rehearsal buffer to store input activation tuples for the representatives. 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 11, 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 obtain an accurate DL model sooner. This allows to switch from conventional computations to DL inferences earlier, saving more HPC resources.

9.3. Comparison of Tike vs. PtychoNN Stitched Reconstructions

Our next series of experiments focuses on evaluating the end-impact of our proposal on 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 real-life scenarios in which localized reconstructions are delegated to the DL model at the edge), 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 12. 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 10. 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 [75].

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

To quantify the differences between the final images produced by all four approaches, we use the PSNR and SSIM metrics introduced in Section 9.1. The results are listed in Table 2. 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 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 9.2).

10. Discussion

Efficiency at Scale. The accumulation of representatives in the distributed rehearsal buffers may grow to 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). As in [45], one could suspect that training repeatedly over a limited number of representatives would end up overfitting the rehearsal buffer, which may be an inherent limitation of CL. In this regard, our approach of using a distributed rehearsal buffer enables the aggregated size to grow proportionally with the number of processes. Thus, we retain a large and diverse

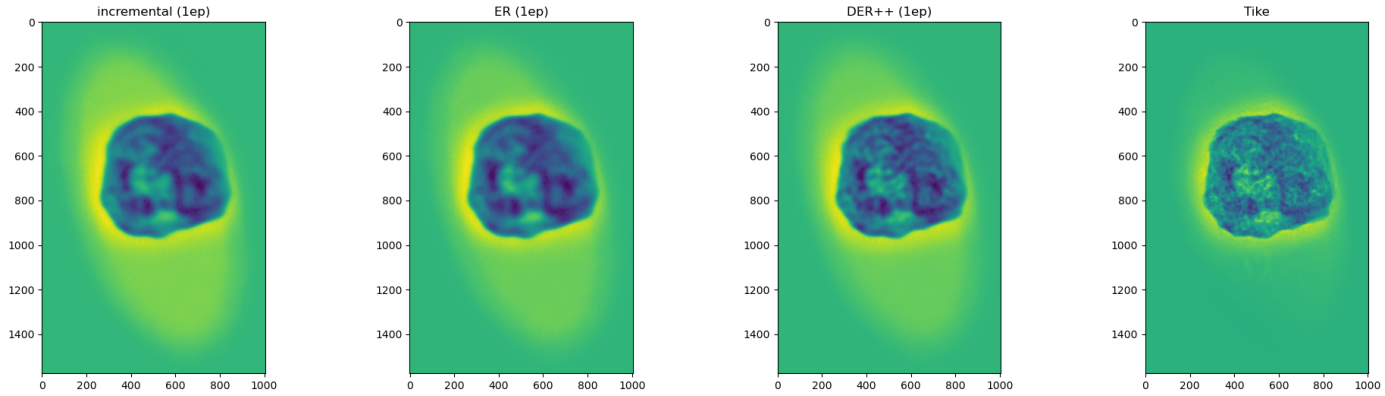


Figure 12: Stitched reconstructions obtained from four different approaches. No task gets revisited during training (one epoch per task). From left to right: incremental training, vanilla Experience Replay, Experience Replay + knowledge distillation (DER++), and the ground truth reconstruction computed by Tike. One can notice a gradual improvement in reconstruction quality.

set of representatives, which increases the quality of continual learning in combination with data-parallel training beyond the limits acknowledged by other state-of-art approaches.

Generality. Our distributed rehearsal buffer stores generic tensors and supports dynamic addition of new classes. By addressing the most difficult scenario (i.e., class-incremental tasks), we can easily adapt our approach to support domain-incremental classification problems (by keeping the number of classes fixed from one task to another) and domain-incremental generative problems (by using a flat output space conceptually equivalent to a single class). Furthermore, the extensions we provided in Section 6 enable heterogeneous data to be served by rehearsal buffers as annotated tuples, which enables the flexibility to use different experience replay strategies. We demonstrated this capability by integrating our rehearsal buffers with Dark Experience Replay (DER++).

