

# Fragment: Efficient DNN Checkpoint with Relaxed Model Consistency

MANOJ SAHA, Florida International University, Miami, United States

YANZHAO WU, Florida International University, Miami, United States

CLÁUDIA BRITO, INESC TEC, Porto, Portugal and University of Minho, Braga, Portugal

RAJU RANGASWAMI, Florida International University, Miami, United States

JOÃO PAULO, INESC TEC, Porto, Portugal and University of Minho, Braga, Portugal

RICARDO MACEDO, INESC TEC, Porto, Portugal and University of Minho, Braga, Portugal

JANKI BHIMANI, Florida International University, Miami, United States

Training large Deep Neural Networks (DNNs) is inherently resource-intensive and time-consuming, driving significant research into high-frequency checkpointing for enhanced fault tolerance. However, the overhead associated with checkpointing can prolong overall training time. State-of-the-art (SOTA) solutions break down the checkpointing operation into smaller phases, such as snapshot and persist, and pipeline these phases with foreground training operations. Additionally, some strategies leverage faster dynamic or persistent memory devices to minimize overheads. Despite these advancements, SOTA methods still struggle to eliminate training stalls, primarily due to two critical factors: (i) a bandwidth-limited PCIe bus and (ii) consistency requirements for copying the entire model state from the GPU. We propose Fragment—a novel checkpointing solution that relaxes model consistency requirements for fault tolerance, strategically divides the model state into multiple independent pieces (referred to as *fragments*), decreases checkpointing overheads, and increases checkpointing frequency, all without significantly impacting model accuracy. Partitioning complex models into fragments creates challenges that we address by ensuring the integrity of the model state during both creation and restoration, while optimizing the selection of layers. Fragment reduces checkpointing overheads by 15% to 94% while decreasing the total data checkpointed by 43% to 80%, without compromising fault-tolerance needs compared to SOTA solutions. Fragment can also reduce recovery time by up to 97% upon each failure recovery, in the worst case.

This work was supported by the following NSF grants: CSR-2402328, CAREER-2338457, CSR-2406069, CSR-2323100, IIS-2331908, OAC-2530965, CCF-1718335, CNS-1956229, as well as by the DOE Office of Science User Facility, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, using NERSC award DDR-ERCAP0035598, and also supported by gifts from Seagate and NetApp. In addition, this work was supported by the European Regional Development Fund (ERDF) through the Innovation and Digital Transition program (COMPETE 2030) under Portugal 2030, within the scope of the project CDMS, reference 17409 (COMPETE2030-FEDER-01193000).

Authors' Contact Information: Manoj Saha (corresponding author), Florida International University, Miami, Florida, United States; e-mail: msaha002@fiu.edu; Yanzhao Wu, Florida International University, Miami, Florida, United States; e-mail: yawu@fiu.edu; Cláudia Brito, INESC TEC, Porto, Porto District, Portugal and University of Minho, Braga, Braga, Portugal; e-mail: claudia.v.brito@inesctec.pt; Raju Rangaswami, Florida International University, Miami, Florida, United States; e-mail: raju@cis.fiu.edu; João Paulo, INESC TEC, Porto, Porto District, Portugal and University of Minho, Braga, Braga, Portugal; e-mail: joao.t.paulo@inesctec.pt; Ricardo Macedo, INESC TEC, Porto, Porto District, Portugal and University of Minho, Braga, Braga, Portugal; e-mail: ricardo.g.macedo@inesctec.pt; Janki Bhimani, Florida International University, Miami, Florida, United States; e-mail: jbhimani@fiu.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1544-3973/2026/09-ART99

<https://doi.org/10.1145/3830082>

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → **Machine learning**; • **Information systems** → **Information storage technologies**;

Additional Key Words and Phrases: Deep neural network, fault tolerance, model consistency, ML checkpoint

#### ACM Reference Format:

Manoj Saha, Yanzhao Wu, Cláudia Brito, Raju Rangaswami, João Paulo, Ricardo Macedo, and Janki Bhimani. 2026. Fragment: Efficient DNN Checkpoint with Relaxed Model Consistency. *ACM Trans. Arch. Code Optim.* 23, 3, Article 99 (September 2026), 26 pages. <https://doi.org/10.1145/3830082>

## 1 Introduction

The widespread adoption of **Deep Neural Networks (DNNs)** across various domains has spurred extensive research. However, DNN training is resource-intensive, often running for days or weeks, even in multi-GPU clusters. Consequently, hardware, network, system (e.g., out-of-memory), power (e.g., outages), and application failures are common in DNN training [11]. In addition, many DNN training jobs use low-cost ad-hoc instances (e.g., AWS spot instances) where idle resources in a data center are rented at low cost, but are interrupted and taken away at a moment's notice as resource demand increases [25]. In such cases, frequent DNN checkpoints are critical for rapid recovery.

To minimize work loss due to failures or interruptions, an ideal checkpointing mechanism would save the model state after each iteration [40, 44]. However, the checkpointing time for large DNN models can far exceed the iteration time. For example, in our experiments, the checkpointing time for VGG16 is  $13\times$ – $112\times$  longer than the average iteration time, depending on the configuration. This per-operation bottleneck fundamentally limits checkpoint frequency; even when amortized over an epoch, checkpointing increases training time by up to 94% compared to training without checkpointing (Section 5). GEMINI [40] reported that checkpointing the MT-NLG model, a 530-billion parameter transformer trained across 2,240 NVIDIA A100 GPUs using 3D parallelism [34], over a 20 Gbps network storage can take up to 42 minutes. While some of this delay can be hidden by pipelining checkpointing with foreground training, large model sizes make space and time overheads, which grow linearly with the checkpoint frequency, untenable.

To mitigate this, existing solutions reduce checkpoint frequency, with some models checkpointed as infrequently as every three hours to reduce space overheads [40]. However, this increases recovery time, i.e., the time to load the latest checkpointed model state and re-train the model to restore model accuracy prior to failure. While loading time for the model state is usually low, time to re-train can be high. For example, failure recovery for MT-NLG takes an average of 105 minutes [40]. These overheads were reported under production-scale training conditions, underscoring that checkpointing costs are representative of real-world large-model training rather than exceptional cases. Intuitively, unless stall time, i.e., the time foreground training operations wait for checkpoint operations to finish, is eliminated or reduced to a minimal level, frequent checkpointing cannot save more in recovery time than it wastes in training stalls.

However, none of the existing checkpointing methods can both reduce stalls and space overheads without compromising checkpoint frequency [6, 11, 25, 44] due to large checkpoint size and PCIe bandwidth limitations. In this work, we propose Fragment, a novel checkpointing method that can reduce and manage checkpointing stalls. We explore the fundamental design choice of relaxing the assumption that DNN checkpoints need to capture the complete training state (i.e., model state, optimizer state, data loader state, and other hyperparameters) in each checkpoint. Our core insight is that the entire model, along with the optimizer state and other training states, can be divided into multiple chunks, each chunk referred to as a *fragment*. Each fragment consists of a fraction of the total model state and the corresponding optimizer state parameters that are captured

consistently<sup>1</sup> during a checkpointing operation. All training operations, including weight updates and gradient computations, proceed with full consistency; the relaxation applies solely to what is captured in the persistent checkpoint. The state of the model is captured across multiple fragments rather than saving the entire model at each checkpoint. This reduces checkpoint stalls and space overhead without sacrificing checkpoint frequency. However, achieving this requires carefully designed fragment creation algorithms and an optimized checkpointing workflow.

The **major contributions** of our work are as follows.

- We conduct a comprehensive analysis to evaluate the feasibility of restoring DNN training states from layer-wise checkpoints that fuse the latest snapshots of various layers and optimizer states captured at different times, while preserving model accuracy.
- We design novel methods for fragment creation, efficient tracking and restoration, and optimizer state management to preserve model accuracy and minimize training disruptions.
- We develop optimization techniques for fragment-based parallel and out-of-order checkpointing and restoration to facilitate high-frequency checkpointing.
- We implement Fragment by optimizing the snapshot creation and restoration phases during checkpointing and failure recovery in PyTorch.

We evaluate Fragment with different image classification models, workloads, and optimizer configurations, along with comparative analysis against existing strategies. Compared to SOTA checkpointing techniques, Fragment reduces the time of checkpointing stalls by 15% to 94% while decreasing the total data checkpointed by 43% to 80%. Additionally, we assess the recovery time of Fragment by looking at the final model accuracy instead of the immediate model accuracy upon failure recovery. Fragment can reduce the failure recovery time by up to 97% in the worst case, when proactive checkpointing is enabled.

## 2 DNN Convergence Analysis

DNN training jobs are iterative in nature. Let us assume a DNN application, that is, training a model consisting of  $N$  layers, identified as  $l_1, l_2, \dots, l_N$ . Each layer  $l_n$  consists of one or more tensors. Tensors are groups of parameters  $W$ , e.g., weight  $x_n$  and bias  $b_n$ . Let  $W^{(i)} \in \mathbb{R}^d$  denote the complete set of trainable DNN model parameters at iteration  $i$ . The training update is:

$$W^{(i+1)} = f(W^{(i)}, D_{train}^{(i)}, OH_{train}^{(i)}), \quad (1)$$

where  $f$  is the DNN training algorithm. For simplicity, we can omit the training data  $D_{train}^{(i)}$  and non-trainable parameters  $OH_{train}^{(i)}$  that stores training state at iteration  $i$ , from Equation (1) when analyzing model convergence. This transforms Equation (1) into

$$W^{(i+1)} = f(W^{(i)}).$$

The trainable parameters  $W$  is considered to have converged to an optimal  $W^*$ . If a training job fails before converging, the work is lost and the training job is restarted. To reduce the amount of lost work due to failures, trainable parameters  $W$  are checkpointed periodically, and restarted from the latest checkpoint if a failure occurs. In existing checkpointing schemes, all trainable parameters  $W$  are saved during each checkpoint to maintain the consistency of the model and to eliminate any perturbations upon failure recovery. In Fragment, we explore an alternative method of checkpointing by relaxing model consistency during checkpointing. Suppose the model parameters

<sup>1</sup>Fragment checkpoints are captured consistently from a system perspective, even when model consistency requirements are relaxed.

