Towards Low-Overhead Resilience for Data Parallel Deep Learning

Bogdan Nicolae[†], Tanner Hobson[*], Orcun Yildiz[†], Tom Peterka[†], Dmitry Morozov[‡],
[*]*University of Knoxville, Tennessee, USA*
Email: thobson2@vols.utk.edu
[†]*Argonne National Laboratory, USA*
Email: {bnicolae,oyildiz,tpeterka}@anl.gov
[‡]*Lawrence Berkeley National Laboratory, USA*
Email: dmorozov@lbl.gov

***Abstract*—Data parallel techniques have been widely adopted both in academia and industry as a tool to enable scalable training of deep learning models. At scale, DL training jobs can fail due to software or hardware bugs, may need to be preempted or terminated due to unexpected events, or may perform suboptimally because they were misconfigured. Under such circumstances, there is a need to recover and/or reconfigure data-parallel DL training jobs on-the-fly, while minimizing the impact on the accuracy of the DNN model and the runtime overhead. In this regard, state-of-art techniques adopted by the HPC community mostly rely on checkpoint-restart, which inevitably leads to loss of progress, thus increasing the runtime overhead. In this paper we explore alternative techniques that exploit the properties of modern deep learning frameworks (overlapping of gradient averaging and weight updates with local gradient computations through pipeline parallelism) to reduce the overhead of resilience/elasticity. To this end we introduce a failure simulation framework and two resilience strategies (immediate mini-batch rollback and lossy forward recovery), which we study compared with checkpoint-restart approaches in a variety of settings in order to understand the trade-offs between the accuracy loss of the DNN model and the runtime overhead.**

***Index Terms*—deep learning; data-parallel training; failure simulation; performance model; trade-off analysis**

I. Introduction

Deep learning (DL) applications are rapidly gaining traction both in industry and scientific computing thanks to the accumulation of massive data sizes. In the area of HPC, scientific instruments collect data in the order of GB/s, accumulating 100+ TB/day, all of which present a wide range of learning opportunities. Unsurprisingly, this creates significant interest to adopt DL on HPC machines for various scientific areas: fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, cancer research, etc.

Such massive data sizes make the training of deep neural network (DNN) models challenging, especially since this involves a large number of epochs, where each epoch is a complete pass over the dataset. The computations performed during each epoch are non-trivial: they involve iterating over the dataset in a random order in small batches (mini-batches), each of which is composed of a forward pass, responsible to predict the result using the model, and a back-propagation, responsible to update the parameters of the model such that the difference between the predicted result and the actual result is minimized (typically achieved using techniques such as stochastic gradient descent).

In a quest to alleviate this challenge, various parallelization techniques have been proposed that leverage multiple compute nodes. A popular technique is *synchronous data-parallel* [1] training. It creates replicas of the DNN model on multiple workers, each of which is placed on a different device and/or compute node. The input data is randomly shuffled and partitioned among the workers at each epoch. During the forward pass, the workers simply proceed in an embarrassingly parallel fashion by iterating over the partition of their dataset. Then, during back-propagation, the weights are not updated based on the local gradients as it is normally done for a serial training, but by using a global average computed across all workers using all-reduce operations. This effectively results in all workers learning the same pattern, to which each individual worker has contributed. Note that these operations need not be serialized. Instead, this can be done in a pipeline: as soon as the computations of the local gradients of a layer are finished, the computations for the previous layer can start, while all-reduce operations to average the local gradients and update the model weights can be performed in parallel. Such an optimization is called *pipeline parallelism*. The combination of data parallelism and pipeline parallelism is widely used in practice, which is why we choose it as the focus of our work.

Despite such optimizations, DL training remains a time-consuming process. Since there are a large number of compute nodes involved, failures due to software or hardware errors are a common occurrence, with mean time between failures (MTBF) of 4-22 hours often reported [2] for HPC machines. As data centers are increasingly adapted for running DL applications (increasing number of GPUs per compute node, increasing complexity of the software stack), the failure frequency is projected to increase. Indeed, a study of large-scale DNN training clusters at Microsoft [3] has found the MTBF to be as low as 45 minutes.

Furthermore, workflows that run multiple DL training in-

stances and/or integrate them with other tasks are becoming increasingly complex, causing *unexpected events* that mimic the impact of failures. For example, if a high-priority, on-demand task needs to be started immediately, then some of the workers of the data-parallel training may need to be killed in a timely fashion, leaving very narrow room to react [4]. Another example is elastic scheduling, which involves the need to redistribute the resources among the tasks. In this case, workers need to be frequently terminated (or suspended) and respawned (or resumed) later. In these scenarios, the frequency of unexpected events can be much higher than the MTBF (order of seconds).

To address failures and unexpected events, tightly coupled applications such as HPC simulations typically rely on checkpoint-restart approaches for resilience: they capture a globally consistent execution state of all processes at regular intervals (checkpoint) and roll back to the most recent one in case of an unexpected event (restart). Data-parallel DL training can also take advantage of checkpoint-restart to replay the mini-batches since the last checkpoint (by making stochastic operations deterministic through the use of pseudo-random number generators with fixed seeds). While this solves the problem of obtaining identical results compared with a failure-free run, it suffers from high performance overheads due to loss of progress, since DNN models are typically checkpointed at the end of each epoch or less frequently.

In this paper, we focus on alternative resilience strategies that relax the requirement to obtain identical results compared with a failure-free run in order reduce the performance overhead. To this end, we leverage a key observation: DL training is a stochastic process, therefore it is possible to relax several consistency considerations, in particular during the back-propagation. For example, data-parallelism may still produce a viable model even if only a subset of the whole process group contributes to the gradient averages. Furthermore, a DNN model may still be viable even if the weights of its layers were only partially updated during a mini-batch. Thus, an interesting question arises: can we sacrifice consistency to avoid a rollback to a previous checkpoint in case of unexpected events (and thus avoid loss of progress) without significant impact on the accuracy of the DNN model? Our contributions aim to answer this question. We summarize them below:

- We propose two resilience strategies, both of which are based on the idea of capturing the state of the DNN model on the surviving workers of the data-parallel training *after* the unexpected event happened and without access to any previous checkpoint. Using this (potentially inconsistent) state, we either: (1) launch additional workers to replace the failed ones, broadcast the state to them and then globally replay the last mini-batch; (2) continue the back-propagation by averaging the gradients of the surviving workers (Section III-B).
- We design an develop a failure simulation framework specifically designed for the combination of data-parallel and pipeline parallel DL training using the *Tensorflow* and *Horovod* DL runtimes. Our approach enables fine-grain control over the moment when to simulate a failure and captures the state of the DNN model at that moment. This allows a study of the proposed resilience strategies in a variety of simulated failure scenarios (Section III-C).
- We run a series of extensive experiments on an HPC testbed using two DL applications in order to study the trade-offs between performance and accuracy for the proposed strategies in a variety of simulated failure scenarios. The results show our proposed resilience strategies incur a significantly lower performance overhead, at the expense of negligible accuracy loss (Section IV).