Application to Edge-to-HPC streaming applications. Our experiment with ptychographic image reconstruction has shown that advanced experience replay strategies for continual learning such as DER++ can outperform simple experience replay (ER) both in terms of convergence speed and the final accuracy after convergence. In particular, the higher convergence speed enables such streaming applications to replace conventional analysis techniques earlier with equivalent DL models. This lowers the overall resource utilization, and makes better use of the bandwidth between the edge and the HPC cluster in cases where further training of the model is not necessary. While we did not demonstrate this aspect experimentally, we plan to investigate it in future work and to quantify the benefits: first in the context of ptychography, then for other HPC applications that fit the same pattern.

Limitations. We measured the top-5 accuracy to evaluate the performance of the models used in Section 8. The classification community often relies on the top-1 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 4 tasks). In this sense, any accuracy metric seems appropriate. More generally, the novelty of our work lies in the techniques to make rehearsal scalable in the context of data-parallel training. In Section 9, we used the loss to monitor the training, which is appropriate to demonstrate the faster convergence enabled by the conjunction of our distributed rehearsal buffer and DER++. We leave it to the domain experts working on light sources and ptychography imaging to interpret the benefits of the absolute reconstruction quality values we obtained.

11. Conclusions

This research contributes to the field of CL by leveraging the concept of rehearsal buffer as a foundational element for addressing the challenges posed by evolving datasets in DL models. The concept is extended, to make it suitable for data-parallel training, thus enhancing the efficiency and scalability of DL models. We designed and implemented a distributed rehearsal buffer that handles the selection of representative training samples, updates of the local rehearsal buffers, and the preparation of augmented minibatches (sampled from all remote rehearsal buffers using optimized RDMA-enabled techniques) asynchronously in the background. A key aspect is the incorporation of innovative design principles, including asynchronous techniques and the utilization of low-overhead, RDMA-aware, all-to-all communication patterns. Furthermore, our rehearsal buffer was designed to serve heterogeneous data as annotated tuples of tensors, which makes it a reusable component that can be optimized independently and easily integrated with a variety of experience replay strategies. We demonstrated such an integration with a technique leveraging knowledge distillation: Dark Experience Replay.

Practicality is a core focus of this work: the implementation of a distributed rehearsal buffer prototype integrated with PyTorch, a widely adopted AI framework, demonstrates the applicability of these concepts to real-life scenarios. Extensive experiments on up to 128 GPUs using a classification benchmark (ImageNet dataset and ResNet-50 family of DL models) have underlined an improvement of the top-5 evaluation accuracy from 23.1% to 80.55% compared with incremental train-

ing, with just a small runtime increase—an ideal trade-off that combines the best of both baselines used in the comparison. Furthermore, we made the case for Edge-to-HPC streaming applications that benefit from continual learning to train DL models, with the goal of replacing expensive computations to make better use of resources and improve performance. Experiments with a ptychographic image reconstruction application that makes use of generative DL models have shown better image reconstruction sharpness and faster convergence compared with incremental training, especially when paired with Dark Experience Replay. Encouraged by these promising results, in future work we plan to complement our distributed rehearsal buffer with more sophisticated sampling policies, as discussed in Section 5.2.

References

[1] M. Alam, M. Samad, L. Vidyaratne, A. Glandon, K. Iftekharuddin, Survey on deep neural networks in speech and vision systems, Neurocomputing 417 (2020) 302–321.

[2] S. Rasp, M. S. Pritchard, P. Gentine, Deep learning to represent subgrid processes in climate models, Proceedings of the National Academy of Sciences 115 (39) (2018) 9684–9689.

[3] J. Kates-Harbeck, A. Svyatkovskiy, W. Tang, Predicting disruptive instabilities in controlled fusion plasmas through deep learning, Nature 568 (7753) (4 2019).

[4] P. Balaprakash, R. Egele, M. Salim, S. Wild, V. Vishwanath, F. Xia, T. Brettin, R. Stevens, 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] S. Zhang, S. M. H. Bamakan, Q. Qu, S. Li, Learning for personalized medicine: a comprehensive review from a deep learning perspective, IEEE reviews in biomedical engineering 12 (2018) 194–208.

[6] C. Shorten, T. M. Khoshgoftaar, B. Furht, Deep learning applications for COVID-19, Journal of Big Data 8 (1) (2021) 1–54.