$W$  are partitioned into  $M$  disjoint fragments for a training job running for  $n$  iterations, such that

$$W = \bigcup_{m=1}^M w_m, \quad w_m \cap w_n = \emptyset \text{ for } m \neq n. \quad (2)$$

At each checkpoint, only one fragment  $w_m$  is saved. Upon failure recovery at iteration  $k$ , the restored parameters are

$$\tilde{W}^{(k)} = \bigcup_{m=1}^M w_m^{(t_m)}, \quad (3)$$

where  $t_m \leq k$  is the iteration at which the latest checkpoint for fragment  $w_m$  was taken. We define the perturbation at recovery as:

$$\delta^{(k)} = \tilde{W}^{(k)} - W^{(k)}.$$

After a failure recovery at iteration  $k$ , the training resumes from fragment checkpoints, i.e., the perturbed state  $\tilde{W}^{(k)}$ ,

$$\tilde{W}^{(k)} = W^{(k)} + \delta^{(k)},$$

where  $\delta^{(k)}$  is the perturbation vector due to restoring fragments from earlier iterations. The subsequent update is:

$$W^{(k+1)} = f(\tilde{W}^{(k)}),$$

which is equivalent to the perturbed iteration in [30]:

$$W^{(k+1)} = f(W^{(k)} + \delta^{(k)}).$$

Without loss of generality, we assume the DNN training algorithm is iterative-convergent, i.e., there exists  $0 < c < 1$  such that:

$$\mathbb{E}\|f(W) - W^*\| \leq c \cdot \mathbb{E}\|W - W^*\|, \quad (4)$$

for the optimal  $W^*$ . We then have the model parameters after  $t$  further iterations as  $W^{(k+t)}$ . By recursively applying Equation (4), we have

$$\mathbb{E}\|W^{(k+t)} - W^*\| \leq c^t \|\tilde{W}^{(k)} - W^*\|.$$

Given  $\tilde{W}^{(k)} = W^{(k)} + \delta^{(k)}$ , by the triangle inequality, we have

$$\|\tilde{W}^{(k)} - W^*\| \leq \|W^{(k)} - W^*\| + \|\delta^{(k)}\|.$$

Therefore, we can derive

$$\mathbb{E}\|W^{(k+t)} - W^*\| \leq c^t (\|W^{(k)} - W^*\| + \|\delta^{(k)}\|).$$

It shows that the impact of the perturbations  $\delta^{(k)}$  decays exponentially with the number of iterations after failure recovery. The self-corrective nature of the iterative-convergent algorithm ensures that even if the recovered model parameters are not perfectly up-to-date (as in Fragment), the DNN training algorithm can still quickly return to its original trajectory toward the optimal  $W^*$ .

**Bounds on Perturbation:** In practice, the perturbation  $\|\delta^{(k)}\|$  is determined by the staleness of each fragment at recovery. If each fragment is checkpointed at least every  $s$  iterations, and the per-iteration parameter change is bounded by  $\gamma$ , then

$$\|w_m^{(k)} - w_m^{(t_m)}\| \leq s \cdot \gamma,$$

so we have

$$\|\delta^{(k)}\|^2 = \sum_{m=1}^M \|w_m^{(k)} - w_m^{(t_m)}\|^2 \leq M(s\gamma)^2, \quad (5)$$

and hence

$$\|\delta^{(k)}\| \leq s\gamma\sqrt{M}. \quad (6)$$

By increasing the checkpointing frequency (decreasing  $s$ ), the perturbation can be very small, and the effect on DNN convergence becomes negligible.

**Iteration Cost Bound:** To reach a target accuracy  $\epsilon$ , i.e.,  $\mathbb{E}\|W^{(k+t)} - W^*\| < \epsilon$ , it suffices that:

$$c^t (\|W^{(k)} - W^*\| + \|\delta^{(k)}\|) < \epsilon. \quad (7)$$

Compared to the failure-free case (where  $\delta^{(k)} = 0$ ), the extra number of iterations required to reach  $\epsilon$ -accuracy is at most:

$$\frac{\log\left(1 + \frac{\|\delta^{(k)}\|}{\|W^{(k)} - W^*\|}\right)}{\log(1/c)}, \quad (8)$$

as shown by [30]. Therefore, as long as the perturbation  $\|\delta^{(k)}\|$  is bounded, Fragment checkpointing guarantees that DNN training will converge to the same optimal  $W^*$  as in the failure-free case, with at most a small increase in the number of iterations. Our empirical evaluation also demonstrates that Fragment preserves DNN model convergence under failure recovery.

### 3 Failure Recovery in DNN Training

In this section, we formalize DNN failure recovery using checkpoints. Earlier in Equation (2), for simplicity of convergence analysis, we assumed only trainable parameters  $W$  are checkpointed. From a DNN failure recovery perspective, each full checkpoint  $Z^{(i)}$  triggered at iteration  $i$  is a union of trainable parameters  $W^{(i)}$  (a.k.a. model state) and non-trainable parameters  $OH^{(i)}$ . The non-trainable parameters  $OH^{(i)}$  consists of optimizer state  $O^{(i)}$ , data iterator state  $d_{train}^{(i)}$  and other hyperparameters  $S_{train}^{(i)}$ . Thus, a full checkpoint  $Z^{(i)}$  is defined as:

$$Z^{(i)} = W^{(i)} \cup O^{(i)} \cup d_{train}^{(i)} \cup S_{train}^{(i)}.$$

Based on Equation (2), a fragment checkpoint  $Z_m^{(i)}$  is defined as:

$$Z_m^{(i)} = w_m^{(i)} \cup O_m^{(i)} \cup d_{train}^{(i)} \cup S_{train}^{(i)}. \quad (9)$$

From a tooling<sup>2</sup> point of view, Fragment  $Z_m^{(i)}$  contains weights and biases of  $\beta$  consecutive layers, where  $\beta = e - s$ ,  $1 \leq \beta < n$ ,  $1 \leq s \leq e \leq n$  and the value of  $\beta$  may or may not be the same for two different Fragments  $Z_u$  and  $Z_m$ . In short,

$$Z_m^{(i)} = l_e^{(i)} \cup \dots \cup l_s^{(i)} \cup o_s^{(i)} \cup \dots \cup o_e^{(i)} \cup d_{train}^{(i)} \cup S_{train}^{(i)}. \quad (10)$$

In other words, each Fragment  $Z_m^{(i)}$  is a checkpoint containing weights and biases of  $\beta$  consecutive layers, along with their corresponding optimizer state, data loader state and other hyperparameters. The value of  $\beta$  for a Fragment  $Z_m^{(i)}$  can be decided based on different criteria (see Section 4.1).

When failure happens in DNN training at iteration  $j$ , training is restarted from a previously saved checkpoint  $Z^{(i)}$  at iteration  $i$ , where  $i < j$ . The lost iterations, equal to  $|j - i|$ , are re-trained and the recovery cost can be defined as:

$$T_{recovery}^{(j-i)} = T(f(W^{(i)}, D_{train}^{(i)}, S_{train}^{(i)})) * |j - i|. \quad (11)$$

Failure recovery schemes in DNN training are primarily concerned with correctly restoring training state (i.e.,  $(W^{(i)}, D_{train}^{(i)}, S_{train}^{(i)})$ ) and reducing recovery cost (i.e.,  $T_{recovery}^{(j-i)}$ ). The recovery cost

<sup>2</sup>Tools are deep learning frameworks such as PyTorch, TensorFlow, Darknet and so on.

reduction can be achieved by increasing checkpoint frequency, which reduces the distance  $|j - i|$  (i.e., number of lost iterations). However, optimal checkpoint frequency depends on checkpointing overheads and system constraints. Hence, recent DNN checkpointing solutions are also concerned with reducing additional training time incurred due to checkpointing overheads  $T_{overhead}$  and reducing system resource (e.g., storage space) utilization by checkpoints. Thus, we can view the primary goal of DNN checkpointing solution is to reduce overall training time  $T$ , without negatively impacting model convergence. The total training time  $T$  of a training job with  $n$  iterations,  $c$  checkpoints and  $p$  failures can be defined as Equation (12), where  $T_{compute,avg}$  is the average time to train an iteration,  $T_{overhead,avg}$  is the average overhead incurred for checkpointing model state and  $T_{recovery,avg}$  is the average time to recover from failures.

$$T = T_{compute,avg} * n + T_{overhead,avg} * c + T_{recovery,avg} * p. \quad (12)$$

### 3.1 Model Consistency

The current DNN failure recovery techniques are based on the notion that the complete training state (i.e.,  $W^{(i)}$ ,  $D_{train}^{(i)}$ ,  $S_{train}^{(i)}$ ) needs to be restored to bound recovery time (i.e.,  $T_{recovery}$ ) only for lost training iterations. Existing checkpointing schemes assume that any perturbations in  $W^{(i)}$ ,  $D_{train}^{(i)}$ , or  $S_{train}^{(i)}$  will lead the model far from convergence. It is assumed to prolong training time  $T$ , as it may take longer for the training job to converge. However, recent works on DNN training demonstrate that consistency of model parameters (i.e.,  $W^{(i)}$ ) can be relaxed without impacting model accuracy  $M(W^{(n)}, D)$  [4, 30, 43]. Robust training algorithms tend to have self-correction mechanisms that allow adjusting partial model state (i.e., some of the parameters in  $W^{(i)}$ ) to a prior state without any impact on final model accuracy  $M(W^{(n)}, D)$  [4, 43]. In fact, most DNN models are over-parameterized [26], and not all model parameters are critical for the model accuracy [43]. Changing the state of non-critical or robust parameters in a trained (i.e., converged) neural network to an earlier or completely random state does not significantly impact model accuracy  $M(W^{(n)}, D)$  [43]. Researchers at Google [4] demonstrated that systemic reinitialization and randomization can be used for all parameters in a neural network during training without sacrificing model accuracy  $M(W^{(n)}, D)$ . SCAR [30] also demonstrated that partial recovery of model state in a **parameter server (PS)** training architecture with linearly converging training algorithms can reduce recovery time  $T_{recovery}$ . However, SCAR's [30] parameter priority-based checkpointing cannot reduce checkpointing overhead  $T_{overhead}$  and requires additional compute resources and time to identify which parameters should be prioritized for each checkpoint.

The tolerance of model inconsistency has also been extensively studied in adjacent domains. In asynchronous distributed training, Hogwild [31] demonstrated that lock-free parallel SGD converges despite workers operating on stale parameter versions, provided updates are sparse. More broadly, asynchronous SGD [9] allows gradient updates computed on outdated parameters, with theoretical guarantees that bounded staleness preserves convergence [16]. In federated learning, FedAvg [24] tolerates substantial temporal divergence by allowing clients to perform multiple local updates before synchronizing, effectively training on inconsistent model states across participants. Fragment's relaxed consistency is analogous to these paradigms but differs in a key aspect: in asynchronous SGD and federated learning, staleness is a continuous phenomenon throughout training, whereas in Fragment, inconsistency arises only at the moment of failure recovery (a rare event). Furthermore, after recovery, all subsequent iterations proceed with fully consistent weight updates Equation (13). This makes Fragment's perturbation impact strictly less severe than what these systems tolerate routinely.

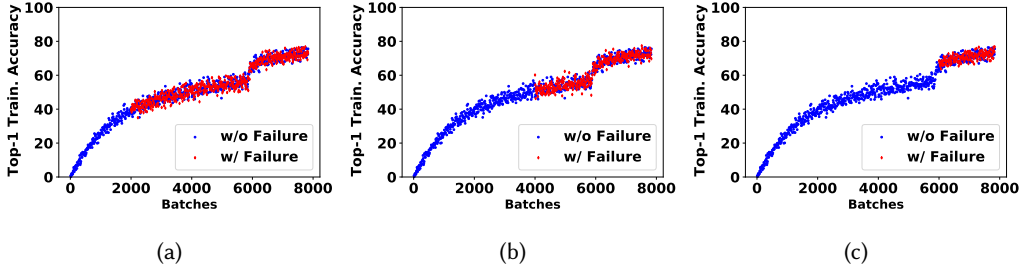


Fig. 1. Failure recovery using Fragment [ResNet18, Tiny Imagenet, learning rate 0.001]. (a) Failure after 2000 batches. (b) Failure after 4,000 batches. (c) Failure after 6,000 batches.

