ENGINEERING Information Technology & Electronic Engineering
https://www.academax.com/EITEE; https://www.engineering.org.cn
https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=11344065
E-mail: eng.itee@zju.edu.cn

Engineering Journals
Available on IEEE Xplore®

Research Article
https://doi.org/10.1631/ENG.ITEE.2025.0034

FastCheck : Fast Checkpointing and Recovery for DNN Training via Parallel Transmission and Compression

Yun Teng[1], Dawei Sun[1], Shipeng Hu[2], Zhiyue Li[2], Guangyan Zhang[2]✉, Haidong Tian[3], Rui Chang[3]

[1]*School of Artificial Intelligence, China University of Geosciences Beijing, Beijing 100083, China*
[2]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*
[3]*State Key Laboratory of Mobile Network and Mobile Multimedia Technology, ZTE Corporation, Shenzhen 518057, China*

Abstract: Training large-scale deep neural networks (DNNs) is prone to software and hardware failures, with critical failures often requiring full-machine reboots that substantially prolong training. Existing checkpoint–recovery solutions either cannot tolerate such critical failures or suffer from slow checkpointing and recovery due to constrained input/output (I/O) bandwidth. In this paper, we propose FastCheck, a checkpoint–recovery framework that accelerates checkpointing and recovery through parallel transmission and tailored compression. First, FastCheck partitions checkpoints into shards and leverages multiple nodes for parallel checkpointing and recovery. Second, it further reduces checkpoint size and overhead with delta compression for weights and index compression for momentum. Third, FastCheck applies a lightweight consensus protocol that accurately tracks node health, preventing checkpoint transmission to failed nodes. We implement FastCheck in PyTorch and evaluate it on multiple DNN models against two baselines. Experimental results show that FastCheck reduces checkpointing time by up to 78.42% and recovery time by up to 77.41%, while consistently improving efficiency across different training stages.

Key words: Deep neural network (DNN) models, Critical failures, Parallel transmission, Data compression, Checkpointing and recovery

1 Introduction

In recent years, with the rapid evolution of deep neural networks (DNNs), especially the pretrained large-scale language models (commonly known as LLM), the number of model parameters has continuously increased. For instance, PaLM (Chowdhery et al., 2023) has 560 billion parameters, which is 360 times larger than GPT-2 (Radford et al., 2019). Such an immense number of parameters entails the use of massive machines during DNN training. A large-scale training cluster induces frequent hardware and software failures. For instance, training a trillion-parameter model on the Sunway platform with 100,000 GPUs, hardware or software failures occur once an hour on average. In Microsoft's GPU training clusters, the mean failure duration over a 2-month period reached 45 min (Jeon et al., 2019). The 54-d training of LLaMA 3.1 (Meta AI LLaMA Team, 2024) on 16,000 GPUs encountered 419 failures, with failure occurring every 3 h (Lian et al., 2025).

To achieve quick training resumption instead of restarting from the beginning, checkpointing is universally adopted in DNN training. This involves periodically saving both model weights and optimizer momentum to persistent storage. When failures occur, the checkpoints are read to continue training, which reduces the recovery time. However, some naive checkpointing solutions are ineffective, and high failure frequency leads to substantial recovery overhead and low efficiency; for instance, the OPT-175B model (Zhang et al., 2022) suffered an extra overhead of 178000 GPU h due to training failures. Existing studies indicate that checkpointing in DNN models significantly hinders training progress: especially during failure recovery, it substantially incurs wasted training time, which can slow the process by up to 43% (Maeng et al., 2021). This inefficiency occurs due to the time overhead during recovery and checkpointing. Current solutions struggle to (i) effectively reduce the training cost associated with failures and (ii) reduce recovery overhead. As model sizes grow, the storage space and checkpointing time become substantial burdens. Taking a DNN model with a trillion parameters as an example,

✉ gyzh@tsinghua.edu.cn
Yun Teng, https://orcid.org/0000-0001-5425-5111; Guangyan Zhang, https://orcid.org/0000-0002-3480-5902
CLC number: TP302

Received Sep. 12, 2025; Revision accepted Jan. 23, 2026; Crosschecked

© The Authors 2026. Published by Zhejiang University Press Co., Ltd. This is an open access article distributed under the terms of the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)

each checkpoint needs to store approximately 12 TB of model parameters. The massive size of the checkpoint significantly increases checkpointing time. CheckFreq (Mohan et al., 2021) adaptively adjusts checkpoint frequency and implements asynchronous checkpointing; however, it still performs poorly due to the huge size of the checkpoint and the low bandwidth of persistent storage. Moreover, traditional compression techniques such as GZIP (Deutsch, 1996) are ineffective for checkpoints, due to the significant randomness of the ending mantissa bits (Hu et al., 2020).

Nevertheless, since checkpoints reside in persistent storage, their checkpointing and recovery speeds are constrained. Many existing works (Tang et al., 2017; Cai et al., 2022; Chorey and Sahu, 2024) save checkpoints into memory, which can enhance access performance. An analysis of cluster failures (Hu et al., 2024) reveals that failures can be categorized into normal failures and critical failures. Many hardware failures and software failures require full-machine reboots: we call them critical failures, wherein the checkpoints locally stored in CPU memory are unavailable. Normal failures dominate in frequency (89% of total failures), while critical failures comprise only 11%; but the latter account for more than 82% of total failure recovery overhead. As clusters expand and computational demands escalate, the rising frequency of critical failures will lead to substantial recovery overhead. Therefore, we must pay more attention to mitigating the impact of critical failures. Gemini (Wang et al., 2023) stores checkpoints in peer node memory and remote persistent storage. When a critical failure occurs, it recovers using the checkpoint from another node; if the checkpoints on both nodes are unavailable, it loads the checkpoints from remote persistent storage. While leveraging high-speed memory storage improves checkpoint input/output (I/O) performance, Gemini neither reduces checkpoint size nor utilizes multi node parallelism for storing checkpoints within a single node.

To reduce checkpointing time and substantial recovery overhead caused by critical failures, this paper presents FastCheck, an approach that optimizes checkpoint accessing performance through parallel transmission and tailored compression. FastCheck constructs a hierarchical system of storage to save checkpoints, which includes local memory, other nodes' memories, and remote persistent storage. FastCheck stores checkpoints to local CPU memory and transmits checkpoints to the CPU memory of multiple other nodes in parallel. FastCheck compresses the weights and momentum of checkpoints by delta compression and index compression, respectively, according to their different characteristics. In addition, FastCheck also stores checkpoints to remote persistent storage. Compared to existing solutions, FastCheck reduces checkpointing time and recovery overhead for critical failures.