[7] E. A. Huerta, A. Khan, E. Davis, C. Bushell, W. D. Gropp, D. S. Katz, V. V. Kindratenko, S. Koric, W. T. C. Kramer, B. McGinty, K. McHenry, A. Saxton, Convergence of artificial intelligence and high performance computing on NSF-supported cyberinfrastructure, Journal of Big Data 7 (1) (2020) 88.

[8] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, ACM Comput. Surv. 52 (4) (2019).

[9] OpenAI, AI and compute, `https://openai.com/research/ai-and-compute` (2023).

[10] M. McCloskey, N. J. Cohen, Catastrophic interference in connectionist networks: The sequential learning problem, in: Psychology of Learning and Motivation, Vol. 24, Elsevier, 1989, pp. 109–165.

[11] S. Thrun, A lifelong learning perspective for mobile robot control, in: Intelligent Robots and Systems, Elsevier, 1995, pp. 201–214.

[12] R. Hadsell, D. Rao, A. A. Rusu, R. Pascanu, Embracing change: Continual learning in deep neural networks, Trends in Cognitive Sciences 24 (12) (2020) 1028–1040.

[13] M. K. Titsias, J. Schwarz, A. G. d. G. Matthews, R. Pascanu, Y. W. Teh, Functional regularisation for continual learning with gaussian processes, arXiv preprint arXiv:1901.11356 (2019).

[14] P. Pan, S. Swaroop, A. Immer, R. Eschenhagen, R. Turner, M. E. E. Khan, Continual deep learning by functional regularisation of memorable past, Advances in Neural Information Processing Systems 33 (2020) 4453–4464.

[15] S. I. Mirzadeh, M. Farajtabar, R. Pascanu, H. Ghasemzadeh, Understanding the role of training regimes in continual learning, Advances in Neural Information Processing Systems 33 (2020) 7308–7320.

[16] P. Buzzega, M. Boschini, A. Porrello, D. Abati, S. Calderara, Dark experience for general continual learning: a strong, simple baseline, Advances in Neural Information Processing Systems 33 (2020) 15920–15930.

[17] A. Kalia, M. Kaminsky, D. G. Andersen, Design guidelines for high performance {RDMA} systems, in: 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016, pp. 437–450.

[18] T. Bouvier, B. Nicolae, H. Chaugier, A. Costan, I. Foster, G. Antoniu, Efficient Data-Parallel Continual Learning with Asynchronous Distributed Rehearsal Buffers, in: CCGrid 2024 - IEEE 24th International Symposium on Cluster, Cloud and Internet Computing, Philadelphia (PA), United States, 2024, pp. 1–10. `doi:10.1109/CCGrid59990.2024.00036`.
URL `https://inria.hal.science/hal-04600107`

[19] M. J. Cherukara, T. Zhou, Y. Nashed, P. Enfedaque, A. Hexemer, R. J. Harder, M. V. Holt, Ai-enabled high-resolution scanning coherent diffraction imaging, Applied Physics Letters 117 (4) (2020).

[20] F. Pfeiffer, X-ray ptychography, Nature Photonics 12 (1) (2018) 9–17.

[21] A. V. Babu, T. Zhou, S. Kandel, T. Bicer, Z. Liu, W. Judge, D. J. Ching, Y. Jiang, S. Veseli, S. Henke, et al., Deep learning at the edge enables real-time streaming ptychographic imaging, Nature Communications 14 (1) (2023) 7059.

[22] M. J. Cherukara, Y. S. Nashed, R. J. Harder, Real-time coherent diffraction inversion using deep generative networks, Scientific reports 8 (1) (2018) 16520.

[23] A. N. Laboratory, Tike: A toolbox for tomographic reconstruction of 3d objects from ptychography data, `https://tike.readthedocs.io` (2023).

[24] C. Y. Hémonnot, J. Reinhardt, O. Saldanha, J. Patommel, R. Graceffa, B. Weinhausen, M. Burghammer, C. G. Schroer, S. Köster, X-rays reveal the internal structure of keratin bundles in whole cells, ACS nano 10 (3) (2016) 3553–3561.

[25] A. Björling, L. A. Marçal, J. Solla-Gullón, J. Wallentin, D. Carbone, F. R. Maia, Three-dimensional coherent bragg imaging of rotating nanoparticles, Physical Review Letters 125 (24) (2020) 246101.