Inspired by the above observations, we demonstrate that the idea of checkpointing and restoring with Fragment. In Fragment, the model parameters  $W$  is divided into independent chunks or fragments  $w_1, w_2, \dots$  for checkpointing purposes and only one fragment is saved at each checkpoint based on Equation (2), (9), and (10). Empirical evidence shows that model parameters restored in this method achieve comparable training accuracy  $M(W^{(i+1)}, D_{train}^{(i+1)})$  after restore without increasing overall training time  $T$ , when compared to training performance without failure. As a proof-of-concept experiment, we train a Resnet18 [15] image classification model on Tiny Imagenet dataset [20] for 40 epochs without any failure. We train the same model again with the same parameter initialization and checkpoint the training state using Fragment. The whole model is divided into three fragments, each consisting of  $\sim 10$  consecutive layers. As illustrated in Figure 1, we conduct three targeted experiments simulating failures after 2,000, 4,000, and 6,000 iterations, respectively. Simulating failures at different stages of training is done to assess if failure recovery using fragments work regardless of how far or close the model is from convergence. Following each setback, we rebuild the training state from the latest fragments as defined in Equation (2), (9), and (10), and proceed with the training. The Top-1 training accuracy of both the uninterrupted baseline model and the model restored using fragment checkpoints follows the same trendline, as depicted in Figure 1. The preliminary results empirically validate the potential of Fragment as a viable DNN checkpointing method.

### 3.2 Recovery Time

The training time defined in Equation (12) assumes an ideal case where the failure recovery scheme will faithfully restore the training state using a full checkpoint. In such a case, non-trainable parameters  $S_{train}^{(i)}$  capture all training state and device state that impacts function  $f$  in Equation (1). During restore, only the lost iterations are computed again, incurring recovery cost  $T_{recovery}$  defined in Equation (11). In reality, whether or not the full training and device state  $S_{train}^{(i)}$  can be faithfully captured and restored for a deterministic restart depends on tooling. A group of researchers [32] observed that TensorFlow [3] and Chainer [36] do not provide deterministic restart, especially with distributed training, even though failure-free training can be deterministic. In our experiments, we observe this complexity with PyTorch [29] as well for distributed training with CheckFreq’s resumable data iterator [25]. In our experiments, we observed that model accuracy upon failure recovery can be lower, higher, or marginally equal to model accuracy prior to interruptions.

In the absence of deterministic restart and proposed perturbation in  $W^{(i)}$ , the question arises as to whether recovery time  $T_{recovery}$  can be limited to lost training iterations only. Theoretically, perturbation(s) in  $W^{(i)}$ ,  $D_{train}^{(i)}$  or  $S_{train}^{(i)}$  may impact immediate model performance upon recovery

and final model convergence time. Perturbations in model parameters  $W^{(i)}$  similar to Equation (3) deviate the weight update operation from the failure-free path. In a full checkpoint, the weight update operation and its corresponding gradient  $\nabla g$  can be elaborated as:

$$(w_1^{(i+1)} \cup \dots \cup w_m^{(i+1)}) = (w_1^{(i)} \cup \dots \cup w_m^{(i)}) - \nabla g(w_1^{(i)} \cup \dots \cup w_m^{(i)}). \quad (13)$$

In contrast, the weight update using perturbed model parameters can be viewed as:

$$(w_1^{(i+1)} \cup \dots \cup w_m^{(i+1)}) = (w_1^{(i-k)} \cup \dots \cup w_m^{(i)}) - \nabla g(w_1^{(i-k)} \cup \dots \cup w_m^{(i)}). \quad (14)$$

In both Equation (13) and (14), the weight update operation maintains the validity of the gradients for its corresponding parameter chunk  $w_m$ . Semantically, Equation (13) is the same as synchronous distributed training on  $m$  machines, whereas Equation (14) is similar to asynchronous distributed training. The question is, *do perturbations in model parameters due to failure and weight updates similar to Equation (14) after failure recovery have a lasting effect on model convergence?* In Fragment, such perturbations affect the gradient and weight update calculations only once after each failure recovery, as defined in Equation (14). Hence, perturbations can be deemed insignificant compared to the overall training iterations. As for the effect of such perturbations, SCAR [30] shows theoretical analysis that both linear and sub-linear training algorithms wash out the impacts incurred by perturbations in  $W^{(i)}$  quickly, when perturbations are not too large and do not move too far away from the convergence target  $W^*$  (see Equation (5)–(8)). In other words, if failures are not cluster-wide and fragment checkpoints are triggered more often than full checkpoints, it reduces the amount of perturbations in the training environment, such as PS. However, SCAR [30] fails to address an important question. When the entire training cluster fails and both full and fragment checkpoints are triggered at the same frequency, can the latter reach the convergence target at the same time? The empirical evidence from our experiments shows that the model convergence time does not increase when the iterative distance  $|i - k|$  between the checkpoints is small, even with a very high degree of perturbations. Intuitively, when we save model parameters with frequent fragment checkpoints, usually after every few iterations, the distance between the perturbed and unperturbed parameters is small enough. If the perturbed model parameters caused oscillations away from the convergence target  $W^*$ , the distance  $\|W^{(i)} - W^*\|$  would be small enough to allow for rapid recovery (see Equation (6)).

### 3.3 Frequent Checkpointing

Reducing failure recovery time requires increasing checkpoint frequency. The checkpoint frequency  $\mu_{cp}$  is a function of checkpoint size  $\alpha_s$ , iteration time  $T_{compute}^{(i)}$ , and system constraints  $\phi$ . The system constraints include memory and interconnect bandwidth of the various devices (e.g., GPU, DRAM, and disk) involved in the checkpoint process. If we had a computing system with infinite memory, storage media, and interconnect bandwidth, then the checkpoint frequency would be every iteration. However, practical computing systems have limited media, interconnect bandwidth, and compute power. If sufficient media bandwidth  $md_{bw}$  and interconnect bandwidth  $ic_{bw}$  are available for checkpointing operations, the theoretical checkpointing frequency  $\mu_{cp}$  can be simplified as:

$$\mu_{cp} = \frac{T_{compute}^{(i)}}{T_{compute}^{(i)} + \frac{2*\alpha_s}{\min(md_{bw}, ic_{bw})}}. \quad (15)$$

Iteration time  $T_{compute}^{(i)}$  in Equation (15) does not show high variance, while effective media and interconnect bandwidth is harder to control. Hence, realizing frequent checkpoints primarily depends on reducing checkpointing overheads. Researchers have introduced various methods that

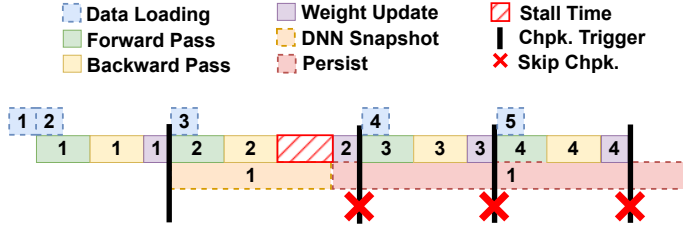


Fig. 2. Typical DNN workflow with asynchronous checkpointing. *Numbers correspond to iterations.*

alter the checkpointing workflow to reduce training overhead  $T_{overhead}$ . Asynchronous checkpointing methods [6, 11, 25, 27, 28] try to reduce training stalls by performing checkpoint operations in the background while training can proceed in the foreground (see Figure 2). Asynchronous checkpoint proposed by CheckFreq [25] divides the checkpoint operation into snapshot and persist phases. These operations can execute in the background, while training operations such as forward-backward pass and weight update continue in the foreground (see Figure 2). Thus, the asynchronous checkpointing mechanism increases checkpoint frequency, and transforms checkpoint frequency  $\mu_{cp}$  from Equation (15) into

$$\mu_{cp} = \frac{T_{compute}^{(i)}}{\max(T_{compute}^{(i)}, \frac{2 * \alpha_s}{\min(md_{bw}, ic_{bw})})}. \quad (16)$$

However, existing asynchronous checkpointing schemes still stall foreground training if CPU snapshot operation (i.e., GPU-CPU copy) cannot be completed before the model parameter update of the next iteration to maintain consistency of the model parameters. Stalls during CPU snapshot operation occur due to PCIe bandwidth bottlenecks [28]. Checkpointing solutions have taken two different routes to solve snapshot bottlenecks. SCAR [30], DeepFreeze [27], DCP [1], MCP [2], and ByteCheckpoint [37] use rank-based snapshot sharding across distributed machines, where the latter two are specific to **large language models (LLM)**. The domain-specific and closed-source nature of these solutions hinders their wide adoption (see Table 1). CheckFreq [25] took a different route by introducing GPU snapshots (i.e., GPU-GPU copy) for checkpointing. This approach can reduce runtime overhead by at least 96.5% compared to the prevalent synchronous checkpointing for different image classification models [25].

However, GPU snapshots are not always feasible. Multiple works, including SWIFT [44], LightCheck [6], and Check-N-Run [11], have pointed out how GPU snapshots can be impractical due to memory constraints for many DNN training workloads. GPU snapshots increase the overall operations needed for a checkpoint, regardless of synchronous or asynchronous snapshots, increasing the overall checkpoint time and decreasing checkpoint frequency. We observe the impact of GPU and CPU snapshot techniques on training time per epoch in Figure 3. As demonstrated in Figure 3, we observe that the asynchronous snapshot scheme in CheckFreq [25] cannot eliminate stalls. LightCheck [6] observed that newer techniques such as pipelining snapshot operations by tensors on memory devices also cannot eliminate snapshot stalls, since these techniques checkpoints the complete model state and cannot bypass the bandwidth bottlenecks of the system.