II. Related Work

Checkpoint-Restart: is a resilience strategy applied both for loosely coupled and tightly coupled applications. When I/O bandwidth is a concern (especially for tightly-coupled HPC applications running at large scale), *multi-level checkpointing* [5], [6] can be used to leverage complementary strategies (partner replication, erasure coding) adapted for HPC storage hierarchies. VELOC [7], [8] takes this approach further by introducing asynchronous techniques to apply such complementary strategies in the background. When the checkpoints of different processes have similar content, techniques such as [9], [10] can be applied to complement multi-level checkpointing. However, for data-parallel training, the DNN models are known to be identical in advance, therefore such techniques introduce unnecessary overheads.

DNN Model Checkpointing: popular runtimes such as Tensorflow and PyTorch implement rudimentary checkpointing into either standardized formats (e.g. HDF5) or custom formats (e.g., SaveModel[1]). However, such techniques are not optimized to take advantage of multiple storage tiers, asynchronous I/O, nor to leverage the redundancy of identical DNN model replicas. To this end, checkpointing approaches such as DeepFreeze [11] introduce sharing techniques that write different parts of the DNN model in parallel from each replica, which can be overlapped with the back-propagation by embedding the write operations directly into the execution graph. Building on top of such techniques is the related problem of cloning, which involves checkpointing an ongoing data-parallel training while simultaneously starting a separate data-parallel training from the checkpoint [12]. Other approaches such as CheckFreq [13] focus on determining the optimal checkpointing frequency through systematic online profiling of the overhead. Such approaches can be used to reduce the overhead of capturing the DNN model state after an unexpected event has occured.

Silent errors in DNNs: are explored in a variety of scenarios. For example, FT-CNN [14] uses algorithmic-based fault tolerance techniques that are capable of protecting DNN models against non-catastrophic soft errors by using lightweight checksums to minimize both the compute and data redundancy. Li at al. [15] study silent error propagation in DNNs, concluding that their impact depends mostly on the data structures

[1]https://www.tensorflow.org/guide/saved_model

being used, frequency of reuse and sensitive bits. They propose mitigation techniques such as symptom-based error detectors and selective latch hardening. Similar studies focus on silent errors due to running DNNs on accelerators using reduced voltage to limit energy consumption [16]. Silent errors were also investigated in trade-offs, e.g. to reduce the overhead of all-reduce gradient averaging in data-parallel training by ignoring stragglers [17].

To summarize, state-of-art techniques either focus on checkpoint-restart or study specific scenarios such as silent errors. By contrast, our work focuses on fail-stop errors and unexpected events that kill processes but without introducing the overhead of checkpointing during failure-free execution. To our best knowledge, *we are the first to explore this aspect*.

III. Failure Analysis Framework

This section details our approach, starting from the principles behind the Tensorflow and Horovod runtimes that enable the combination of data parallelism with pipeline parallelism, which we exploit to design and develop two resilience strategies, a failure simulator and a performance model.

A. Background

DL runtimes like Tensorflow take advantage of multi-core and hybrid architectures to parallelize computations at fine granularity. To this end, they enforce a programming model based on data-flow graphs: applications describe a computation by defining a graph whose nodes represent the operators and whose edges represent the operands (which take the form of tensors, or multi-dimensional arrays). Then, based on this data-flow graph, Tensorflow optimizes the schedule of where to run the operators and how to transfer the inputs and outputs between them.

However, it is non-trivial for application developers to reason at low level directly in terms of data-flow graphs. Therefore, Tensorflow provides high-level abstractions built on top of data-flow graphs that are exposed in programming languages such as Python: *data pipelines* that allow asynchronous fetching and pre-processing of the training samples, building blocks of DL models (e.g., certain arrangements of layers) or even full standardized DL models, optimizers that implement the forward pass and back-propagation using various techniques (SGD, Adam, etc.). These abstractions are then combined and translated automatically into a data-flow graph. At this point, pipeline parallelism is implemented during the back-propagation by simply reusing the output produced by the sub-graph responsible for the gradient calculations of a given layer for two other parallel sub-graphs: (1) the gradient calculations of the previous layer using the chain rule; (2) the weight updates of the given layer.

Since much of the DL ecosystem has evolved around Python, complementary runtimes that enable data parallelism such as Horovod continue this tradition. Specifically, Horovod hides the details of parallelization by wrapping around the optimizer instantiated by the DL application in order to transparently augment the data-flow graph to average the local gradients of all workers using an all-reduce collective communication pattern (e.g., as provided by MPI implementations) before proceeding with the weight updates. Using this approach has two advantages: (1) at application level, there are only minimal changes necessary take advantage of data parallelism; (2) all-reduce collective communication is overlapped with the rest of the operations performed by the data-flow graph, thereby reducing communication overheads.

B. Resilience Strategies

We propose two resilience strategies that take advantage of the low-level data-flow graph to capture the DNN model on the surviving workers *after* an unexpected event happened and continue the DNN training *without* loss of progress.