[26] M. V. Holt, S. O. Hruszkewycz, C. E. Murray, J. R. Holt, D. M. Paskiewicz, P. H. Fuoss, Strain imaging of nanoscale semiconductor heterostructures with x-ray bragg projection ptychography, Physical review letters 112 (16) (2014) 165502.

[27] Y. Jiang, Z. Chen, Y. Han, P. Deb, H. Gao, S. Xie, P. Purohit, M. W. Tate, J. Park, S. M. Gruner, et al., Electron ptychography of 2d materials to deep sub-ångström resolution, Nature 559 (7714) (2018) 343–349.

[28] K. Datta, A. Rittenbach, D.-I. Kang, J. P. Walters, S. P. Crago, J. Damoulakis, Computational requirements for real-time ptychographic image reconstruction, Applied Optics 58 (7) (2019) B19–B27.

[29] Z. Liu, H. Sharma, J.-S. Park, P. Kenesei, A. Miceli, J. Almer, R. Kettimuthu, I. Foster, Braggnn: fast x-ray bragg peak analysis using deep learning, IUCrJ 9 (1) (2022) 104–113.

[30] A. V. Babu, T. Bicer, S. Kandel, T. Zhou, D. J. Ching, S. Henke, S. Veseli, R. Chard, A. Miceli, M. J. Cherukara, Ai-assisted automated workflow for real-time x-ray ptychography data analysis via federated resources, arXiv preprint arXiv:2304.04297 (2023).

[31] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, ACM Computing Surveys 52 (4) (2019) 1–43.

[32] G. M. Van de Ven, A. S. Tolias, Three scenarios for continual learning, arXiv preprint:1904.07734 (2019).

[33] M. Mermillod, A. Bugaiska, P. Bonin, The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects, Frontiers in Psychology 4 (2013) 504.

[34] H. Hu, A. Li, D. Calandriello, D. Gorur, One pass imagenet, arXiv preprint arXiv:2111.01956 (2021).

[35] R. Ratcliff, Connectionist models of recognition memory: constraints imposed by learning and forgetting functions., Psychological Review 97 (2) (1990) 285.

[36] A. Robins, Catastrophic forgetting, rehearsal and pseudorehearsal, Connection Science 7 (2) (1995) 123–146.

[37] Y. Balaji, M. Farajtabar, D. Yin, A. Mott, A. Li, The effectiveness of memory replay in large scale continual learning, arXiv preprint arXiv:2010.02418 (2020).

[38] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, Experience replay for continual learning, Advances in Neural Information Processing Systems 32 (2019).

[39] P. Buzzega, M. Boschini, A. Porrello, S. Calderara, Rethinking experience replay: A bag of tricks for continual learning, in: 25th International