### 3.4 Storage Cost

The systems view of checkpointing is also concerned with the storage cost of storing checkpoints. The choice of storage media can increase persist time and reduce checkpoint frequency. **Solid State Drive (SSD)** can persist large checkpoints significantly faster than **Hard-Disk Device**

Table 1. Comparison of Existing DNN Checkpointing Schemes

| Solution            | Model Consistency | Checkpoint Optimization                                        | Other Notable Features                                                                            |
|---------------------|-------------------|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| CheckFreq [25]      | Strong            | - Asynchronous<br>- Frequent checkpoints                       | - Open Source                                                                                     |
| DeepFreeze [27]     | Strong            | - Asynchronous<br>- Rank-based sharding                        | - HPC friendly                                                                                    |
| DCP [1]             | Strong            | - Rank-based sharding                                          | - Distributed training only<br>- Open source                                                      |
| MCP [2]             | Strong            | - Rank-based sharding                                          | - Distributed training only<br>- Open source                                                      |
| ByteCheckpoint [37] | Strong            | - Rank-based sharding                                          |                                                                                                   |
| SCAR [30]           | Relaxed           | - Rank-based sharding                                          | - Parameter Server only<br>- Parameter prioritization<br>- Cluster failure averse                 |
| LC-Checkpoint [7]   | Strong            | - Compression                                                  | - SGD-only                                                                                        |
| Check-N-Run [11]    | Strong            | - Differential checkpoint                                      | - Recommendation model friendly                                                                   |
| FastPersist [39]    | Strong            | - Rank-based sharding                                          | - NVMe SSD only<br>- Distributed env. only                                                        |
| GPM [28]            | Strong            | - Frequency improvement                                        | - In-memory checkpointing<br>- Persistent memory only<br>- GPU-native<br>- Open source            |
| LightCheck [6]      | Strong            | - Fine-grained asynchronous checkpointing                      | - In-memory checkpointing<br>- Persistent memory only<br>- Distributed env. only<br>- Open source |
| LATTICE [33]        | Strong            | - Space optimized model versions                               | - In-memory versioning<br>- Persistent memory only<br>- Open source*                              |
| Portus [22]         | Strong            | N/A                                                            | - In-memory checkpointing<br>- Persistent memory only<br>- RDMA only                              |
| Gemini [40]         | Strong            | N/A                                                            | - Just-in-Time checkpointing<br>- Node recovery only                                              |
| Swift [44]          | Strong            | N/A                                                            | - Just-in-Time checkpointing<br>- Node recovery only<br>- Open source                             |
| JIT-checkpoint [13] | Strong            | N/A                                                            | - Just-in-Time checkpointing<br>- Node recovery only                                              |
| Mnemosyne [41]      | Strong            | N/A                                                            | - Just-in-Time checkpointing<br>- Device proxy-based node recovery                                |
| AdaCheck [23]       | Strong            | - Redundancy utilization                                       | - Optimized for partial failures                                                                  |
| Fragment            | Relaxed           | - Heuristic-based snapshot sharding<br>- Frequency improvement | - Cluster recovery<br>- To be open sourced                                                        |

[HPC=High-Performance Computing; SGD=Stochastic Gradient Descent].

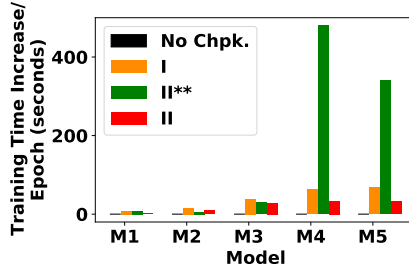


Fig. 3. No snapshot scheme can eliminate stalls, resulting in longer training time per epoch [I=GPU snapshot(synchronous), II\*\*=CPU snapshot(synchronous), II=CPU snapshot(asynchronous), see Table 2].

(HDD), while **Persistent Memory (PMem)** devices can persist checkpoints faster than SSD. Thus storage bandwidth, a type of system constraint, directly affects checkpoint frequency. Storage bandwidth bottlenecks lead to long persist time and less frequent checkpoints. Hence, GPM [28], LightCheck [6], Portus [22] and LATTICE [33], leverage the relatively lower latency of PMem devices, compared to SSD and HDD, to persist large checkpoints efficiently. This, however, increases the infrastructure cost even further, and the limited availability of such devices (e.g., 3D-Xpoint) impacts the wide adoption of direct persistence-based checkpoints. Gemini [40], Swift [44], JIT-checkpoint[13], and Mnemosyne [41], on the other hand, propose failure recovery mechanisms by retrieving in-memory training state from other active nodes in the training cluster. However, such failure recovery methods lose all progress in case of the training cluster failure and need to use traditional checkpointing solutions periodically to save cluster-wide progress.

Another approach to reduce storage cost emphasizes checkpoint size reduction. The storage space required to store a large number of checkpoints, which is used to support offline analysis and other applications, can also be very large. Although space utilization can be reduced by maintaining only the latest checkpoints on disk, when only fault tolerance is needed [25], degradation of device lifetime continues at the same rate, since the same amount of data is written to disk for checkpointing training state. Given the prevalence of lifetime-limited NAND Flash storage devices (i.e., SSDs) in current ML infrastructure, this is of particular concern. Hence, we define the storage cost  $\pi$ , in Equation (17), as the amount of data written for checkpointing training state, where the primary goal is to statically or dynamically reduce compression factor  $\xi^{(l)}$  for each checkpoint.

$$\pi = \sum_{l=1}^c \alpha_s * \xi^{(l)}, \quad 0 < \xi^{(l)} \leq 1. \quad (17)$$

Solutions proposed by [7] and [18] use dynamic compression to reduce checkpoint size and storage cost. This, however, increases the checkpoint time due to dynamic compression overheads, results in less frequent checkpoints and increases recovery time  $T_{recovery}$  in Equation (12). Incremental differential checkpointing for large recommendation models proposed by Check-N-Run [11] also reduces snapshot size. However, incremental differential checkpointing cannot be generalized for other types of models, such as CNNs, as most of the parameters change in each iteration for the latter. For example, in our experiments with CIFAR-10 [20] and Tiny ImageNet [21] datasets for different image classification models, 99.9% of the parameters change in each iteration, making the techniques proposed by Check-N-Run [11] ineffective in reducing storage cost. AdaCheck [23] reduces storage cost by utilizing state redundancy in distributed training. However, it is optimized for partial cluster failures LLM training. MoEvemement [12] deploys sparse checkpointing to abate

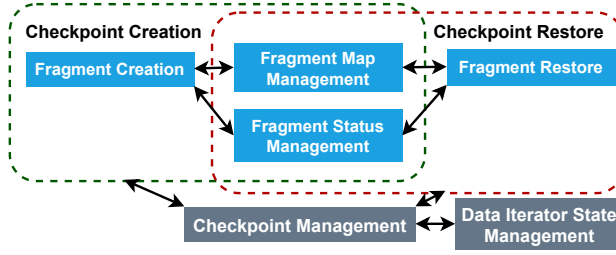


Fig. 4. Fragment design components [colored Royal Blue are newly proposed].

storage overhead, but supports **Mixture-of-Experts (MoE)** models only. Hence, efficient methods to persist training state for DNN models require further investigation.

To summarize, Fragment introduces a high-frequency checkpointing scheme that relaxes the consistency requirements of checkpoints and can recover from failure by splicing different fragments without hampering model convergence. Unlike the method proposed by SCAR [30], Fragment works with non-uniform parameter partitions and checkpoint frequency. Instead of prioritizing parameter partitions with highest gradient values during each checkpoint, which requires additional GPU memory and compute resources, Fragment avoids using additional memory and compute resources and prioritizes frequency of checkpoints.

## 4 Fault Tolerance Using Fragment

In this section, we demonstrate **Fragment**'s architecture, operations and optimizations. Our goal is twofold: first, to optimize checkpoint size to substantially reduce operational stalls during training; and second, to ensure minimal progress loss upon recovering after failures. The challenges include formalizing Fragment to enable efficient fragment creation without impacting model convergence and training time. We also need to optimize the checkpointing workflow to leverage Fragment for more frequent checkpointing.

### 4.1 Checkpointing and Restoring in Fragment

The design components of Fragment can be divided into two major modules—*Checkpoint Creation* and *Checkpoint Restore*, as depicted in Figure 4. The *Fragment Map Management* component, which maps model layers into different fragments, and *Fragment Status Management* component, responsible for keeping track of checkpoint status and persisted checkpoints for each fragment, are shared among the two modules. The *Fragment Creation* and *Fragment Status Management* components work in tandem for fragment checkpoints. The *Checkpoint Management* component provides the *Checkpoint Creation* module with the *data iterator state*. During restore, the fragments are restored to reconstruct the complete DNN model, along with saved *data iterator state*.

**Checkpointing Assumptions.** Fragment operates at the DNN layer granularity. Each fragment captures a consistent snapshot of its assigned layers' weights, biases, and corresponding optimizer state. Snapshots are taken between training iterations, after the weight update step completes and before the next forward pass begins (see Figure 2). At this point, no synchronization primitives (e.g., barriers, locks, or atomics) span across layers, since all inter-layer dependencies are resolved within a single iteration. Therefore, barriers, locks, and atomics are not part of the checkpointed state and do not interact with fragment boundaries. In **distributed data-parallel (DDP)** training, gradient synchronization across ranks is completed before the weight update step; Fragment's snapshot is taken after this synchronization, ensuring each rank's fragment is internally consistent.

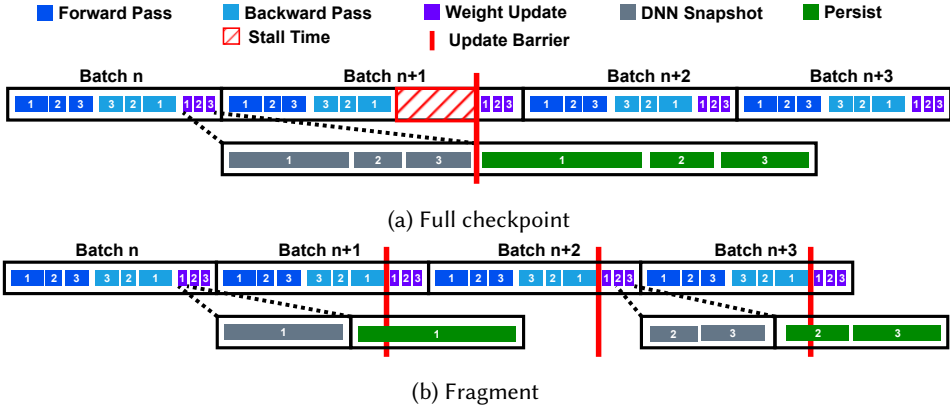


Fig. 5. (a) Asynchronous complete model state snapshot can stall update operations in the next iteration. (b) Fragment snapshots reduce stalls by making the snapshot size smaller [1,2,3 denote model layers].

**4.1.1 Creating Fragment.** Efficiently partitioning models, especially those with complex architectures, into fragments presents significant challenges, including maintaining the integrity of the model’s state during fragment creation and restoration, optimizing the selection of layers for snapshotting, and ensuring minimal disruption to the training process. Additionally, balancing the trade-off between checkpoint size and frequency while avoiding any detrimental impact on model accuracy and recovery time is critical.