Immediate mini-batch rollback: is based on the idea replaying the same mini-batch. The key observation we leverage is that even if a single DNN model replica survived out of the entire group participating in the data-parallel training, it is enough to capture all information needed to continue. If the failure happened during the forward pass, then the DNN model is in a consistent state. If the failure appended during the back-propagation, then the weights of the DNN model may have been updated only partially, therefore it may be in an inconsistent state. However, it is important to note that the partial updates are not random: even when considering fine-grain parallelism, the updates in the data-flow graph are likely to be applied in reverse order of the layers. Therefore, up to the point where the last update was applied, the DL model will use the same initial weights, while from that point on, it will funnel the results in a slightly different but coherent direction. Thus, there is a high chance that this kind of inconsistencies do not break the viability of the DNN model. We can leverage this observation to simply checkpoint the DNN model on the surviving workers, launch a new set of workers (or to use spare workers that are waiting in stand-by) and broadcast the DNN model to them (using simple techniques such as electing a leader among the survivors to perform the broadcast or more advanced techniques such as cloning [12]). Once all workers hold synchronized DNN model replicas, the failed mini-batch can be replayed. Note that each new worker needs to reload the original data partition and assemble a new mini-batch. However, this process can be easily made deterministic by using a pseudo-random number generator with a fixed seed (e.g., all workers agree on a random seed at the beginning of the training, to which they add their id).

Lossy forward recovery: is based on the idea of continuing the training only with the surviving workers. Regardless whether the failure happened during the forward pass or the back-propagation, ultimately each surviving worker independently updates its own DNN model replica, therefore it will maintain consistency if allowed to complete the mini-batch. However, the question is, what gradients should the surviving workers use to perform the weight updates? If each worker uses its own local gradients, then the influence of important training samples belonging to other partitions may be lost. If the survivors pay an additional overhead to average the

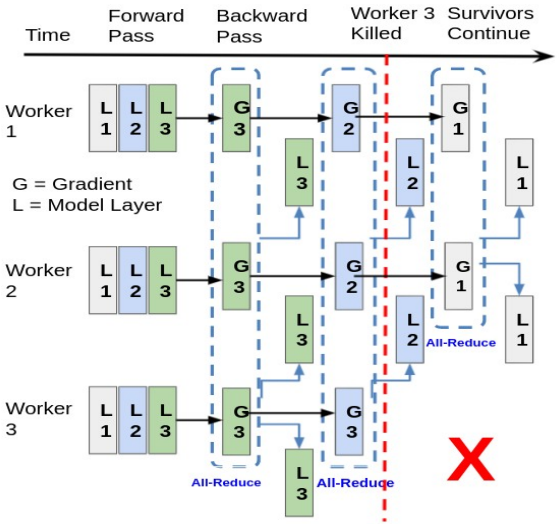


Fig. 1: Lossy forward recovery: Illustration for three workers, each of which is updating its DNN model replica during back-propagation using pipeline parallelism. After worker 3 fails, the survivors continue the back-propagation to complete the mini-batch by averaging the gradients among themselves.

gradients among themselves, then they will lose only the influence of the training samples belonging to the mini-batches of the failed workers. We advocate for the latter, since the overhead of restoring communication among the survivors (e.g., by creating a new MPI communicator) is typically small. An example of how this works is illustrated in Figure 1.

Note that this strategy is also particularly well suited to support unexpected events that do not necessarily terminate the workers, as detailed in Section I for scenarios such as task pre-emption and elasticity. In this case, workers can be put in a dormant state where they do not perform any computations and simply contribute with token values that are ignored to the gradient all-reduce. Using this approach, the workers can be suspended and resumed in near real-time, without the need to reinitialize or reconstruct MPI communicators.

C. Failure/Unexpected Event Simulation Framework

To study the impact of failures and unexpected events on the accuracy and performance overhead of both resilience strategies discussed in Section III-B, we propose the design and implementation of a simulation framework that is capable of generating controlled scenarios. Note that the general principles discussed in this section can be easily extended to study other resilience strategies as well.

We set three design goals: (1) ability to control the moment when a simulated failure happens; (2) generate the same outcomes as if a real failure happened; (3) minimize the modifications to the original code in order to accurately measure the performance overhead. To this end, we introduce the following components:

Failure injection using a custom optimizer: we propose a custom optimizer that extends the standard Keras optimizers to allow the user to specify a DNN model layer beginning with which a failure is simulated during the back-propagation on a specified number of model replicas (called *casualties*). We refer to the model replicas on which no failures are being simulated as *survivors*. Our assumption is that the weights of all successive layers (in reverse order beginning with the last) have been successfully updated by the time the data-flow graph reaches the specified layer on both the casualties and the survivors. Then, instead of aborting the training immediately, we modify the data-flow graph of the casualties to set the content of the local gradients for the specified layer and all preceding layers with zero and run the mini-batch to completion. Using this approach, we satisfy goal (1).

Immediate mini-batch rollback simulation: In the case of immediate mini-batch rollback, the number of survivors is zero, which means the average gradients across all workers beginning with the specified layer is zero. Therefore, from that moment on, the weights of the model replicas will not be updated, which means the outcome is equivalent to stopping all workers. This enables us to reach goal (2). To simulate immediate mini-batch rollback, it is sufficient to let the failed mini-batch run to completion, checkpoint the model on one of the workers, restart all workers from the checkpoint, then re-run the same mini-batch in failure-free mode. Using this approach, the whole process is greatly simplified. Furthermore, it is important to note that the extra runtime overhead for running the failed mini-batch to completion is negligible (because we directly set the tensors to zero in the data-flow graph, therefore no computation is performed to calculate any gradients beginning with the specified layer). This enables an accurate approximation of the overall runtime overhead, thus satisfying goal (3).

Lossy forward recovery simulation: In the case of lossy forward recovery, there must be a non-zero number of survivors. One approach to address this case is to simply modify the average of the gradients starting from a specified layer to perform the sum as usual over all workers, but divide the result over the number of survivors, which effectively ignores the zero-filled gradients of the casualties. This is equivalent to averaging the gradients of the survivors directly, satisfying goal (2). However, such an approach requires modifications to the Horovod framework in order to compute a different average function. Instead, we adopt a simpler approach: starting from the specified layer, every worker divides its local gradients by the number of survivors instead of the number of total workers (except for the casualties, for which the gradients remain zero). Then, instead of reducing the gradients using an average function as usual, we reduce them using the sum function, which is available as an alternative in Horovod. Using this approach, we satisfy both goals (2) and (3).