In summary, this paper makes the following contributions:

- We design FastCheck, a parallel checkpoint transmission framework, to address the impact of critical failures. FastCheck divides checkpoints into uniform shards, which are then transmitted to the CPU memory of multiple nodes in parallel, significantly enhancing checkpointing and recovery performance.
- We implement two compression algorithms to reduce the huge size of checkpoints: delta compression for weights; and index compression for momentum. Furthermore, we optimize these algorithms with delta merging and parallel filling. Our algorithms achieve up to 77.4% data reduction across the checkpoints of multiple DNN models.
- We design a lightweight consensus protocol that ensures consistent tracking of node health across nodes. This protocol avoids transmitting to failed nodes, with small extra overhead.
- We evaluate FastCheck and two baselines on checkpoints of multiple DNN models. The overall checkpointing time is reduced by up to 78.42% compared to baseline methods, and the overall recovery time is reduced by up to 77.41%.

2 Background and motivation

2.1 Background

When training a small model in limited number of machines, failures are rarely encountered due to the relatively small number of model parameters, limited number of hardware components, insufficient computational load, and short runtime duration. In contrast, when training one or more large-scale models on clusters, the massive nature of computation and the machines involved frequently induces transient failures. For instance, during the training of the OPT-175B model across 992 NVIDIA A100 GPUs, 110 failures were identified in 2 months (Zhang et al., 2022). Similar patterns of failures were observed during the training of BLOOM (BLOOM Chronicles, 2022). Some failures need full-machine rebooting, such as operating system (OS) failures or hardware malfunctions, categorized as critical failures. Although critical failures occur less frequently than normal failures, the overhead of critical failures is significantly high.

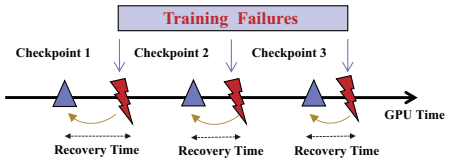

Fig. 1 When meeting failures, the training process loads the latest checkpoint for recovery.

When failures occur during training, we can restart training from the checkpoints, without starting over. As shown in Fig. 1, weights and optimizer momentum are periodically persisted during training. Traditional models typically perform checkpoints at the end of an epoch (MLPerf, 2020). If a failure occurs, the most recent checkpoint is loaded to continue the training process. In large-scale training scenarios with many failures, checkpointing is critical.

A checkpoint primarily contains two types of data: model weights and optimizer momentum. The optimizer momentum includes the first-order and second-order moments for each parameter (Li et al., 2024). As model sizes grow, checkpoints require much more space for storage, which can slow down the training process. In this paper, FastCheck optimizes the

performance of accessing checkpoints, reducing both the overhead of checkpointing and the recovery overhead of loading checkpoints after meeting failures.

2.2 Limitation of existing approaches

Traditional checkpoint solutions store checkpoints on persistent storage media such as SSDs or HDDs. However, the limited bandwidth of these devices affects the checkpointing/recovery speed and slows down the training process. Therefore, it is essential to utilize high-performance memory for storing checkpoints. For instance, some approaches such as Gemini and Diskless (Tang et al., 2017; Cai et al., 2022; Qi et al., 2025) store checkpoints in CPU memory. As shown in Fig. 2, when normal failures occur, we can restart training through these in-memory checkpoints. However, for critical failures, such as bit corruptions (Tiwari et al., 2015; Jeon et al., 2019), OS failures, or network failures (Gill et al., 2011; Tan et al., 2019), the cluster must reboot the failed machine or replace the affected hardware to restart training swiftly. Consequently, the checkpoints stored in the failed machines' CPU memory cannot be accessed.

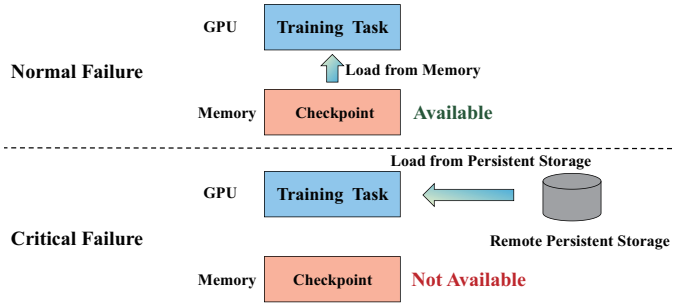

Fig. 2 Difference between normal failure and critical failure. During normal failures, we load checkpoints from memory; during critical failures, we load checkpoints from remote persistent storage because the checkpoint in memory is not available.

Existing research analyzing failures in clusters reveals that although critical failures account for only a small percentage of the total failures, they cause higher recovery costs. To address the recovery overhead of critical failures, Gemini stores checkpoints not only in local memory but also in the memory of other peer nodes in the same cluster, enabling recovery from remote nodes when critical failures occur. However, Gemini stores the checkpoint in a single remote node's memory. This limited bandwidth still results in a significantly long recovery time when handling a large checkpoint.

Furthermore, existing approaches fail to adequately address the issue of massive checkpoint size. Many approaches (Han et al., 2015; Xiao et al., 2023) apply quantization to reduce the storage footprint. For instance, a representation of a float is reduced from 32 bits to 16 bits, thereby halving the space occupancy. However, this introduces uncertainty regarding model accuracy. Existing general compression techniques achieve a very limited reduction on DNNs. As these compression techniques are not well-suited for checkpoints, a tailored compression algorithm should be investigated.

2.3 Observation and motivations

To address the overhead of critical failures, we can store checkpoints on peer nodes' memory. Under the idealized assumption of identical ingress and egress bandwidth across nodes, transmitting to one node or multiple nodes would involve the same aggregate bandwidth. However, in realistic deployments, training nodes often concurrently serve other workloads that consume ingress bandwidth. For example, training–inference integrated systems (e.g., Huawei FusionCube A3000(Huawei, 2024);) simultaneously process large volumes of inference traffic, such as video semantic analysis(Lin et al., 2019), while training models, substantially reducing available ingress bandwidth. In such cases, a node's effective ingress bandwidth becomes lower than its egress bandwidth. We can transmit checkpoints to multiple peer nodes in parallel, which mitigates this imbalance by distributing traffic across multiple destination nodes. We emulate this scenario in our experiments by introducing background ingress traffic that fluctuates between 30% and 60% of the total available bandwidth.

To reduce the size of checkpoints, their characteristics should be considered. Based on the distinct properties of weights and momentum, different compression strategies can be applied. Specifically, since weights exhibit limited changes between consecutive epochs, delta compression can be used to store the incremental changes of weights between adjacent checkpoints. Moreover, momentum is relatively small and often contains repetitive exponent bits, making it suitable for indexing. We encode prefixes with reduced-length indices.

Based on the above analysis and observations, this paper presents FastCheck , an approach to accelerate checkpointing/recovery performance.

3 FastCheck design

3.1 System architecture

The system architecture of FastCheck is illustrated in Fig. 3. It comprises three main components: training nodes, KV storage, and remote persistent storage.