The key to our solution lies in recognizing the optimal size of a fragment based on system bottlenecks and model architecture (as depicted in Figure 5). We use Algorithm 1 to create a *fragment map* to record the details of how the DNN model layers are grouped into different fragments. This persistent map is generated or reconstructed just once at the start of a training session by the *Fragment Map Management* component and is subsequently utilized for restoring the model by the *Fragment Restore* component and/or for all future checkpoints by the *Fragment Creation* component. To create the *fragment map*, we calculate the optimal fragment size to balance checkpointing efficiency and runtime overhead (lines 4–6) first. The fragment size threshold is computed to ensure that the snapshot time for a single fragment does not exceed the iteration time, thereby avoiding training stalls. Specifically, given the measured effective PCIe bandwidth  $bw$  between GPU and CPU and the average iteration time  $T_{compute}^{(i)}$ , the fragment size threshold  $F_{threshold}$  is:

$$F_{threshold} = \frac{T_{compute}^{(i)} \times bw}{2},$$

where the factor of two accounts for copying both model state and optimizer state. If the total model and optimizer state size is  $\alpha_s$ , the number of fragments is  $m = \lceil \alpha_s / F_{threshold} \rceil$ . In practice, layer boundaries constrain exact equal partitioning, so some fragments may be slightly larger or smaller than  $F_{threshold}$ . Once determined, we use a layer size-based or block size-based fragment creation strategy to divide the model into different fragments for checkpointing.

**Layer Size-based Fragment Creation Strategy:** This strategy focuses on optimal system operations by sequentially assigning model layers to a fragment until their cumulative size is equal to or less than the optimal fragment size (line 10). It minimizes stalls for each fragment during frequent checkpointing and optimizes convergence time by preserving recent progress for restoration in case of failure. Our experiments (Section 5) show that this approach is effective for most models, providing simplicity and broad applicability. If the whole model can be divided into  $m$  equal-sized

**ALGORITHM 1:** Create Fragment Map

---

**Require:**  $M$  - DNN model state  
**Require:**  $O$  - Optimizer state  
**Require:**  $b$  - Use DNN blocks as fragments  
**Ensure:**  $F$  - Fragment dictionary for checkpointing

```

1: function CREATE_FRAGMENT_MAP( $M, O, b$ )
2:    $OMap = create\_optim\_state\_map(M, O)$ 
3:    $F = OrderedDict()$ 
4:    $BW = get\_bandwidth(M)$ 
5:    $F_{optimal} = compute\_fragment\_size(M, O, BW)$ 
6:    $M\_copy = copy(M)$ 
7:   if  $b$  then:
8:      $M\_copy = consolidate\_layers(M, O, OMap)$ 
9:   end if
10:   $Bdict = layers2blocks(M\_copy, O, OMap)$ 
11:  for  $Fid, param$  in  $blocks2F(F_{optimal}, Bdict)$  do
12:     $F[Fid] = param['tensors']$ 
13:  end for
14: end function

```

---

fragments, then the total storage cost  $\pi$  for  $c$  checkpoints would be

$$\pi = \frac{1}{m} \sum_{l=1}^c \alpha_s. \quad (18)$$

**Block Size-based Fragment Creation Strategy:** This strategy optimizes operations based on the architecture of ML models. For example, advanced models like ResNet50 and ResNet152 feature residual block-based architectures, where each residual block comprises multiple layers and behaves like a subnetwork with unique characteristics [15]. It is preferable to preserve the internal connections of parameters within these blocks, particularly for complex model architectures, as this is vital for rapid convergence during restoration after failures. We achieve this by first identifying blocks (lines 7–9) and then grouping them into fragments based on the entire block size rather than individual layer sizes (lines 10–12). If fragments of a model are checkpointed in order and at equal frequency, then the total storage cost  $\pi$  for this strategy would be as same as Equation (18), where the amount of space saved in total would be a factor of  $1/m$ .

While the layer size-based strategy prioritizes simplicity and versatility, the block size-based strategy, despite its potential for larger fragments, is essential for preserving the model accuracy of complex models upon restore. Each strategy is selected based on the model’s architectural requirements and performance characteristics. By providing users with the option to carefully select and apply these strategies, we optimize checkpointing procedures to minimize training disruptions and enhance the system’s resilience to failures, thereby supporting robust and efficient deep learning workflows.

**4.1.2 Managing Optimizer State in Fragment.** The optimizer state, denoted  $O$  in Algorithm 1, keeps track of necessary information (e.g., moving average of gradients and their squares) required for updating the trainable model parameters (i.e., weights and biases). Depending on the optimizer being used and precision of the parameters, the optimizer state can account for approximately half of memory used during training [39]. In certain training instances, e.g., when using Adam optimizer, restoring the optimizer state correctly during failure recovery is crucial to not incur additional delays to reach convergence [43]. In other instances, e.g., when using SGD without momentum,

restoring the optimizer state is not mandatory. This can have a significant impact on checkpointing and recovery time, given the large size of the optimizer state. Interestingly, existing checkpointing solutions do not leverage this information. Rather, it is the norm to checkpoint the optimizer state at all times, regardless of its impact on failure recovery. Fragment provides multiple implementations of the *Fragment Map Creation* component to checkpoint with and without optimizer state. However, for the sake of fair comparison with existing solutions, we use the implementation with optimizer state checkpointing in our evaluations.

**4.1.3 Tracking and Restoring Latest Fragments.** To correctly restore the latest fragments during failure recovery, we need to recover the fragment map and the location of the latest checkpoint files. This task is executed by the *Fragment Status Management* component by persisting the *manifest*, which keeps track of the latest checkpoint files for each fragment, on disk. During failure recovery, the *Fragment Restore* component executes or coordinates the following tasks. First, the *fragment map* is recovered. Then, each fragment is restored by consulting the *Fragment Status Management* component. Next, *data iterator state* and hyperparameters (e.g., learning rate and weight decay) are restored from the latest fragment. This step is particularly important, since fragments are restored in number order (i.e., fragment #1, fragment #2 ...), instead of the order in which fragments were checkpointed. Lastly, we update the *optimizer direct map* and *optimizer reverse map*. To ensure that training can recover successfully at any point in time, the first checkpoint during a training session is persisted as a full checkpoint.

## 4.2 Checkpointing Workflow

Although dividing full checkpoint into fragments reduces individual checkpoint size, it may not result in a significant checkpointing frequency increase using the current checkpointing workflow. In this subsection, we propose three modifications for the checkpointing workflow using Fragment.

**4.2.1 Opportunistic Checkpointing.** It is crucial to determine the frequency at which each fragment should be checkpointed during the training process. The straightforward approach to checkpointing involves establishing a fixed frequency, where checkpoints are captured at regular intervals, regardless of system utilization. However, more sophisticated checkpointing methods, such as the one proposed by [25], utilize periodic instrumentation to dynamically adapt the frequency based on available resources and system dynamics. Instead of adhering to a predefined schedule, these methods continuously monitor system performance metrics such as CPU utilization, memory usage, and I/O latency. By analyzing these metrics, the checkpointing system can dynamically adjust the frequency of checkpoint operations to optimize resource utilization and minimize overhead.

However, in our experiments, we observed that the dynamic frequency strategy cannot adapt quickly enough to changing system resource utilization, leading to a reduced total number of checkpoints. For example, persist operation times can intermittently spike due to ongoing underlying storage media operations (e.g., SSD garbage collection) [14]. In our experiments, we found that persist operations for VGG16 can increase by up to 17×, compared to median persist time, while using an SSD as persistent storage. Checkpointing frequency is reduced in response to such events, but it does not revert to its prior value immediately after the write latency returns to its usual level. This reduction leads to a decrease in the total number of checkpoints in a training session.

Therefore, to align with the goal of frequent checkpointing while ensuring efficient resource utilization, Fragment manages checkpointing independently for each fragment based on available opportunities, such as the completion of previous checkpoint operations. For example, at the end of each iteration, if the previous checkpoint operation for a fragment has completed, a new checkpoint is triggered for that fragment. If a checkpoint operation is still ongoing, Fragment skips triggering a new checkpoint in that iteration and attempts to trigger a checkpoint in the next iteration. This

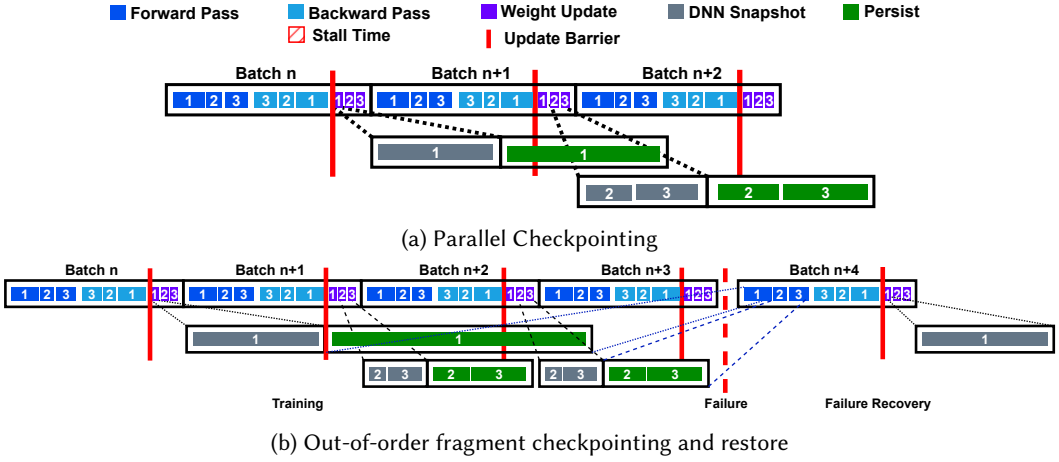


Fig. 6. (a) Parallel checkpointing increases checkpoint frequency. (b) Out-of-order checkpointing increase checkpoint frequency and ensures restoring the latest fragments [1,2,3 denote model layers].

opportunistic checkpointing strategy helps Fragment adapt to changing system behavior and can increase the total number of checkpoints, thereby reducing retraining time in the event of failure.

**4.2.2 Fragment-Based Parallel Checkpointing.** Fragment significantly enhances the capability for fine-grained parallel checkpointing. Traditional checkpointing solutions typically manage checkpoint operations through a global status for the entire model and other intermediate parameters [6, 25], which delays the initiation of a new checkpoint until the previous one finishes. In contrast, since each fragment is independent, multiple fragments can be checkpointed concurrently, as shown in Figure 6(a). For example, if a model is divided into two fragments, checkpoint operations on fragment #1 and fragment #2 can proceed independently. That is, a new checkpoint operation on fragment #1 can occur in parallel with an ongoing checkpoint operation on fragment #2, unless there is an active ongoing checkpoint operation for the same fragment. In this design, when one fragment is persisting the snapshot, the other fragment can initiate its own snapshot operation.

**4.2.3 Out-of-Order Checkpoint and Restore.** A straight-forward round-robin checkpointing approach maintains strict order, processing fragments sequentially (e.g., fragment #1 and then #2), even when checkpointed in parallel. However, if fragments vary significantly in size, this method is inefficient. Larger fragments may delay checkpoints for smaller ones, as the next checkpoint is not triggered until the larger fragment completes. This can reduce the overall checkpoint frequency.