D. Implementation Details

We implemented the resilience strategies and simulation framework introduced above for the Keras library shipped with *Tensorflow* 2.5.0. For the purpose of this work, we have chosen to use the tensorflow.keras.optimizers.SGD optimizer as a base class, since it the most widely used in practice. However, it is

important to note that our approach can be trivially adapted to any other optimizer by using a different base class. Specifically, we intercept the _compute_gradients method responsible to generate the data-flow graph corresponding to the gradient tape: after executing the code of the super-class, starting with a given gradient (expressed as percent out of the total number of gradients) we fill the content of the tensors with zero on the casualties. Then, we return the modified data flow graph, which is used by the optimizer internally to obtain the actual gradients and update the weights of the model. This approach integrates seamlessly with Horovod: the distributed Horovod optimizer that normally wraps around the original Keras optimizers, simply needs to wrap around our own optimizer instead. In addition, we implemented a callback for model.fit that is responsible to perform checkpointing (using the default HDF5 format) at a given epoch and mini-batch number, as well as to monitor and to log relevant metrics, such as duration of each mini-batch, loss and accuracy.

IV. Experimental Evaluation

This section describes the experimental evaluation we performed in order to study the trade-off between the performance and accuracy achieved by the robustness strategies proposed in Section III-B in several controlled scenarios that simulate failures and/or unexpected events.

A. Experimental Setup

Our experiments were performed on Argonne's *ThetaGPU* cluster, a testbed specifically optimized for training DNN models at scale. It comprises 24 NVIDIA DGX A100 nodes, each with eight NVIDIA A100 Tensor Core GPUs and two AMD Rome CPUs. Memory-wise, each node is equipped with 1 TB of DDR4 memory and 320 GB GPU memory, for a total of 24 TB DDR4 and 7.6 TB GPU memory. The nodes are interconnected using 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. External storage is provided by a Lustre parallel file system, which is mounted using POSIX and provides an aggregated I/O bandwidth of 250 GB/s.

For the purpose of this work, we use one full ThetaGPU node to run a data-parallel training on its eight GPUs. Each Horovod worker is responsible for a DNN model replica and runs on its own dedicated GPU.

In terms of deep learning software, we use *Horovod* v0.20.3 and *Tensorflow* v2.5, which comes with its own optimized implementation of the *Keras* library. They are running on top of Python v.3.8.5 and CUDA v11.1.105.

B. Applications

We choose two representative real-life DL applications for our study. They exhibit complementary DNN model properties (small size with large number of layers vs. large size with small number of layers), which results in different behavior patterns with respect to loss of accuracy and recovery overheads.

ResNet-50: ResNet is a family of DNN where the layers learn residual functions with reference to the input layers, instead of learning unreferenced functions. This allows ResNet to train extremely deep neural networks with 150+ layers, which was difficult prior to its introduction due to the problem of vanishing gradients [18]. Thanks to this breakthrough, ResNet became a highly popular image classification benchmark. In this paper, we study the ResNet-50 variant, which is the most popular in the family. We rely on the default implementation shipped with *Keras*. As training data, we use the TinyImageNet dataset [19], which is $\approx$ 200 MB large and includes 100,000 samples. The training data is partitioned among all workers. Each worker randomly samples its partition to obtain a different mini-batch compared with the other workers. This DNN model is representative of a large number of layers, each of which is small (in the order of MiB).

CANDLE-NT3: CANDLE [20] (Cancer Distributed Learning Environment) is a project that aims to combine the power of Exascale computing with deep learning to address a series of loosely connected problems in cancer research. In this context, we study on NT3 [21], which consists of a 1D convolutional network for classifying tissue, expressed as gene sequences, as normal or tumorous. This type of network follows the classic architecture of convolutional models with multiple 1D convolutional layers interleaved with pooling layers followed by final dense layers. The optimizer used by NT3 is SGD (stochastic gradient descent). The training data size for this benchmark is $\approx$ 600 MB, which includes 1120 training samples. NT3 is a representative DNN model of a small number of layers, each of which can grow to huge sizes (hundreds of MiB).

C. Methodology

We compare the resilience strategies described in Section III-B with a baseline strategy that checkpoints every epoch and restarts from the latest checkpoint in case of failures or unexpected events. For brevity, we use the following notations: Immediate–Rollback refers to the immediate mini-batch rollback, Lossy–Forward refers to the lossy forward recovery, and Checkpoint–Restart refers to the baseline strategy.

Our goal is to study the trade-off between accuracy loss and performance overhead for all three strategies. Since it is not feasible to simulate all possible points of failures or unexpected events, we focus our study on a subset of representative epochs and mini-batches that capture the behavior of the training at different moments during the runtime. We to these representative mini-batches as *scenarios*. As mentioned in Section III-C, failures during the forward pass are trivial to handle, because the model remains read-only. Therefore, we study only on the back-propagation in each of these scenarios.

In terms of accuracy, recall that Checkpoint–Restart involves a replay of all mini-batches since the beginning of the epoch starting from a consistent checkpoint, resulting in the same accuracy as an equivalent failure-free run. On the other hand, for Immediate–Rollback and Lossy–Forward, the accuracy is not guaranteed to remain identical to the equivalent

failure-free run, which is why we measure its deviation from the failure-free run.

In terms of performance, Checkpoint–Restart incurs three overheads: (1) *replay overhead*, which measures the time necessary to roll back to the last checkpoint and replay all lost mini-batches until the moment of failure or unexpected event; (2) *checkpointing overhead*, which measures the increase in runtime during failure-free execution due to periodic checkpointing; (3) *restart overhead*, which measures the various overheads involved in restarting the training (initializing the model replicas, communicators, loading the model weights and optimizer state, etc.). In the case of Immediate–Rollback, we also have a restart overhead followed by a replay overhead, but involves a single mini-batch. There is no periodic checkpointing overhead. In the case of Lossy–Forward, if the survivors continue without spawning more workers, no additional overheads are present. Otherwise, Lossy–Forward incurs a restart overhead as well, albeit only partially for the new workers that join the survivors.

Only the replay overhead of Checkpoint–Restart depends on the scenario. Therefore, we report all other overheads separately. They can be added to the scenario-specific replay overhead to obtain an end-to-end overhead.