Conference on Pattern Recognition (ICPR), IEEE, 2021, pp. 2180–2187.
[40] R. Aljundi, M. Lin, B. Goujaud, Y. Bengio, Gradient based sample selection for online continual learning, Advances in Neural Information Processing Systems 32 (2019).
[41] R. Aljundi, K. Kelchtermans, T. Tuytelaars, Task-free continual learning, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 11254–11263.
[42] A. Chaudhry, A. Gordo, P. K. Dokania, P. Torr, D. Lopez-Paz, Using hindsight to anchor past knowledge in continual learning, arXiv preprint arXiv:2002.08165 3 (2020).
[43] M. Riemer, I. Cases, R. Ajemian, M. Liu, I. Rish, Y. Tu, G. Tesauro, Learning to learn without forgetting by maximizing transfer and minimizing interference, arXiv preprint arXiv:1810.11910 (2018).
[44] M. Boschini, L. Bonicelli, P. Buzzega, A. Porrello, S. Calderara, Class-incremental continual learning into the extended der-verse, IEEE Transactions on Pattern Analysis and Machine Intelligence 45 (5) (2022) 5497–5512.
[45] E. Verwimp, M. De Lange, T. 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.
[46] J. Liu, B. Nicolae, D. Li, J. M. Wozniak, T. Bicer, Z. Liu, I. Foster, Large scale caching and streaming of training data for online deep learning, in: FlexScience'22: The 12th IEEE/ACM Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures, Minneapolis, USA, 2022, pp. 19–26.
[47] NVIDIA, Nvidia data loading library, `https://developer.nvidia.com/DALI` (2023).
[48] A. V. Kumar, M. Sivathanu, Quiver: An informed storage cache for deep learning, in: 18th USENIX Conference on File and Storage Technologies, 2020, pp. 283–296.
[49] S. Nakandala, Y. Zhang, A. Kumar, Cerebro: A data system for optimized deep learning model selection, Proceedings of the VLDB Endowment 13 (12) (2020) 2159–2173.
[50] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, W. Yu, Entropy-aware I/O pipelining for large-scale deep learning on HPC systems, in: IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2018, pp. 145–156.
[51] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, Q. Luo, Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training, in: 49th International Conference on Parallel Processing, 2020, pp. 1–11.
[52] J. Mohan, A. Phanishayee, A. Raniwala, V. Chidambaram, Analyzing and mitigating data stalls in DNN training, arXiv preprint arXiv:2007.06775 (2020).
[53] N. Dryden, R. Böhringer, T. Ben-Nun, T. Hoefler, Clairvoyant prefetching for distributed machine learning I/O, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.
[54] J. Liu, B. Nicolae, D. Li, Lobster: Load balance-aware I/O for distributed DNN training, in: ICPP '22: The 51st International Conference on Parallel Processing, Bordeaux, France, 2022.
[55] D. Munoz, C. Narváez, C. Cobos, M. Mendoza, F. Herrera, Incremental learning model inspired in rehearsal for deep convolutional networks, Knowledge-Based Systems 208 (2020) 106460.
[56] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, C. H. Lampert, icarl: Incremental classifier and representation learning, in: IEEE conference on Computer Vision and Pattern Recognition, 2017, pp. 2001–2010.
[57] A. Chaudhry, M. Rohrbach, M. Elhoseiny, T. Ajanthan, P. K. Dokania, P. H. Torr, M. Ranzato, On tiny episodic memories in continual learning, arXiv preprint arXiv:1902.10486 (2019).
[58] L. Meyer, M. Schouler, R. A. Caulk, A. Ribés, B. Raffin, 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.
[59] R. B. Ross, G. Amvrosiadis, P. H. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, R. W. Robey, D. Robinson, B. W. Settlemyer, G. M. Shipman, S. Snyder, J. Soumagne, Q. Zheng, Mochi: Composing data services for high-performance computing environments, J. Comput. Sci. Technol. 35 (1) (2020) 121–144.
[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, Advances in Neural Information Processing Systems 32 (2019).
[61] A. Sergeev, M. Del Balso, Horovod: Fast and easy distributed deep learning in TensorFlow, arXiv preprint arXiv:1802.05799 (2018).
[62] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL `https://www.tensorflow.org/`
[63] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, Q. Zheng, Mochi: Composing data services for high-performance computing environments, Journal of Computer Science and Technology 35 (2020) 121–144.
[64] T. Bouvier, Distributed rehearsal buffers repository, `https://github.com/thomas-bouvier/distributed-continual-learning` (2023).
[65] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database, in: IEEE Conference on Computer Vision and Pattern Recognition, Ieee, 2009, pp. 248–255.
[66] K. Yang, J. H. Yau, L. Fei-Fei, J. Deng, O. Russakovsky, A study of face obfuscation in ImageNet, in: International Conference on Machine Learning, PMLR, 2022, pp. 25313–25330.
[67] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[68] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch sgd: Training imagenet in 1 hour, arXiv preprint arXiv:1706.02677 (2017).
[69] K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, C. Xu, Ghostnet: More features from cheap operations, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 1580–1589.
[70] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H. Wu, Mixed precision training, in: International Conference on Learning Representations, 2018.
[71] T. Akiba, S. Suzuki, K. Fukuda, Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes, arXiv preprint arXiv:1711.04325 (2017).
[72] L. Bottou, F. E. Curtis, J. Nocedal, Optimization methods for large-scale machine learning, SIAM review 60 (2) (2018) 223–311.
[73] L. 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.
[74] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE transactions on image processing 13 (4) (2004) 600–612.
[75] J. Wang, W. Zhou, J. Tang, Z. Fu, Q. Tian, H. Li, Unregularized auto-encoder with generative adversarial networks for image generation, in: Proceedings of the 26th ACM international conference on Multimedia, 2018, pp. 709–717.