To address this challenge, Fragment introduces out-of-order checkpointing and restore, allowing fragments to be checkpointed independently, improving flexibility and efficiency. For example, checkpoint version 2 of fragment #2 can be triggered while version 1 of fragment #1 is still being checkpointed. Additionally, the entire model can be restored from these out-of-order checkpoints (see Figure 6(b)). This approach is inspired by similar strategies like SCAR [30], which prioritizes checkpointing specific shards in distributed systems based on changes since the last checkpoint. However, unlike SCAR, which incurs significant memory and computational overhead to track changes, our method avoids these costs, leveraging its opportunistic checkpointing feature to offer a more efficient solution.

The above three strategies work together as a *proactive checkpointing* scheme to improve checkpointing frequency and reduce recovery time due to failures.

Table 2. DNN Models Evaluated

| ID | Name       | Size(MB) | # of Layers |
|----|------------|----------|-------------|
| M1 | ResNet18   | 87       | 43          |
| M2 | ResNet50   | 183      | 70          |
| M3 | Resnet152  | 448      | 206         |
| M4 | VGG-16     | 1,031    | 25          |
| M5 | VGG-19     | 1,072    | 27          |
| M6 | DistilBERT | 767      | 6           |

## 5 Evaluation

In our evaluation, we aim to answer the following questions:

- How well does Fragment recover from failure without compromising accuracy? (Section 5.2)
- How does Fragment’s performance compare with existing checkpointing strategies in terms of overall training time? (Section 5.2)
- How does Fragment provide equivalent or faster failure recovery times, relative to existing techniques? (Section 5.3)
- Can Fragment help increase checkpoint frequency? (Section 5.4)
- How well does Fragment performance scale with larger models? (Section 5.4)

### 5.1 Experimental Setup

We evaluate Fragment on the servers of the Chameleon Cloud [19] and on our in-house server. The Chameleon Cloud servers are equipped with an Intel Xeon processor, 2× NVIDIA P100 GPU, 128GB DDR4 DRAM, and 250 GB HDD. This server was used to run prolonged accuracy experiments. Our in-house server is equipped with an Intel(R) Xeon(R) Gold 5218R processor, 2× NVIDIA Quadro RTX 5000 GPU, 128 GB DDR4 RAM, 4 TB SAS HDD, and 512 GB Intel SSD Pro 760p. The in-house server is used to measure checkpoint frequency and overheads because of its availability to utilize different storage devices for storing checkpoints and datasets. We evaluate Fragment using PyTorch’s DDP training.

For a fair comparison with full checkpointing methods, we fragment the optimizer state. However, the *optimizer state dict* implementation in PyTorch gives rise to some engineering challenges in implementing Fragment. The framework uses two packages to manage model layers and optimizer state that operate independently without a shared mapping system for their respective *state dict*. In addition, parameter IDs in the *optimizer state dict* can change during each failure recovery. This makes checkpointing and restoring the associated parameters in the *optimizer state dict* for each fragment challenging. We address this issue by identifying the relationship of all trainable parameters between the *model state dict* and the *optimizer state dict*, and use this to create two key mappings: *optimizer direct map* and *optimizer reverse map*. The *direct map* is used for creating fragments, while the *reverse map* is used during restoration. These mappings are created once at the beginning of training and updated whenever training restarts.

We evaluate Fragment using five different image classification models on top of the Tiny ImageNet dataset [21] and one Transformer-based language model for binary sequence classification using Stanford Sentiment Treebank v2 dataset [35] (Table 2).

**Compared Strategies.** We compare Fragment with other GPU-based and CPU-based checkpointing strategies. All the strategies are implemented in a mainstream and representative training framework using PyTorch [29]. We use CheckFreq [25] as a reference checkpointing system since it incorporates GPU and CPU snapshots and pipelines the checkpointing process with foreground training operations. Subsequent works such as Check-N-Run [11], LightCheck [6], and GPM [28] are based on the same principles as CheckFreq.

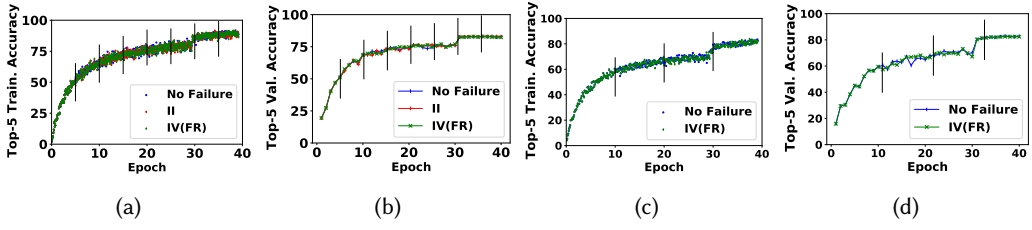


Fig. 7. ResNet18 training (SGD) with Tiny Imagenet —(a) Top-5 Training accuracy and (b) Top-5 Test Accuracy. ResNet50 (ADAM)—(c) Top-5 Training accuracy and (d) Top-5 Test Accuracy. [Vertical black lines denote failure recovery in II and IV(FR)].

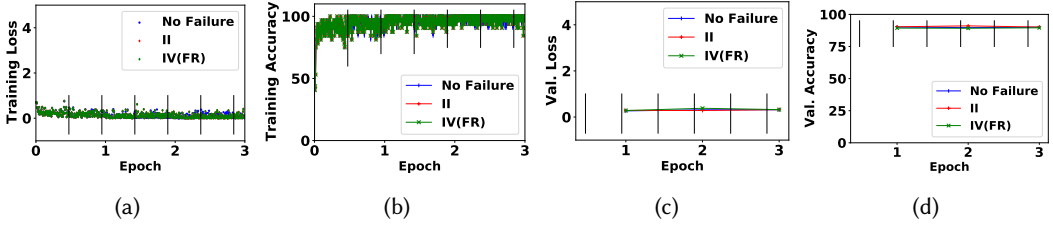


Fig. 8. DistilBERT fine-tuning with SST-2 (AdamW, batch size=32). (a) Training loss, (b) Training accuracy, (c) Validation loss, and (d) Validation accuracy [Vertical black lines denote failure recovery in IV(FR)].

The **No Chpk.** strategy refers to training without saving checkpoints and serves as the baseline to evaluate training time overheads. **I** is a full checkpoint with synchronous GPU snapshot as done in CheckFreq [25]. **II** is a full checkpoint with asynchronous CPU snapshot. **III (FR)** is a layer size-based fragment checkpoint and **IV (FR)** is a block size-based fragment checkpoint.

## 5.2 Model Convergence

We evaluate the accuracy of different models trained using Fragment’s checkpointing and failure recovery scheme. First, we train the models in Table 2 for 40 epochs without any interruption or failure and measure Top-1 and Top-5 training and validation accuracy at regular intervals. Next, we train the models using configuration II (i.e., asynchronous CPU snapshot-based full checkpoints) with a failure happening every thousand iterations, as this is the fastest full checkpoint strategy in our setup. Last, we train the models using Fragment (configuration IV), which also uses asynchronous CPU snapshots. For different models, a training job failed between 3 and 30 times for both configuration II and IV (Fragment). The exact number of failures depends on the batch size or the number of batches in an epoch for a particular model. We set the batch size based on the capacity of the GPU memory. The reason behind inducing so many failures during training is to understand if Fragment can successfully recover from failure at any point in training without any significant impact on accuracy. Regardless of how close or far the model is from convergence, can Fragment recover successfully?

We evaluate different DNN models (listed in Table 2) and present our results for ResNet18 and ResNet50 as representative workloads in Figure 7, and for DistilBERT fine-tuned on SST-2 in Figure 8. In our evaluation, Fragment does not negatively impact the model’s accuracy after failure recovery. On average, the immediate accuracy following failure for Fragment is comparable to that of the full checkpoint strategy, and the time required for Fragment to regain the restored accuracy is negligible. We observe a wide range of immediate accuracy after recovery for both Fragment and

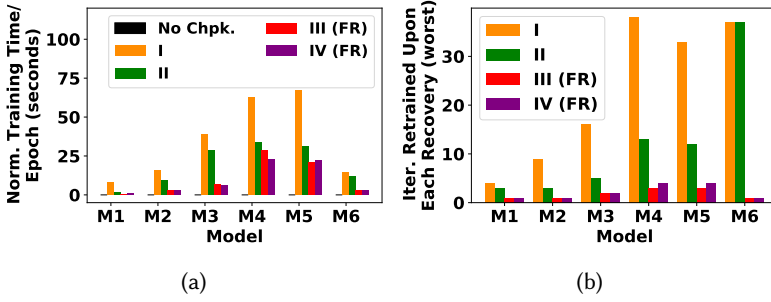


Fig. 9. Fragment (a) reduces training time compared to training with full checkpoint, with same checkpoint frequency, and (b) also reduces number of iterations needed to be retrained upon each failure recovery, when proactive checkpointing is enabled.

full checkpoint methods, which makes direct comparison challenging. To address this, we plot the overall accuracy over time instead of focusing on a single instance immediately after failure. We anticipate that the variability in immediate accuracy may arise from the inability to capture and restore certain internal states during DDP training, which can influence the results. Nonetheless, in the case of both full checkpoint and Fragment, the Top-1 and Top-5 training and test accuracy, as demonstrated in Figure 7, suggest that Fragment can be used for failure recovery without any reservations about negating the training progress. To evaluate Fragment beyond CNN architectures, we fine-tune DistilBERT on the SST-2 binary sentiment classification task using AdamW with mixed-precision training (Figure 8). Based on the observations in [38], we fine-tune DistilBERT for three epochs; we verified that extended training beyond this point drives the training loss to zero but leads to overfitting, and Fragment preserves the loss curve trend throughout. After three epochs, the no-failure baseline achieves 90.02% validation accuracy, CheckFreq achieves 89.45%, and Fragment (Mode IV) achieves 89.91%—within 0.11 percentage points of the no-failure baseline and 0.46 percentage points above CheckFreq. These results confirm that Fragment’s relaxed consistency model preserves model quality on both transformer architectures and relatively simpler datasets, consistent with our CNN findings.

Although it is common practice to measure training stalls only in terms of snapshot stalls, we find this can sometimes be misleading. For example, frequent checkpointing can saturate PCIe bandwidth, leading to longer data loading times, increasing overall training time. However, this may not necessarily result in longer snapshot times. To address this, we evaluate increased stall time by examining the increase in training time per epoch compared to training without checkpointing operations. We analyze Fragment’s training time per epoch using two different fragment creation strategies: layer size-based [III (FR)] and block size-based [IV (FR)] (Section 4.1.1), while keeping the overall checkpoint frequency consistent with the most frequent full checkpoint (i.e., II - asynchronous CPU snapshot in this case). To recall, in the layer size-based strategy, tensors from adjacent layers are grouped based on the optimal Fragment threshold, regardless of the model’s internal grouping strategy. In contrast, block size-based fragments consider both the model’s internal grouping and the optimal fragment threshold.