Note that DL training is by nature a stochastic process, because it relies on random sampling. Thus, to guarantee a fair comparison between the three strategies, we force it to become a deterministic process by using pseudo-random number generators with fixed seeds. Specifically, we fix the seeds of three different generators that are used both directly and indirectly at various levels: TensorFlow, NumPy, and Python's standard library. Using this approach, we guarantee the selection of the same training samples for each mini-batch, which results in identical gradients and therefore identical weight updates across different runs.

D. Scenarios

Our first set of experiments focuses on the selection of the scenarios for both ResNet–50 and CANDLE–NT3. To this end, we run both applications failure-free on eight GPUs (single ThetaGPU node) until the DNN models converge. We record the loss and accuracy after each mini-batch and depict the results in Figure 2a and Figure 2b respectively.

As expected, the accuracy increases faster in the beginning and gradually flattens with an increasing number of epochs. In the case of ResNet–50, we can observe four distinct phases: a sharp increase in the accuracy for the first 8 epochs, followed by a gradual decline starting from epoch 8, 20, 30. Therefore, we pick four representative scenarios: mini-batch 200 (representing the beginning of the training during epoch 0), mini-batch 4000 (representing the second phase during epoch 10), mini-batch 10000 (representing the third phase during epoch 25) and mini-batch 25000 (representing the fourth phase during epoch 64). Similarly, in the case of CANDLE–NT3 we observe three phases: a sharp increase in accuracy for the first two epochs, a gradual increase until epoch 5, and then convergence after epoch 5. Therefore we pick three representative scenarios: mini-batch 10 (representing the beginning of the training during epoch 0), mini-batch 250 (representing the second phase during epoch 4), mini-batch 500 (representing the third phase during epoch 8).

TABLE I: Model properties and overheads for both ResNet–50 and CANDLE–NT3 (replicated on 8 GPUs for data-parallel training)

Property	ResNet-50	CANDLE-NT3
Model size	600 MiB	91 MiB
Layers	10	50
Mini-batch size	32	20
Mini-batches/epoch	390	55
No epochs	90	10
Avg. mini-batch duration	0.17 s	2.2 s
Init overhead	3.0 s	1.9 s
Checkpoint overhead	0.48 s	3.4 s
Restart overhead	4.09 s	2.5 s

Next, we summarize the properties and scenario-independent overheads of the DNN models in Table I. As can be observed, the two models exhibit differences in the corresponding overheads: initialization and restart overhead is larger for ResNet–50, while checkpointing overhead and average mini-batch duration is significantly larger for CANDLE–NT3. This can be explained by the fact that ResNet–50 has a larger number of layers, which results in the need to construct a more complex execution graph. On the other hand, the CANDLE–NT3 model has a larger size, which results in a longer serialization and longer I/O operations (that make up the majority of the checkpointing overhead).

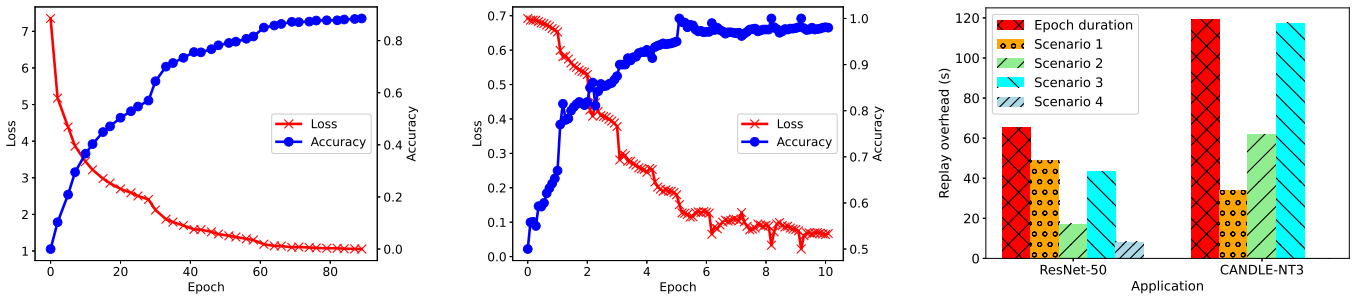
Note that the duration of each mini-batch is close to the average. Therefore, we did not explicitly depict the evolution of the runtime for an increasing number of epochs. This can be easily deducted using Table I.

E. Checkpoint-Restart

The checkpointing and restart overheads mentioned in Section IV-E were measured using the default model loading/saving primitives available in Tensorflow. While they can be further reduced using state-of-art checkpointing approaches (see Section II), even in the ideal case when checkpointing and restart overheads are negligible, Checkpoint–Restart will incur a significant replay overhead. Thus, we focus our study on this aspect.

We assume a typical setup in which the DNN model is checkpointed at the end of each epoch. In this case, Figure 2c depicts the replay overhead caused by restarting from the latest checkpoint, which was taken at the beginning of the previous epoch. For comparison, the total epoch duration is also included.

As expected, the result depends on how many mini-batches were processed since the beginning of the epoch. In fact, given that the duration of the mini-batches is close to the average, the replay overhead is easy to predict by multiplying the number of mini-batches since the beginning of the epoch with the average mini-batch duration taken from Table I. Figure 2c confirms this prediction.



(a) ResNet-50: Accuracy (higher is better) and loss (lower is better) for an increasing number of epochs.

(b) CANDLE-NT3: Accuracy (higher is better) and loss (lower is better) for an increasing number of epochs.

(c) Replay overhead caused by restarting from checkpoints taken at epoch boundary relative to epoch duration.

Fig. 2: Evolution of failure-free data-parallel training of ResNet-50 and CANDLE-NT3 on eight GPUs vs. replay overhead caused by Checkpoint–Restart.

In general, each failure will cause on the average a replay overhead that equals to half of the epoch duration. While this may not seem significant for the considered examples (for which the loss of progress is in the order of minutes), it is also important to remember that DNN training can scale to a much larger number of GPUs and may consume massive amounts of input data, which means each epoch consists of a much larger number of mini-batches, each of which will take longer to complete. Therefore, the average loss of progress is expected to be greatly amplified at scale even for a *single* failure or unexpected event. At the same time, the frequency of failures is constantly increasing, while the frequency of unexpected events can be significantly higher than the mean time between failures due to elastic scheduling, as mentioned in Section I. Thus, an already amplified loss of progress is expected to happen more frequently, leading to much high performance overheads.