In the FastCheck design, multiple nodes participate in model training. Each training node contains GPUs for model training and periodically generates checkpoints. The generated checkpoints are loaded into the local machine's CPU memory and processed by the partitioning and parallel transmission module. Here, each checkpoint is partitioned into multiple equal-sized shards. The data compression module then compresses each shard individually, applying delta compression to weights and index compression to momentum. The compressed shards are transmitted in parallel to the CPU memory of multiple nodes via Ethernet.

Before transmitting, FastCheck maintains real-time health status records (available/unavailable) for all training nodes, avoiding transmission of checkpoints to failed nodes. Due to the health status with small request size and high frequency, FastCheck uses a KV storage. Remote persistent storage provides a complete checkpoint replica when all in-memory checkpoint replicas become unavailable.

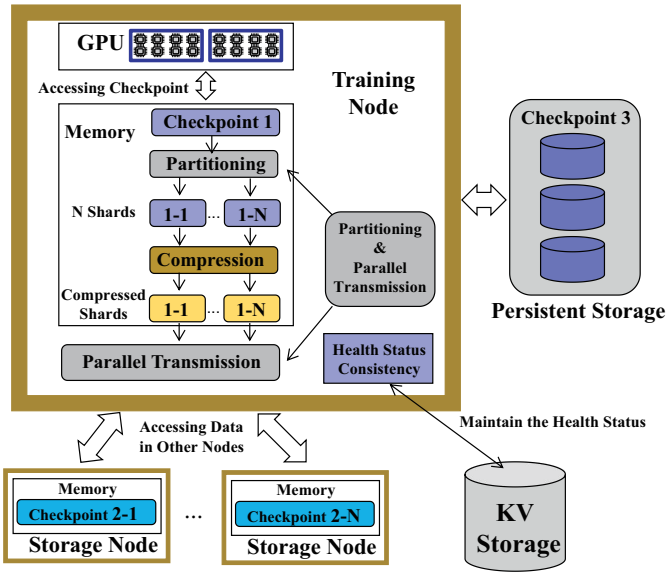

Fig. 3 The overall system architecture.

3.2 Overview of design

Our design contains three modules: partitioning and parallel transmission module, compression module; and health status consistency module. Next, we introduce each module individually.

3.2.1 Partitioning and parallel transmission module

This paper uses a triple-replica placement strategy for checkpoints: local memory, other training nodes' memory, and remote persistent storage. When storing checkpoints to other nodes' memory, to leverage parallelism among multiple nodes, each checkpoint is split into multiple equal-sized shards. FastCheck selects several nodes and writes these checkpoint shards to them in parallel to reduce checkpointing time. This is feasible because it is rare to encounter two and more failures simultaneously(BLOOM Chronicles, 2022; Zhang et al., 2022). When failures occur, FastCheck considers two cases: if it is a normal failure, training nodes read the local checkpoint directly from local memory; if it is a critical failure, training nodes read checkpoint shards in parallel from other nodes, which can accelerate recovery; if the checkpoints in other nodes are lost, FastCheck recovers them from remote persistent storage. Note that each training node's memory stores checkpoints from other training nodes. During recovery, we also asynchronously restore checkpoints of other training nodes from remote persistent storage.

3.2.2 Compression module

To reduce the size of checkpoints, FastCheck applies different lightweight compression algorithms for weights and momentum, which have distinct characteristics. Model weights exhibits continuous variation: the same weight across two consecutive checkpoints has a small change, so FastCheck uses delta compression to record deltas between consecutive versions. During decompression, storage nodes merge the deltas to compute the latest weights in advance, which can reduce transmission overhead during decompression. Since the values of momentum are small, in the binary representation of momentum, the possible types of prefixes are limited; we can use index values with fewer bits to replace the original prefix. This reduces the size of momentum. During the decompression of momentum, the original prefix data are retrieved from the maintained index table (with small overhead).

3.2.3 Health status consistency module

When a node encounters a critical failure, all nodes must be notified promptly to prevent training nodes from writing a checkpoint to the failed node. To maintain each node's health status, we use a KV store to record each nodes' health status. To ensure consistency of training nodes' status, a training node is designated as the master node. The master node maintains a modification bitmap. When each node periodically updates its status, it first sends the update to the master node to change the bitmap, then updates the KV store, and finally modifies the bitmap again. When a node's status needs to be checked, FastCheck accesses the master node to determine whether the node's health status has been updated. With the completion of the status update, the KV store is accessed for fetching the latest status. When the master node fails, FastCheck selects an active node to replace the failed one. FastCheck uses a leader election approach (Ongaro and Ousterhout, 2014) to select the new master node.

4 Detailed design

4.1 Partitioning and parallel accessing

For critical failures, existing approaches fail to consider the parallelism across multiple nodes. For instance, Gemini stores checkpoints in the memory of only one storage node. When the checkpoint size is large, Gemini will suffer high latency on checkpointing/recovery with a single node. Therefore, we propose an approach that parallelizes checkpointing and recovery across multiple nodes' memories to enhance I/O performance.

For clarity, in this paper, nodes generating checkpoints are called the *training nodes*, whereas storage nodes storing checkpoint shards are called *storage nodes*. Specifically, as shown in Fig. 4, each checkpoint consists of two dictionaries: one for weights, and the other for optimizer momentum. Each dictionary contains tensors of varying dimensions and sizes. First, FastCheck determines the number of storage nodes N, which means FastCheck sends a checkpoint to N storage nodes. Given the total checkpoint size S, the size of each shard written to each node is S/N while using multiple nodes to save a checkpoint with parallel I/Os.

Each checkpoint is evenly partitioned into N data shards and transmitted in parallel to the memory of N distinct storage nodes. To improve fault tolerance, we extend FastCheck by generating an additional parity shard via XOR across the N data shards and storing it on a separate node. FastCheck transmits the N data shards and a parity shard in parallel. By introducing fault tolerance within an $N+2$ node group, we can tolerate failure up to two nodes.

Prior studies (Zhang et al., 2023) show that single-node failures dominate real-world scenarios. FastCheck is there-

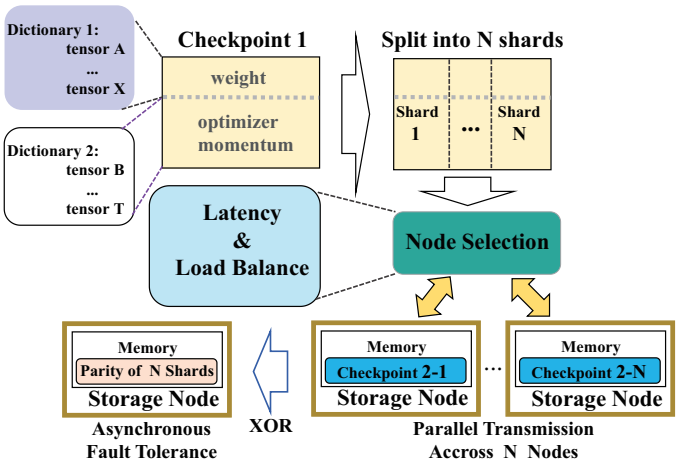

Fig. 4 Partitioning and transmission in parallel.

fore designed to efficiently handle the most common failure cases. Multi-node failures (e.g., three simultaneous failures among $N+2$ nodes) are relatively rare. When they do occur, missing shards are recovered from the underlying persistent storage. Since checkpoints are sharded across multiple nodes, even multiple failures do not render all checkpoints unavailable; only the missing shards need to be fetched. Compared to fully relying on persistent storage, FastCheck significantly reduces recovery I/O overhead.