As shown in Figure 9(a), compared to the fastest full checkpointing configuration (i.e., II - asynchronous CPU snapshot), both Fragment creation strategies reduce training time per epoch. We include the additional time spent to create Fragment maps at the beginning of training in our results. Despite the initial Fragment creation overhead, training time per epoch decreases by 1.3~10.5 seconds with Fragment on average. This time savings will increase linearly as the number of epochs grows. If the model is trained for thousands of epochs, which is common for large

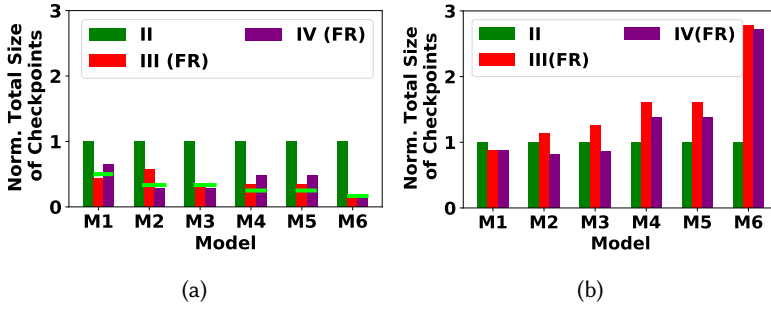


Fig. 10. Fragment (a) decreases the total amount of data (i.e., model state) checkpointed when frequency is matched with its counterpart using asynchronous CPU snapshot [horizontal neon green line shows expected storage cost for Fragment based on Equation (18), when all fragments were equal in size and (b) can checkpoint more data (i.e., model state), when proactive checkpointing is enabled.

production datasets, the total time saved can be a factor of hours and days, leading to significant cost and power savings. In addition, the reduced amount of data being persisted on disk, as shown in Figure 10(a), will reduce PCIe bandwidth saturation, improve capacity utilization, and increase the lifetime of the storage device.<sup>3</sup>

### 5.3 Recovery Time

Recovery time after failure is dependent upon the amount of lost work. That is, the number of iterations required for retraining, which is determined by the state of the data iterator saved in the latest checkpoint. Fragment can reduce the amount of retraining required upon failure recovery if opportunistic checkpointing is enabled (Section 4.2.1). Figure 9(b) shows the number of iterations retrained in the worst case following each failure recovery across different checkpointing mechanisms. In our experiments, Fragment also recovers model accuracy immediately upon restoration (Section 5.2). As observed in Section 5.2, Fragment does not require additional iterations to reach convergence, other than re-training the lost iterations. Thus, Fragment significantly reduces the amount of retraining required upon each failure recovery compared to full checkpointing mechanisms.

In addition to the re-training, all failure recovery operations incur some delay to restore the model from checkpoints. The restore operation involves copying the model state, optimizer state and other hyperparameters from checkpoint files to GPU memory. Fragment copies the same amount of data from storage to GPU memory, compared to a full checkpoint. Hence, it does not incur additional delay for restore operations. In our setup, the restore time is dominated by the sequential read bandwidth of the storage device (Intel SSD Pro 760p), ranging from under a second for ResNet18 (~87 MB) to approximately 1–3 seconds for VGG-19 (~1072 MB). Fragment’s additional overhead—reconstructing the fragment map and consulting the manifest—is negligible; for DistilBERT (M6), the fragment map creation takes only ~5 ms.

### 5.4 Checkpoint Frequency and Storage Cost

Next, we measure the maximum number of checkpoints completed by Fragment when proactive checkpointing is enabled. We start our analysis by calculating the theoretical maximum number of checkpoints achievable per epoch for full checkpoints in our setup using Equation (16). When

<sup>3</sup>We compare Fragment with strategy II in Figure 10(a) and 10(b), because strategy II has the highest checkpointing frequency for full checkpoints in our setup.

Table 3. Checkpoint Count Per Epoch [ $\dagger$  Theoretical Maximum Checkpoints/Epoch Based on Equation (16), in Our Experimental Setup;  $\ddagger$  Theoretical Maximum Checkpoints/Epoch Based on Equation (16) and Checkpointing Workflow Described in Section 4.2]

| ID | Batches | Full Chpk. $\dagger$ | Fragment $\ddagger$ | Full Chpk. [I] | Full Chpk. [II] | Fragment [III (FR)] | Fragment [IV (FR)] |
|----|---------|----------------------|---------------------|----------------|-----------------|---------------------|--------------------|
| M1 | 196     | 196                  | 196                 | 40             | 75              | <b>196</b>          | 192                |
| M2 | 391     | 195                  | 391                 | 46             | 172             | <b>391</b>          | <b>391</b>         |
| M3 | 782     | 195                  | 782                 | 50             | 184             | <b>694</b>          | 598                |
| M4 | 782     | 97                   | 782                 | 21             | 63              | <b>304</b>          | 222                |
| M5 | 782     | 86                   | 782                 | 24             | 69              | <b>332</b>          | 220                |
| M6 | 264     | 66                   | 264                 | 24             | 24              | <b>232</b>          | 226                |

Maximum checkpoint count per epoch achieved for each model is highlighted in bold.

proactive checkpointing is enabled, Fragment’s checkpoint frequency should be 1 for all models in our experimental setup (i.e., Fragment is expected to checkpoint in every iteration). As seen in Table 3, Fragment’s theoretical maximum is  $\leq$  full checkpoints for all models. Interestingly, the actual number of checkpoints achieved by full checkpoint setup **I** and **II**, is always lower than the theoretical maximum for full checkpoints. When the size variance of different fragments is relatively low, Fragment can checkpoint with a frequency of 1 or close to 1, matching or reaching close to the theoretical maximum checkpoint frequency, as observed for models M1 - M3. When high layer size variance leads to high fragment size variance, Fragment struggles to achieve the theoretical checkpoint frequency (see models M4 and M5). Despite this, Fragment still completes a significantly larger number of checkpoints per epoch than full checkpoints.

It might seem that directly comparing Fragment’s checkpoint count with existing solutions is not fair since Fragment only checkpoints a portion of the whole model at a time. However, each checkpoint of Fragment captures the updated state of the training and is sufficient to restore the training from that most recent checkpoint. Our restore accuracy comparisons in Section 5.2, also shows the same. This makes the comparison of the frequency of full checkpoints with Fragment checkpoints fair, as both are equally capable of restoring the most recent state.

**On Checkpoint Frequency in Evaluation.** Our evaluation configures all strategies to checkpoint at their maximum achievable frequency. Fragment is designed for failure-prone environments such as spot instances [25] and large-scale training clusters [40], where high-frequency checkpointing is desirable to minimize lost work. Comparing at maximum frequency exposes the fundamental performance differences between strategies under peak stress.

This evaluation choice is grounded in classical fault-tolerance theory. The optimal checkpoint interval is given by the well-established Young-Daly formula [8, 42]:

$$\tau_{opt} = \sqrt{2\delta M},$$

where  $\delta$  is the checkpoint cost and  $M$  is the mean time between failures (MTBF). The minimum fraction of training time wasted due to checkpointing and rework is  $W_{min} \approx \sqrt{2\delta/M}$ . This formula reveals that reducing checkpoint cost  $\delta$  has a compounding effect: it simultaneously enables more frequent checkpointing (smaller  $\tau_{opt}$ ) and reduces total training overhead (smaller  $W_{min}$ ).

To illustrate, consider a large-scale GPU cluster with an MTBF of  $M = 3$  hours, consistent with reported failure rates in production training environments [10, 40]. For a full checkpoint with  $\delta_{full} = 20$  seconds, the Young-Daly optimal interval is  $\tau_{opt} \approx 85$  seconds, and the minimum overhead is  $W_{min} \approx 33\%$ . If Fragment reduces the per-checkpoint cost to  $\delta_{frag} = \delta_{full}/m$  by checkpointing

one of  $m$  fragments, the optimal interval decreases to  $\tau_{opt} = \sqrt{2\delta_{full}M/m}$ . For  $m = 3$  fragments,  $\tau_{opt} \approx 49$  seconds and  $W_{min} \approx 19\%$ ; for  $m = 5$  fragments,  $\tau_{opt} \approx 38$  seconds and  $W_{min} \approx 15\%$ . Thus, Fragment's higher checkpoint frequency is not an unrealistic evaluation choice; it is the *optimal operating point* prescribed by classical fault-tolerance theory when checkpoint cost is reduced.

Moreover, at lower checkpoint frequencies, the absolute overhead of all strategies decreases, but Fragment's *relative* advantage is preserved: each fragment checkpoint is  $1/m$  the size of a full checkpoint regardless of frequency, and the per-checkpoint stall time reduction holds at any frequency. The storage cost relationship in Equation (18), where Fragment reduces total data written by a factor of  $1/m$ , is independent of how often checkpoints are triggered. Thus, while the absolute amount of total data written increases at lower frequencies, the proportional benefit of Fragment is preserved.

Further, we also compare the total amount of data (i.e., model state) checkpointed using different methods. When Fragment matches the checkpoint frequency for existing solutions, its storage cost reduces (see Figure 10(a)). Due to some variance in the size of different fragments, the actual storage cost may be lower or higher than the expected storage cost when all fragments are of the same size. As shown in Figure 10(b), Fragment checkpoints  $1.6\times$  to  $6\times$  more data compared to synchronous GPU snapshot-based checkpoints (I). When compared to asynchronous CPU snapshot-based checkpoints, Fragment can checkpoint more data in relatively larger models such as ResNet152 (M3) and VGG16 (M4) when proactive checkpointing is turned on. This means that Fragment can be an efficient checkpointing solution if highly frequent checkpointing is desired for large models.

## 6 Discussions

We discuss potential research directions for Fragment here.

**Dynamic Fragment Creation.** In this work, we use a static fragment map that groups trainable parameters into two or more fragments, which is utilized to create fragments throughout the training session. To validate the concept of checkpoints with Fragment, we take a conservative approach by setting the fragment size threshold used to create the static fragment map to create and restore the training state. However, fragments can also be created aggressively on an ad-hoc basis, depending on the current training state and system utilization. Training time for different iterations tends to vary. System utilization can also fluctuate throughout the training session, allowing room for other operations, such as aggressive GPU-CPU data copying. This approach dynamically groups trainable parameters into different fragments based on the current training state and system utilization. For example, we can issue ad-hoc GPU-CPU copy operations for as many tensors as possible before the next weight update operation is triggered. We can maintain a parameter-level map to keep track of the corresponding checkpoint files, instead of relying on the current fragment-level map.

**Importance-based Fragment Creation.** A number of recent works have observed that many models are over-parameterized [5] and that not all parameters are equally important for a model [43]. Reinitialization of certain parameters does not significantly impact model accuracy for many models [4, 43]. If an importance matrix for different parameters is available beforehand, it can be used to create fragments optimally. It might also seem tempting to compute the importance of a layer based on how much the weights or gradients have changed since the last checkpoint, similar to differential checkpointing. However, the differential checkpointing approach tends to have high computational overhead and is generally less desirable.