Furthermore, the additional overheads mentioned in Section IV-C need further consideration. In particular, the initialization and restart overhead is triggered for each failure/elasticity event, while the checkpointing overhead is triggered for each epoch. When accounting for all these factors, it becomes evident that a traditional checkpoint-restart strategy can quickly become a performance bottleneck, which is a high price to pay for reproducing the same result as a failure-free execution.

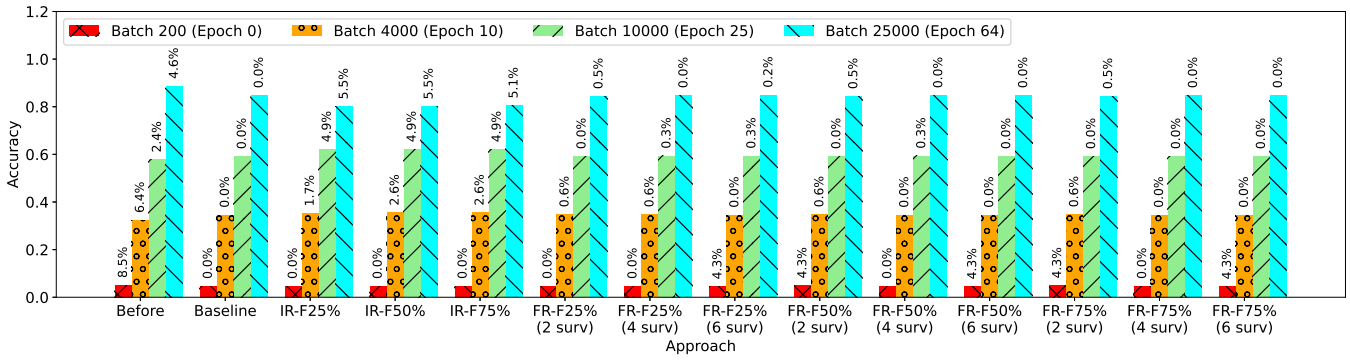
F. Resilience Strategies

Our next experiments evaluate the Immediate–Rollback and the Lazy–Forward strategies. To simulate failure, first we capture a point of reference (denoted Before) by checkpointing the DNN model right before the beginning of the representative epoch and mini-batch corresponding to each scenario. Next, we start from the same Before state for both proposed strategies in all scenarios and proceed with the forward pass of the mini-batch, which is allowed to run to completion. During the back-propagation, we simulate failures at three different rates of progress, i.e., after 25%, 50% and 75% of the gradients were computed and applied to update the weights. After each strategy finished processing the mini-batch, we measure the final accuracy and the performance overheads (averaged across all workers). In the case of Lazy–Forward, we vary the number of survivors as well, starting from two up to six (out of the total of eight workers). For simplicity, we assume the case when the survivors continue without using spares to replace the casualties. In this case, Lazy–Forward has no performance overhead.

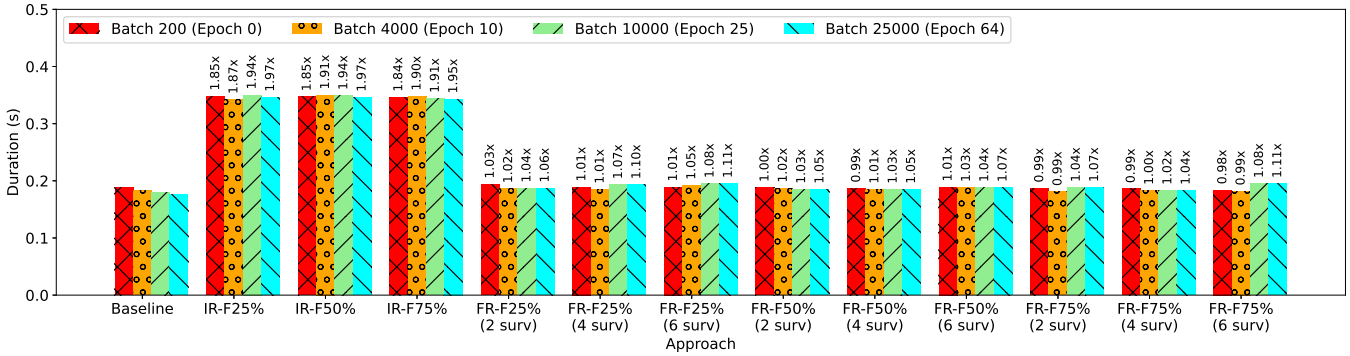
First, we focus on the accuracy results for the ResNet–50 application, which are depicted in Figure 3a. As can be observed, the impact of each mini-batch on the accuracy is significant, because there is a substantial difference (ranging between 2.6% - 8.5%) between the Before state and the state after a failure-free mini-batch processing (denoted Baseline). As expected, this difference is the largest in the beginning of the training.

In the case of Immediate–Rollback, replaying the mini-batch from an inconsistent checkpoint deviates from the accuracy of Baseline by up to 5.5%, which is within the same limits as the difference between Before and Baseline. However, in this case, the difference is growing as the training progresses, which is the opposite of the trend observed for Before. Also interesting to observe is that the rate of progress has limited impact on the accuracy. Specifically, for Scenario 3 (mini-batch 10000, epoch 25), it does not matter at what point during the back-propagation we encounter a failure. Furthermore, for Scenario 2 (epoch 10, mini-batch 4000), an early failure actually reduces the difference from Baseline. However, for Scenario 4 (epoch 64, mini-batch 25000), the opposite is true. There are two complementary factors involved that explain this observation. On one hand, an early failure causes more inconsistencies in the model state (because the error was propagated to fewer layers during the back-propagation). However, on the other hand, the replay of the mini-batch introduces a bias that gets amplified when the failure happens later. Both factors amplify the difference from Baseline.

In the case of Lossy–Forward, it is interesting to observe that the difference from Baseline is much lower than in the



(a) ResNet–50: Accuracy after processing the mini-batch corresponding to the scenario (higher is better). The relative difference to the baseline is indicated in percent on top of each bar.