Before transmitting, FastCheck needs to select nodes. In our work, FastCheck considers both latency and load balancing. FastCheck sends probe requests to all nodes of the cluster each second. FastCheck can then get latencies from these probe requests, and then, it selects N nodes with the lowest N latencies. In the case of network partitions, we exploit the low intra-partition latency by preferentially placing checkpoints within the same partition, thereby minimizing cross-partition traffic. When latency differences are marginal (e.g., due to network or NIC variations), nodes are selected based on load. Specifically, FastCheck targets load balancing by sequentially selecting the nodes with the lowest load. Since checkpoints reside in memory, excessive usage can degrade memory performance. Thus, FastCheck sequentially chooses nodes with the lightest loads.

During critical failures, checkpoints in local memory become unavailable. Recovery involves parallel reads from N nodes storing data shards. The retrieved data are merged to reconstruct the complete checkpoint for continuing training. If a node storing a data shard is also in critical failure, FastCheck recovers the lost data shard by reading $N-1$ data shards and a parity shard.

Due to resource constraints, we did not deploy a large-scale cluster. However, FastCheck does not require increasing the data shard number N with cluster size, as a larger N would introduce additional overhead. Instead, when network latency variation within a cluster is small, a large cluster can be logically partitioned into multiple independent groups, each consisting of $N+2$ nodes storing data and parity shards.

In our experiments, we use five nodes as one such group. Within a group, each node transmits N data shards and a parity shard to the other $N+1$ nodes, while no transmission occurs across groups. Consequently, cluster-scale growth does not adversely affect FastCheck's performance.

4.2 Compression for checkpoints

In FastCheck, as described in Section 4.1, to address the impact of critical failures, in-memory checkpoints are transmitted to multiple storage nodes. As models become larger, checkpoint storage and transmission have become bottlenecks, affecting both underlying storage efficiency and inter-node transmission efficiency. To both reduce the size of data during transmission and accelerate checkpointing, FastCheck applies data compression.

Traditional compression methods are not effective for compressing floats of checkpoints with random ending mantissa bits. Therefore, by analyzing checkpoint data attributes, FastCheck implements distinct compression strategies for different data types.

Each checkpoint consists of two parts: model weights and optimizer momentum. Both are of float type but exhibit distinct characteristics.

4.2.1 Delta compression for weights

For weights, the limited changes between adjacent checkpoints make them suitable for delta compression. During training, weights are updated based on gradients from their previous state, resulting in sparse differences between consecutive checkpoints.

As shown in Fig. 5, assuming that we have the weight matrices of two consecutive checkpoints, we get the delta matrix by calculating weight matrix 1 XOR weight matrix 2.

When two weight matrices xor each other, we xor each weight of the two matrices with the same location. For example, $D_{1,1}$ is the first element of the delta matrix; it is calculated by $W_{1,1}{}^{(1)}$ XOR $W_{1,1}{}^{(2)}$; they are also the first elements of each weight matrix.

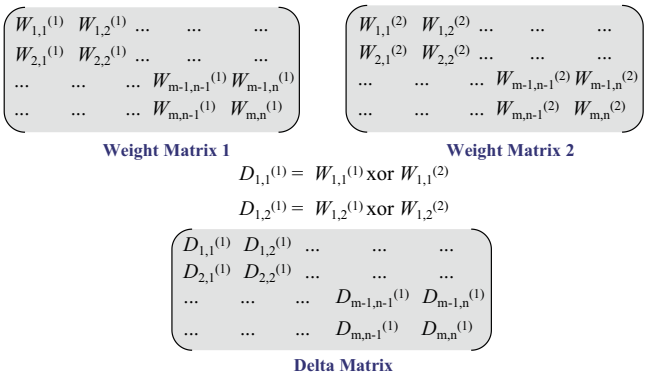

Fig. 5 Calculating the delta matrix from two weight matrices.

We experimentally verify this characteristic by training the deep learning model VGG (Simonyan and Zisserman, 2014) for 30 epochs, during which the model generates a checkpoint for each epoch. We define the reduction ratio R as in Eq. (1), in which S_{origin} is the original size and S_{com} is the size after compression.

$$R = (S_{origin} - S_{com})/S_{origin} \tag{1}$$

We get a delta between consecutive checkpoints, and we calculate the reduction ratio of the original size of weights, as shown in Fig. 6. After the third epoch, the ratio is about 67.6%.

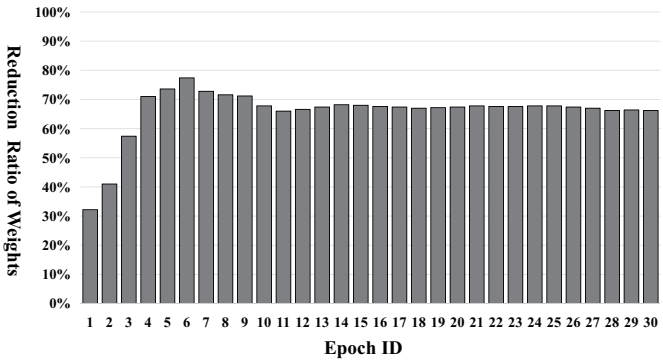

Fig. 6 The data reduction ratio of delta compression over 30 epochs.

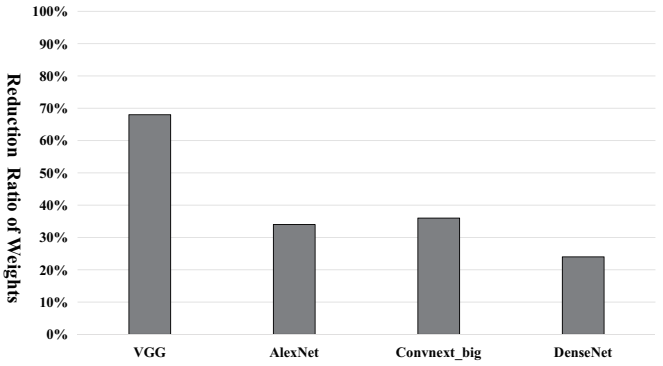

Fig. 7 The data reduction ratio of delta compression under checkpoints of four models.