**Applicability to Other ML Workloads.** Fragment's core insight—iterative-convergent training algorithms tolerate bounded perturbations—extends beyond CNN and Transformer-based models and training setups we evaluated. In data-parallel training, Fragment applies directly since each

rank holds a complete model replica. For model-parallel and pipeline-parallel training [17], each rank already holds a subset of layers, and Fragment can further partition each rank's local state. For tensor-parallel training, fragment boundaries would need to align with the tensor sharding strategy, which we leave to future work. For classical convex models (e.g., an SVMs), perturbations from stale fragments would be corrected even faster due to the unique global optimum, though the practical benefit is limited since such checkpoints are typically small.

**Impact of GPU Execution Parameters.** User-controlled GPU execution parameters, such as work group sizes, thread block configurations, and cuDNN algorithm selection, can affect numerical precision due to non-deterministic floating-point reduction orders [45]. However, these effects are orthogonal to Fragment—they affect all checkpointing strategies equally, including full checkpoints. Fragment's convergence analysis Equation (5)–(8) accounts for perturbations in model parameters regardless of their source. In our experiments, we use standard PyTorch and cuDNN backends with default configurations, which is representative of typical DNN training workflows.

**Optimality and Fragment Size Trade-offs.** Fragment's current heuristic for determining fragment size targets stall elimination as its primary objective, selecting the largest fragment size that avoids training stalls given the system's bandwidth constraints. This heuristic is not provably optimal in a global sense—an optimal solution would jointly minimize the weighted sum of stall time, recovery time, and storage cost, which depends on the (unknown) failure rate and failure patterns. Smaller fragments reduce per-checkpoint stall time and enable higher checkpoint frequency, which decreases the expected retraining time upon failure. However, smaller fragments increase the total number of fragments  $m$ , which increases the staleness bound  $\|\delta^{(k)}\| \leq s\gamma\sqrt{M}$  Equation (6) when the checkpoint cycle length  $s$  grows proportionally. They also modestly increase metadata management overhead (less than 10 MB in our experiments). Larger fragments reduce  $m$  and the number of checkpoint operations per cycle but increase per-checkpoint stall time. We observe in our experiments that the stall-minimizing heuristic performs well across all evaluated models. Future work could explore adaptive fragment sizing that adjusts  $F_{threshold}$  based on runtime profiling of actual stall times and failure rates.

## 7 Conclusion

This article introduces Fragment, a novel checkpointing and recovery approach that relaxes model consistency requirements for fault tolerance and captures snapshots of different model layers at various training stages. By dividing full checkpoints into multiple fragments, Fragment improves efficiency. We propose methods for creating fragments, managing the optimizer state, and a new restore process using the latest checkpointed data iterator state to resume training with minimal accuracy loss. Additionally, Fragment supports parallel checkpointing and out-of-order checkpoint/restore, further enhancing flexibility and efficiency. This approach significantly mitigates the vulnerability of costly failure recovery in training large DNN models. Ultimately, Fragment sets a new standard for optimizing checkpointing processes in deep learning, facilitating more resilient and efficient training workflows. In future, we want to explore the ad-hoc fragment creation based on instantaneous system bottlenecks and the importance of different layers on model performance. We also want to explore the adaptability of this approach beyond DNN failure recovery.

## References

- [1] 2024. Getting Started with Distributed Checkpoint (DCP). (2024). Retrieved from [https://pytorch.org/tutorials/recipes/distributed\\_checkpoint\\_recipe.html](https://pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html)
- [2] 2024. Megatron Team. dist checkpointing package. (2024). Retrieved from [https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/core/dist\\_checkpointing.html](https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/core/dist_checkpointing.html)
- [3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX*

- Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. Retrieved from <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>
- [4] Ibrahim Alabdulmohsin, Hartmut Maennel, and Daniel Keysers. 2021. The impact of reinitialization on generalization in convolutional neural networks. arXiv:2109.00267. Retrieved from <https://arxiv.org/abs/2109.00267>
  - [5] Yasaman Bahri, Quanquan Gu, Amin Karbasi, and Hanie Sedghi. 2021. Over-parameterization: Pitfalls and Opportunities. Retrieved October 29, 2024 from <https://icml.cc/virtual/2021/workshop/8357>
  - [6] Menglei Chen, Yu Hua, Rong Bai, and Jianming Huang. 2023. A cost-efficient failure-tolerant scheme for distributed dnn training. In *Proceedings of the 2023 IEEE 41st International Conference on Computer Design (ICCD)*. 150–157. DOI: <https://doi.org/10.1109/ICCD58817.2023.00031>
  - [7] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. 2020. On efficient constructions of checkpoints. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.). Vol. 119. PMLR, 1627–1636. Retrieved from <https://proceedings.mlr.press/v119/chen20m.html>
  - [8] John T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems* 22, 3 (2006), 303–312.
  - [9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang et al. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. Burges, L. Bottou, and K. Weinberger (Eds.). Vol. 25. Curran Associates, Inc. Retrieved from [https://proceedings.neurips.cc/paper\\_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf)
  - [10] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 610–621. DOI: <https://doi.org/10.1109/DSN.2014.62>
  - [11] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Mikhail Smelyanskiy, and Murali Annavaram. 2020. Check-N-Run: A checkpointing system for training deep learning recommendation models. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.
  - [12] Swapnil Gandhi and Christos Kozyrakis. 2026. Sparse checkpointing for fast and reliable moe training. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 1395–1415. Retrieved from <https://www.usenix.org/conference/nsdi26/presentation/gandhi>
  - [13] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the 19th European Conference on Computer Systems*. 1110–1125.
  - [14] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 127–144. DOI: <https://doi.org/10.1145/3064176.3064187>
  - [15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
  - [16] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P. Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the Advances in Neural Information Processing Systems 26* (2013).
  - [17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu et al. 2019. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. Curran Associates Inc., Red Hook, NY, USA.
  - [18] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. 2019. DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 159–170.
  - [19] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Collieran, Haryadi S. Gunawi, Cody Hammock et al. 2020. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX Association.
  - [20] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (2009).
  - [21] Ya Le and Xuan S. Yang. 2015. Tiny imagenet visual recognition challenge. Retrieved from <https://api.semanticscholar.org/CorpusID:16664790>
  - [22] Yuanhao Li, Tianyuan Wu, Guancheng Li, Yanjie Song, and Shu Yin. 2024. Portus: Efficient dnn checkpointing to persistent memory with zero-copy. In *Proceedings of the 2024 IEEE 44th International Conference on Distributed Computing Systems*. DOI: <https://doi.org/10.1109/ICDCS60910.2024.00015>
  - [23] Weijie Liu, Shengwei Li, Zhiqian Lai, Keshi Ge, Qiaoling Chen, Peng Sun, Dongsheng Li, and Kai Lu. 2026. AdaCheck: An adaptive checkpointing system for efficient llm training with redundancy utilization. In *Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, Santa Clara, CA, 271–289. Retrieved from <https://www.usenix.org/conference/fast26/presentation/liu-weijie>

- [24] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. PMLR, 1273–1282.
- [25] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, fine-grained dnn checkpointing. In *Proceedings of the USENIX Conference on File and Storage Technologies*.
- [26] Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. 2018. Towards understanding the role of over-parametrization in generalization of neural networks. arXiv:1805.12076. Retrieved from <https://arxiv.org/abs/1805.12076>
- [27] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards scalable asynchronous checkpointing of deep learning models. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. 172–181. DOI : <https://doi.org/10.1109/CCGrid49817.2020.00-76>
- [28] Shweta Pandey, Aditya K. Kamath, and Arkaprava Basu. 2022. GPM: Leveraging persistent memory from a gpu. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 142–156. DOI : <https://doi.org/10.1145/3503222.3507758>
- [29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.
- [30] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. 2019. Fault tolerance in iterative-convergent machine learning. In *Proceedings of the International Conference on Machine Learning*. PMLR, 5220–5230.
- [31] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems 24* (2011).
- [32] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M. Badia. 2020. A study of checkpointing in large scale training of deep neural networks. arXiv:2012.00825. Retrieved from <https://arxiv.org/abs/2012.00825>
- [33] Manoj P. Saha, Ashikee Ghosh, Raju Rangaswami, Yanzhao Wu, and Janki Bhimani. 2025. LATTICE: Efficient in-memory dnn model versioning. In *Proceedings of the 18th ACM International Systems and Storage Conference (SYSTOR'25)*. Association for Computing Machinery, New York, NY, USA, 16–29. DOI : <https://doi.org/10.1145/3757347.3759139>
- [34] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti et al 2022. Using DeepSpeed and megatron to train megatron-turing NLG 530B, A large-scale generative language model. arXiv:2201.11990. Retrieved from <https://arxiv.org/abs/2201.11990>
- [35] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 1631–1642.
- [36] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2002–2011. DOI : <https://doi.org/10.1145/3292500.3330756>
- [37] Borui Wan, Mingji Han, Yiyao Sheng, Zhichao Lai, Mofan Zhang, Junda Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2024. ByteCheckpoint: A unified checkpointing system for llm development. arXiv:2407.20143. Retrieved from <https://arxiv.org/abs/2407.20143>
- [38] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop Blackboxnlp: Analyzing and Interpreting Neural Networks For NLP*. 353–355.
- [39] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. FastPersist: Accelerating Model Checkpointing in Deep Learning. Technical Report MSR-TR-2024-27. Microsoft. Retrieved from <https://www.microsoft.com/en-us/research/publication/fastpersist-accelerating-model-checkpointing-in-deep-learning/>
- [40] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast failure recovery in distributed training with in-memory checkpoints. In *29th Symposium on Operating Systems Principles*. ACM, 364–381. DOI : <https://doi.org/10.1145/3600006.3613145>
- [41] Jinyi Xia, Menghao Zhang, Jiaxun Huang, Yuezheng Liu, Xiaohe Hu, Xudong Liu, and Chunming Hu. 2025. Mnemosyne: Lightweight and fast error recovery for llm training in a just-in-time manner. In *Proceedings of the 9th Asia-Pacific Workshop on Networking (APNET'25)*. Association for Computing Machinery, New York, NY, USA, 157–163. DOI : <https://doi.org/10.1145/3735358.3735372>
- [42] John W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Communications of the ACM* 17, 9 (1974), 530–531.

- [43] Chiyuan Zhang, Samy Bengio, and Yoram Singer. 2022. Are all layers created equal? *The Journal of Machine Learning Research* 23, 1 (2022), 2930–2957.
- [44] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. 2023. SWIFT: Expedited failure recovery for large-scale DNN training. arXiv:2302.06173. Retrieved from <https://arxiv.org/abs/2302.06173>
- [45] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in neural network training: Characterizing the impact of tooling. In *Proceedings of the Machine Learning and Systems* 4 (2022), 316–336. Retrieved from [https://proceedings.mlsys.org/paper\\_files/paper/2022/file/427e0e886ebf87538afdf0badb805b7f-Paper.pdf](https://proceedings.mlsys.org/paper_files/paper/2022/file/427e0e886ebf87538afdf0badb805b7f-Paper.pdf)

Received 26 September 2025; revised 27 April 2026; accepted 10 June 2026