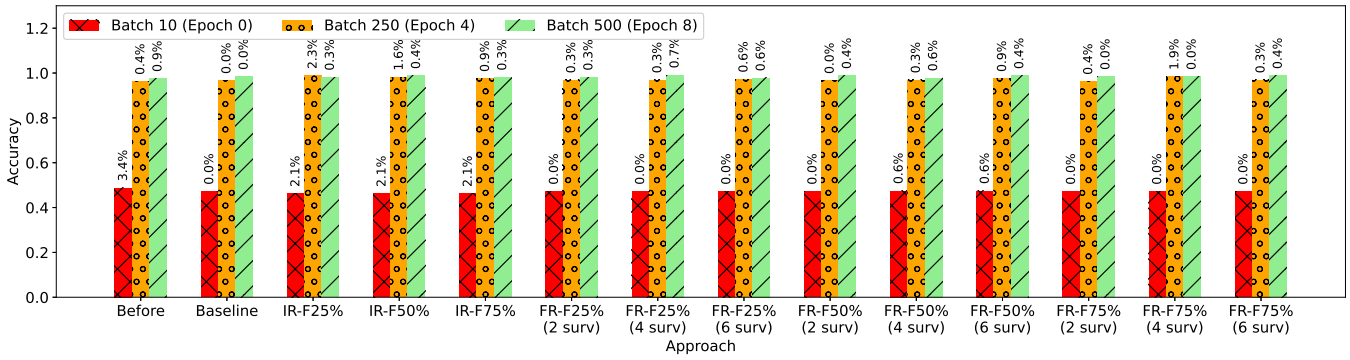
(b) ResNet–50: Performance overhead after processing the mini-batch corresponding to the scenario (lower is better). The slowdown with respect to the baseline (approach duration divided by baseline duration) is indicated on top of each bar.

Fig. 3: ResNet50: Accuracy vs. Performance Overhead for the Immediate–Rollback (denoted as IR) and Lossy–Forward strategies (denoted FR) for data-parallel training on 8 GPUs. Before represents the state at the beginning of the mini-batch. Baseline represents the state after the failure-free run on the mini-batch. The rate of progress during the back-propagation after which a failure is simulated is denoted as F–XX%. In the case of Lossy–Forward, the number of survivors is mentioned explicitly.

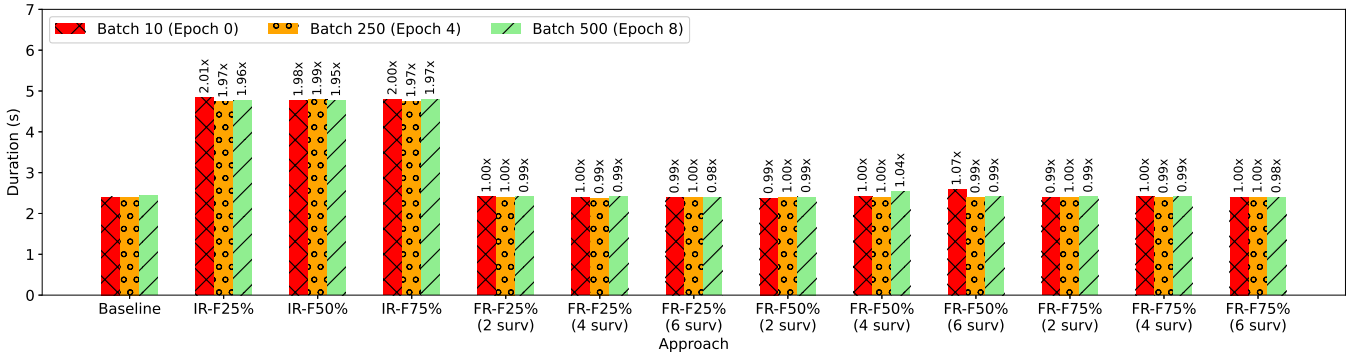
case of Immediate–Rollback for all configurations. This is a surprising finding, especially when the number of survivors is small and the failures happen early, because only the survivors are averaging the remaining gradients (thereby ignoring a large number of gradients from the casualties, which effectively gives them a large bias). Furthermore, we can observe another interesting trend: both the rate of progress and the number of survivors have a non-negligible impact on the accuracy of Lossy–Forward. This is especially visible for the case of four survivors, which consistently perform better than two survivors and show a decreasing difference from Baseline for an increasing progress rate (up to zero difference at 75%).

Next, we study the performance overheads for the ResNet–50 application, which are depicted in Figure 3b. As expected, in the case of Lossy–Forward, the performance overhead is close to Baseline, because the survivors continue the mini-batch undisturbed by the casualties. In fact, there is a slight overhead, which can be traced back to the collective operations used in our simulations to average the gradients, during which the casualties still contribute with zero-filled tensors. Nevertheless, this overhead is negligible compared with the initialization overhead required to reconfigure the Horovod communicators. Thus, a scenario that requires elastic downsizing would benefit from keeping trivial workers that keep sending zero-filled tensors at least until the completion of the mini-batch. In the case of Immediate–Rollback, the slowdown is almost 2x (but consistently lower than that), which is expected given that the mini-batch needs to be replayed.

Next, we focus on the accuracy results for the CANDLE–NT3 application, which are depicted in Figure 4a. The results generally follow a similar trend compared with the ResNet–50 application, but with several interesting differences we discuss below. First, the Immediate–Rollback strategy exhibits a larger difference in accuracy compared with Baseline in the beginning of the training (epoch 0, mini-batch 10) than for the later scenarios. This is a reversal from ResNet–50 and can be explained by the fact that CANDLE–NT3 sees a sharp increase in the accuracy in the beginning, therefore biases introduced by replaying the mini-batch are amplified. Second, in the case of Lossy–Forward, there is no bias introduced by the survivors in the beginning of the training. On the other



(a) CANDLE–NT3: Accuracy after processing the mini-batch corresponding to the scenario (higher is better). The relative difference to the baseline is indicated in percent on top of each bar.



(b) CANDLE–NT3: Performance overhead after processing the mini-batch corresponding to the scenario (lower is better). The slowdown with respect to the baseline (approach duration divided by baseline duration) is indicated on top of each bar.