We also apply delta compression to several other deep learning models, including AlexNet (Krizhevsky et al., 2017), DenseNet (Huang et al., 2017), and Convnext_big (Liu et al., 2022). According to Fig. 7, under the checkpoints of the four DNN models, the delta compression achieved average reduction ratios from 23.7% to 67.6%, demonstrating the significant effectiveness of delta compression.

Fig. 8 details the workflow of delta compression within the system proposed in FastCheck . First, the complete weights of the first checkpoint are stored in local memory, and FastCheck sends the weights to the storage nodes as a base checkpoint. The checkpoints in the figure represent checkpoint shards already split.

Then, FastCheck calculates the delta between the second checkpoint and the first checkpoint. Because there are many leading zeros in $FP32$ or $FP16$, FastCheck records the number of leading zeros and removes them by shifting during the storage of weights, which can reduce the number of bits. Afterward, FastCheck transmits the delta to the storage nodes and calculates the new delta. The second checkpoint's weights are then loaded into memory, and the first checkpoint can be deleted. As checkpoint quantities increase, FastCheck transmits the delta between the weights of the current checkpoint and the weights of the previous checkpoint.

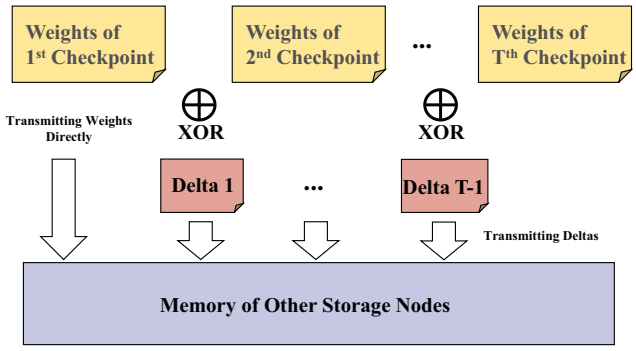

Fig. 8 The process of delta compression.

4.2.2 Merging deltas for recovery

Though the above design effectively reduces the size of checkpoints, it results in multiple independent deltas in the storage nodes, which result in relatively slow recovery speed. When critical failures occur, FastCheck reads the weights of the base checkpoint and all deltas. Furthermore, FastCheck should calculate these deltas to recover the newest weights, incurring significant recovery latency.

To mitigate this overhead, the storage nodes merge local deltas in the background. As shown in Fig. 9, each node receives one base checkpoint and N deltas. FastCheck calculates the base checkpoint XOR delta 1 and gets the weights of the second checkpoint. Through iterative step-by-step computation, FastCheck can get the weights of the final checkpoint. FastCheck reduces both transmission time and computation time compared to the naive method, substantially reducing checkpoint retrieval costs.

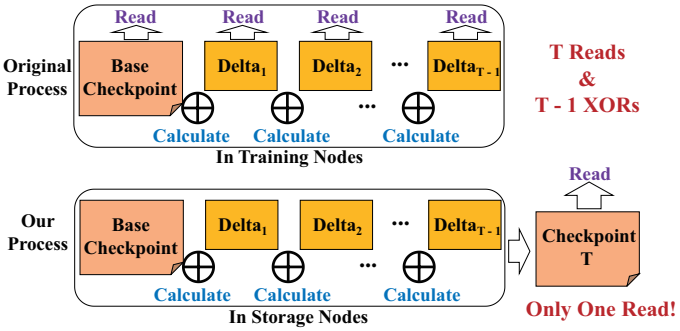

Fig. 9 The optimization of the weight recovery process.

4.2.3 Index compression for momentum

Unlike the weights, the momentum in the checkpoints are the moving average of the first-order and the second-order moments. These are only weakly related to the parameters in the previous checkpoint after updating for hundreds of steps, so using delta compression is not suitable.

However, its data distribution shows distinctive features: momentum values occupy a smaller numerical range, resulting in a highly similar binary form. Hence, the types of prefixes are limited, as shown in Fig. 10. There are four data items; the

first 6 bits are identical. Therefore, FastCheck applies index compression on the optimizer momentum. We can use '0' to replace the first 6 bits of all four items; each float is reduced by 5 bits, as shown in Fig. 10.

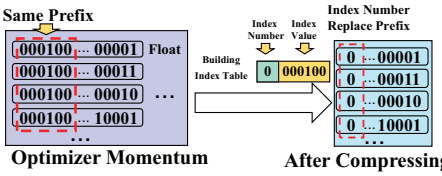

Fig. 10 The process of index compression with a simple example: index compression can reduce each float by 5 bits.

Generally, FastCheck represents the first P bits as an index. To determine the optimal value of P, FastCheck tests common checkpoints, iterating through eight candidate values (ranging from '7' to '14') and selecting the most efficient result. FastCheck maintains an index table containing index numbers and index values, where the index numbers represent distinct P-bit prefix codes. If FastCheck meets a new prefix, FastCheck gives the prefix a distinct code. If the current prefix has been met, it would be ignored. The index values denote the actual P-bit prefixes shared across multiple momentum items. During compression, each momentum's first P bits are replaced by its corresponding index number. Given the limited unique prefixes and high repetition frequency, index compression significantly reduces the size of the momentum. Since index entries exhibit varying occurrence frequencies, to achieve a small size of momentum, we can naturally proceed to optimize this using Huffman coding. FastCheck constructs Huffman codes based on the frequency of these prefixes.

4.2.4 Filling to byte stream in parallel

During compression, in this paper, the floats of checkpoints are $FP32$ or $FP16$; they become variable-length entities after compression (less than 32 or 16 bits). When filling these variable-bit sequences into 8-bit byte streams, serial filling would increase compression time. Therefore, FastCheck implements a parallel approach: As shown in Fig. 11, assuming a maximum parallelism degree P, FastCheck partitions all floats into P groups. Based on each group's total length after compression, FastCheck allocates byte buffers with same size. If the length is not divisible by '8', we allocate a byte for the remaining data less than 8 bits. In the case of Fig. 11, the last float of Group 1 has 26 bits; it can fill to 3 bytes and 2 remaining bits. FastCheck allocates 4 bytes for the last float with a padding of 6 bits. These P groups then perform parallel filling. Although this may cause a minor bit of wastage (typically $< 0.01\%$ of the total), the extra storage overhead is negligible relative to the overall size. Similarly, during decompression, FastCheck applies parallel reconstruction to recover the original data.

4.3 Consistency guarantees

Consistency is a fundamental requirement for storage systems. In this paper, FastCheck primarily addresses two aspects of consistency: replica consistency across multiple checkpoints

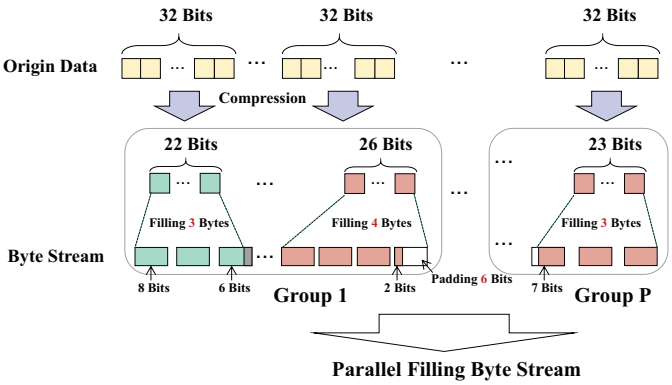

Fig. 11 Parallel filling during compression.

and status consistency of node health. Initially, our guarantee is to maintain consistency between surviving replicas when one becomes unavailable. However, persisting checkpoints are time-consuming and affect foreground training performance.

To mitigate this, FastCheck initiates checkpointing by storing data in local memory first. FastCheck then monitors returns from storage nodes, when checkpointing the second replica. Only after all nodes confirm the successful checkpointing of their respective shards, the checkpointing operation is deemed complete and the training process is allowed to proceed. Finally, the persistence to underlying storage proceeds asynchronously.

While this strategy does not guarantee the strongest consistency, the probability of simultaneous failure for both memory replicas is low. Given the significant training latency caused by waiting for storage persistence, FastCheck strikes a trade-off: Should both memory replicas become unavailable and the persistent storage replica be outdated, training restarts from the previous checkpoint version, incurring additional computational overhead.

For maintaining node health status, FastCheck uses a dedicated KV store. When users query node health status, the system must guarantee status consistency to ensure accurate health reporting.

During the maintenance of health status, one node is designated as the master node. As shown in Fig. 12, the master maintains a bitmap indicating whether each node is currently updating its status. For N training nodes in the whole cluster, the bitmap contains N active bits; a certain bit is set to '1' during status updates and reset to '0' upon completion.

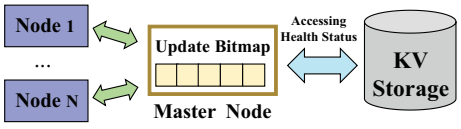

Fig. 12 The overall design of consistency guarantees for health status.

Each node periodically updates its health status. It first sends a request to the master node, the master node changes the corresponding bit of the bitmap to '1' and then updates the status in the KV storage. In this paper, FastCheck uses Memcached as the KV storage. After updating, it changes the corresponding bit to '0'.

In this work, FastCheck transmits checkpoint shards to several storage nodes. Before transmitting, the training node needs to check each storage node's health status. The training node accesses the master node to verify whether the corresponding node's health status is being updated, specifically, to check whether the corresponding bit in the master's bitmap is '0'. If the bit is set to '1', it indicates that the health status is currently updated; the training node waits until completion of the health status update. If the bit is set to '0', it means that the health status can be accessed, and the training node accesses the KV storage to check the latest status.

Fig. 13 illustrates an example of maintaining node health status. The cluster consists of eight nodes, so the bitmap in the master node contains 8 bits. Each bit of the bitmap corresponds to a node. Initially, all bits are set to '0', indicating no ongoing health status modifications.

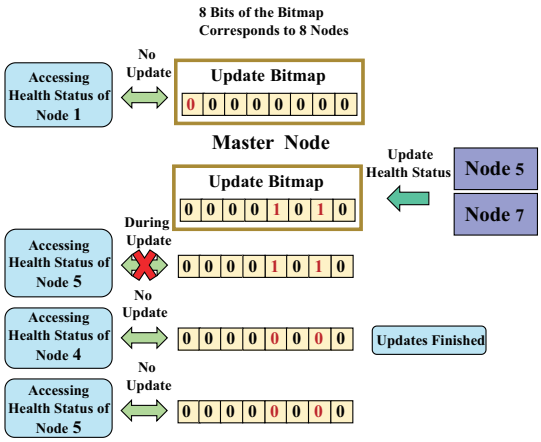

Fig. 13 The example of maintaining health status.

Subsequently, a status request for Node 1 occurs. Since its bit is '0', FastCheck directly accesses its status from the KV storage. Following this, Nodes 5 and 7 update their health status. The master node updates the bitmap, setting the corresponding bits to '1'. During this process, if the user has a health status check for Node 5, as its bit is '1' (update in progress), the request needs to wait to ensure consistency. Concurrently, a check for Node 4 proceeds immediately since its bit remains '0' (no update).

Upon completion of Node 5's update, the master resets Node 5's bit in the bitmap to '0'. This releases the pending Node 5 status request, terminating the wait and accessing the latest health status in KV storage.

5 Performance evaluation

5.1 Setups and implementation

In our experiments, we utilize a GPU server and four CPU servers. The five servers all contain two Intel(R) Xeon(R) Silver 4310 CPUs, one 25G MCX4121A-ACAT NIC, 768 GB memory, and a SATA SSD with 6Gbps bandwidth. The GPU server contains an NVIDIA A100 GPU with a capacity of 40GB, and we use an H3C S6820-56HF switch to connect all five servers. The GPU server trains the model and generates checkpoints; we designate the GPU server as the master node, which maintains health status consistency. The four CPU servers served as the storage nodes for storing data shards and parity shards, and we regard their persistent storage as remote storage.

Table 1 The number of parameters and the checkpoint sizes of the five models used in this paper

Model	Parameters	Checkpoint size
DenseNet	10.2 M	0.15 G
AlexNet	60 M	0.7 G
VGG	139.4 M	1.5 G
ConvNext_big	198 M	2.3 G
GPT-2	1.6 B	18 G

We implement FastCheck in Python and C++ based on PyTorch 2.1.2 (Paszke et al., 2019), with about 3000 lines of code. FastCheck includes three modules: partitioning and parallel transmission, checkpoint compression , and health status consistency. We establish connections between multiple nodes using the TCP protocol based on sockets and used Memcached (Nishtala et al., 2013) to maintain health status consistency.

5.2 Baselines and workloads

In this work, we compare two baselines: CheckFreq (Mohan et al., 2021) and Gemini (Wang et al., 2023). CheckFreq dynamically adjusts the checkpoint frequency but still checkpoints to persistent storage. Gemini addresses the overhead of critical failures by storing each checkpoint in the CPU memory of other single nodes. For normal failures, Gemini reads checkpoints from local CPU memory; for critical failures, Gemini loads checkpoints from other nodes' CPU memory. To enhance the performance of recovery for critical failures, FastCheck stores checkpoints across multiple nodes' CPU memory for parallel checkpointing/recovery and applies a tailored compression method based on the characteristics of weights and momentum.

We evaluate baselines and FastCheck on multiple DNN models, including VGG, DenseNet, AlexNet, ConvNext_big, and GPT2. The parameter size and single checkpoint size of these models are shown in Table 1. FastCheck uses the Adam optimizer for training. The first four models were trained on the ImageNet2012 dataset, while GPT2 used the Wikipedia dataset. We evaluate the checkpointing/recovery performance during critical failures.