Fig. 4: CANDLE–NT3: Accuracy vs. Performance Overhead for the Immediate–Rollback (denoted as IR) and Lossy–Forward strategies (denoted FR) for data-parallel training on 8 GPUs. Before represents the state at the beginning of the mini-batch. Baseline represents the state after the failure-free run on the mini-batch. The rate of progress during the back-propagation after which a failure is simulated is denoted as F–XX%. In the case of Lossy–Forward, the number of survivors is mentioned explicitly.

hand, for Scenario 2 (epoch 4, mini-batch 250) and Scenario 3 (epoch 8, mini-batch 500), the increasing rate of progress and number of survivors does not lower the difference to Baseline, as was the case for ResNet–50. Nevertheless, it can be generally observed that the relative difference from Baseline is lower than in the case of ResNet–50 for all approaches.

Finally, the performance overheads for CANDLE–NT3 are depicted in Figure 4b. As can be observed, Immediate–Rollback is 2x slower than Baseline, while Lossy–Forward is very close to Baseline. Since the mini-batch duration is much higher than in the case of ResNet–50, the performance variations play a much smaller role in this case, which explains why the results almost overlap with Baseline.

Overall, for both ResNet–50 and CANDLE–NT3, the Lossy–Forward and Immediate–Rollback strategies incur at least two orders of magnitude less performance overhead than Checkpoint–Restart, even when considering only the loss of progress (on top of which the other fixed overheads need to be added). Given the small difference in accuracy compared with Baseline, we can conclude they are viable alternatives to Checkpoint–Restart in practice, effectively enabling a data-parallel DNN training to recover from failures and unexpected events with negligible overhead. Furthermore, if Lossy–Forward is applicable (which depends on the availability of the survivors to finish the mini-batch), then it is the preferred choice over Immediate–Rollback, as it leads to both less accuracy loss and less performance overhead compared with Baseline.

V. Conclusions

In this paper, we focus on the problem of making the data-parallel training of deep learning models resilient to failures and/or unexpected events that lead to the partial loss of the workers participating in the training (e.g., job pre-emption, downsizing due to elasticity requirements, etc.). In this context, the main challenge is to minimize the performance overhead of recovering the state of the model and continue the training, while at the same time minimizing the loss of accuracy compared with a failure-free run. State-of-art checkpoint-restart

techniques typically adopted in the HPC community can be used to obtain identical results with a failure-free execution (by making stochastic operations deterministic through the use of pseudo-random number generators with fixed seeds). However, they suffer from high performance overheads due to loss of progress, since DNN models are typically checkpointed at the end of each epoch or less frequently.

To address this problem, we introduce two contributions. First, we propose two resilience strategies: (1) capture a potentially inconsistent state after a failure or unexpected event happened and replay the mini-batch; (2) continue the training by averaging the gradients among the survivors using pipeline parallelism. Second, we design and implement a simulation framework that allows fine-grain control and reproducibility over when and under what circumstances to inject failures during the back-propagation. Using the simulation framework, we study the trade-off between performance overhead and accuracy impact in extensive experiments that involve two AI applications in a variety of scenarios, each of which involves different progress rates during the back-propagation and number of survivors (in the case of the second strategy). The results show that both strategies are viable alternatives to checkpoint-restart: they have a minimal impact on accuracy, while reducing the performance overhead several orders of magnitude thanks to avoiding replay overhead (up to 400x).

Encouraged by these results, we plan to explore several follow-up research directions. First, given the high robustness of the DNN models, one idea is to apply more aggressive recovery strategies, such as simply skipping to the next mini-batch instead of replaying the failed mini-batch. Second, we plan to investigate the bias introduced by immediate rollback and the potential impact of outliers left out during gradient averaging. Third, we plan to implement our strategies (especially lossy forward recovery) in a resilient runtime by leveraging other efforts such as fault-tolerant MPI [22].

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contracts DE-AC02-06CH11357 and DE-AC02-05CH11231, program manager Margaret Lentz.

REFERENCES

[1] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3005–3018, 2020.

[2] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *SC'17: The 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2017, pp. 44:1–44:12.

[3] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC '19: The 2019 USENIX Annual Technical Conference*, Renton, WA, 2019, pp. 947–960.

[4] A. Maurya, B. Nicolae, I. Guliani, and M. M. Rafique, "CoSim: A Simulator for Co-Scheduling of Batch and On-Demand Jobs in HPC Datacenters," in *DS-RT'20: The 24th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Prague, Czech Republic, 2020, pp. 167–174.

[5] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, USA, 2010, pp. 1:1–1:11.

[6] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2011, pp. 32:1–32:32.

[7] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brazil, 2019, pp. 911–920.

[8] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, and F. Cappello, "Towards portable online prediction of network utilization using MPI-level monitoring," in *EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems*, Goettingen, Germany, 2019, pp. 1–14.

[9] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 19–28.

[10] ——, "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead," in *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015, pp. 1023–1032.

[11] B. Nicolae, J. Li, J. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *CGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, Melbourne, Australia, 2020, pp. 172–181.

[12] B. Nicolae, J. M. Wozniak, M. Dorier, and F. Cappello, "DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training," in *CLUSTER'20: The 2020 IEEE International Conference on Cluster Computing*, Kobe, Japan, 2020.

[13] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained DNN checkpointing," in *FAST'21: 19th USENIX Conference on File and Storage Technologies*, 2021, pp. 203–216.

[14] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1677–1689, 2021.

[15] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *SC '17: The 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, 2017, pp. 8:1–8:12.

[16] K. Givaki, B. Salami, R. Hojabr, S. M. Reza Tayaranian, A. Khonsari, D. Rahmati, S. Gorgin, A. Cristal, and O. S. Unsal, "On the resilience of deep learning for reduced-voltage fpgas," in *PDP'20: The 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2020, pp. 110–117.

[17] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in deep learning with partial collective operations," in *PPoPP'20: The 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, USA, 2020, p. 45–61.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16: 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, USA, 2016, pp. 770–778.

[19] J. Deng, W. Dong, R. Socher *et al.*, "ImageNet: A large-scale hierarchical image database," in *CVPR'09: Conference on Computer Vision and Pattern Recognition*, Miami, USA, 2009, pp. 248–255.

[20] J. Wozniak, R. Jain, P. Balaprakash *et al.*, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, no. 19, 2018.

[21] "CANDLE Benchmarks," Available online: https://github.com/ECP-CANDLE/Benchmarks.

[22] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 3, p. 244–254, aug 2013.