5.3 Overall performance

In this subsection, we evaluate the overall performance of our approach against critical failures, by using checkpoints of different models.

5.3.1 Performance of checkpoint accessing

As shown in Fig. 14, we evaluate the overall checkpointing/recovery time of FastCheck and two baselines across the five models mentioned above. For checkpointing performance comparison, Fig. 14 demonstrates that FastCheck consistently outperforms the two baselines across all models. Compared to CheckFreq, FastCheck reduces checkpointing time by up

to 78.42% and, compared to Gemini, by up to 64.23%. This occurs because CheckFreq writes checkpoints to remote persistent storage, resulting in the highest checkpointing latency. While Gemini improves checkpointing speed by storing checkpoints in other nodes' CPU memory, FastCheck further accelerates checkpointing through both data compression and parallel transmission.

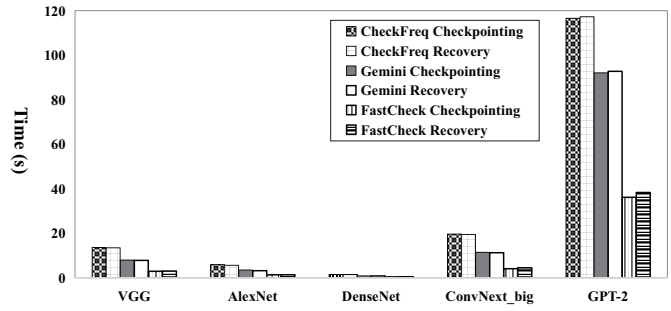

Fig. 14 The checkpointing and recovery performances of FastCheck and the two baselines for the five types of checkpoints.

Under identical conditions, we evaluate recovery performance. Similar to checkpointing, FastCheck achieves the shortest recovery time, which is reduced by up to 77.41% compared to CheckFreq and by 61.11% compared to Gemini. During weight recovery, FastCheck incurs no additional recovery overhead because the total loading size equals the size of one checkpoint, achieved through preemptive delta merging by the storage nodes.

5.3.2 Performance of compression and decompression

A common concern is whether compression/decompression time is significantly less than the time reduced by transmission. To verify this, FastCheck compares the transmission times with and without compression under the ConvNext_big model. As shown in Fig. 15, without compression, the transmission time is 11.694 s. With compression, the total time is the sum of the compression time (0.127 s) and the transmission time (8.958 s), which is 9.085s. With compression, FastCheck saves 22.31% overall checkpointing time.

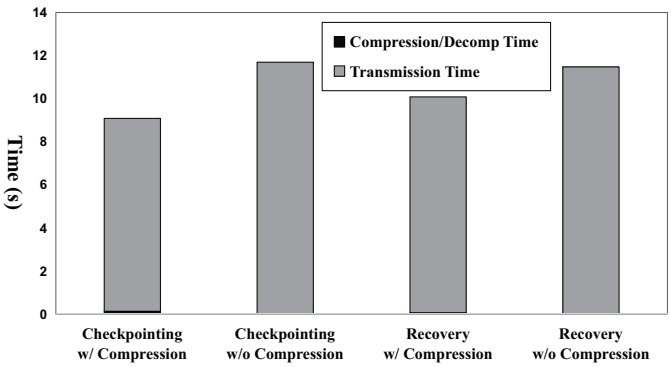

Fig. 15 Time comparison of FastCheck with and without compression.

Similarly, the Fig. 15 also illustrates the time reduction of compression during recovery. Compression reduces recovery time by 12.65%. However, the benefit is smaller during recovery than during checkpointing. This occurs because we compress weights by delta compression, while reading at least one full checkpoint during recovery; decompression provides no reduction for weights, and the advantage comes solely from momentum. During compression, momentum reduction is less than weight reduction, resulting in low overall decompression gains. According to the results, the time of compression or decompression is much less than the transmission time, and our tailored compression yields obvious gains.

5.4 Effects of individual techniques

This paper attributes the improvements in checkpointing/recovery performance to four key techniques: parallel checkpointing/recovery; delta compression for weights; delta decompression with delta merging; and index compression for momentum. We evaluate overall performance by measuring the total checkpointing and recovery times. We implement four versions of FastCheck, each adding a technique on Gemini:

- '+Parallel' adds parallel transmission, which uses employs the parallelism of multiple nodes to reduce time of transmission.
- '+Delta Comp' adds delta compression to '+ Parallel' which reduces the size of weight data.
- '+Delta Decomp' adds delta decompression to '+ Delta Comp' which reduces the cost of weight decompression.
- '+Index Comp' adds index compression to '+Delta Decomp' which reduces the size of momentum. In other words, the '+Index Comp' version is a full version of FastCheck.

We note that delta compression is activated during checkpointing, while delta decompression with delta merging is activated during checkpoint recovery. We evaluate the contribution of different techniques by running the four versions mentioned above. During checkpointing, as shown in Fig. 16, '+ Parallel' contributes 46.28% by leveraging multi-node parallelism to significantly reduce transmission time. Adding delta compression, '+ Delta Comp' further boosts performance by 22.08%. Delta compression exploits the similarity between the weights of adjacent checkpoints, achieving high reduction ratio. With the delta merging process, '+ Delta Decomp' contributes 0% because it is not activated during checkpointing. '+ Index Comp' contributes 14.55% to overall performance by reducing the I/O latency of momentum through index compression.

During checkpoint recovery, as shown in Fig. 16, '+ Parallel' contributes 54.6%, and '+ Delta Comp' contributes -29.69% to performance, which is because traditional delta compression causes read amplification: as the number of checkpoints increases, traditional delta compression reads both base checkpoints and all deltas, causing linear transmission growth. '+ Delta Decomp' contributes 22.89%. FastCheck's delta merging reduces data size to one checkpoint size, eliminating any performance disadvantage versus baselines. Crucially, FastCheck

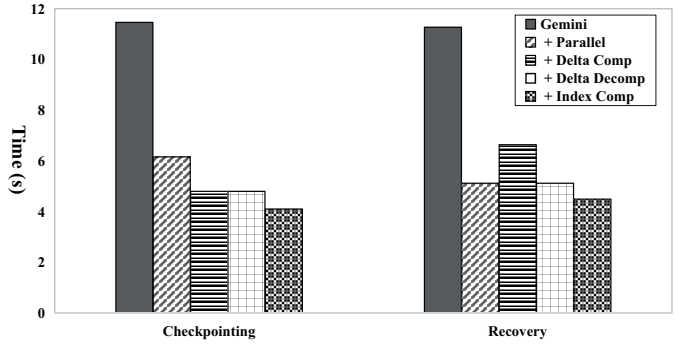

Fig. 16 Performance contributions of individual techniques.

maintains delta compression's substantial time savings during checkpointing while preventing read amplification during recovery. Finally, '+ Index Comp' contributes 12.27% to overall performance.

5.5 Sensitivity to internal parameters

5.5.1 Performance at different training phases

We evaluate overall performance using checkpoints from identical training phases with ConvNext_big. As illustrated in Fig. 17, we measure five checkpoint groups starting at epochs 0, 10, 20, 30, and 40, respectively, with each group's results being the average of three runs. Results indicate that Group 1 (starting with Epoch 0) exhibits slightly higher checkpointing time than the other groups. This occurs because initial training weights exhibit low similarity. As training progresses, the subsequent four groups show little performance variations (less than 2%), demonstrating that the checkpointing and recovery performance has a stable improvement with FastCheck throughout the whole training process. During recovery, all groups demonstrate similar time because FastCheck always loads weights with the same size.

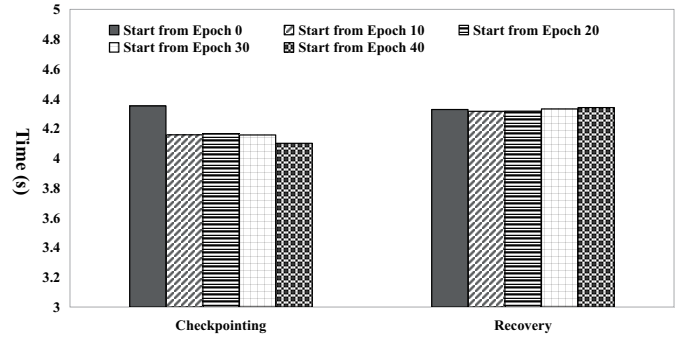

Fig. 17 Performance comparison of different start epochs.

5.5.2 Delta compression with different intervals

For delta compression, we define the interval T as the difference between checkpoint versions. FastCheck computes the deltas between adjacent checkpoints, that is, FastCheck's interval is '1'. When using a larger interval T_1, recovery would require reading only $1/T_1$ of all deltas. However, a larger T may compromise compression efficiency, because of the low similarity in weights of different checkpoints with long intervals. Since FastCheck preemptively merges deltas, FastCheck needs to only determine the optimal T value without reading any deltas during recovery. As illustrated in Fig. 18, we evaluate five interval settings (T from '1' to '5') and measured the reduction ratios across 30 epochs. Results indicate T=1 delivers optimal reduction ratio, which is used by FastCheck.

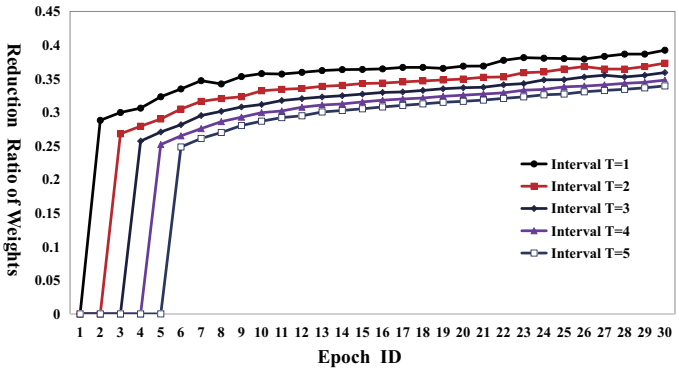

Fig. 18 Reduction ratio of weights by applying different intervals during delta compression.

5.5.3 Different number of threads for filling

During compression, FastCheck fills compressed data into byte streams for transmission. FastCheck applies parallelism to accelerate filling. To evaluate the impact of thread number on filling time, we measure the time of filling compressed data by using five thread configurations: 1, 2, 4, 8, and 16 threads. We evaluate the weight filling time, momentum filling time, and total filling time. The results in Fig. 19 demonstrate that 16 threads achieve optimal total filling time (0.557 s), eight threads demonstrate similar optimal results, and a single thread achieves the worst total filling time (4.254 s); this is because we use node parallelism to accelerate the filling process.

5.5.4 Different numbers of data shards

We conduct experiments varying the number of data shards N from '1' to '4' using ConvNext-Big checkpoints. Fig. 20 reports the checkpointing and recovery times. Under our setup, N=3 achieves optimal performance. Increasing N to '4' yields nearly identical performance due to egress bandwidth saturation.

In practice, N should be chosen according to system bandwidth and workload characteristics. Beyond a threshold, increasing N provides no speedup while introducing additional overhead and higher failure probability.

When a training node transmits data to storage nodes, its egress bandwidth exceeds the ingress bandwidth of any single target node. We therefore use parallel transmission to better utilize the available egress bandwidth. As the number of data shards increases, the aggregate ingress bandwidth of multiple target nodes eventually exceeds the sender's egress capacity, at which point the egress bandwidth becomes fully saturated. Beyond this point, adding more shards does not further improve throughput. Consequently, ingress bandwidth is the bottleneck when transmitting to a single node, whereas egress bandwidth becomes the bottleneck as the number of

shards grows.

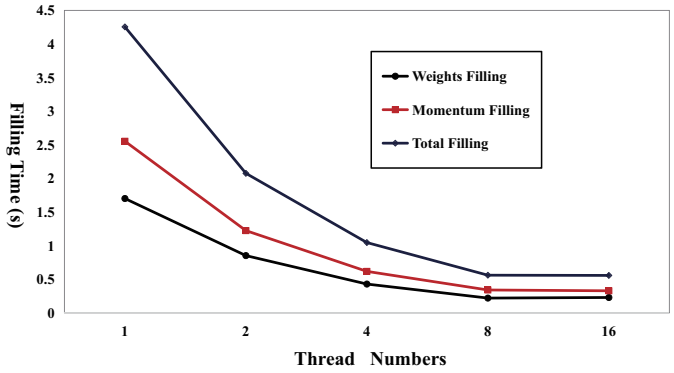

Fig. 19 FastCheck's filling time for different numbers of threads.

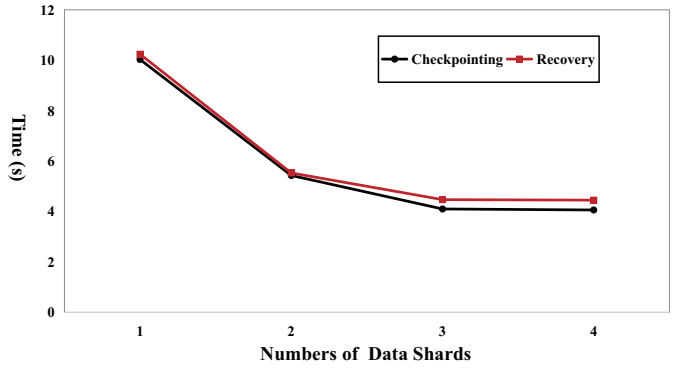

Fig. 20 FastCheck's checkpointing and recovery time for different numbers of data shards.

5.6 Memory overhead and resource contention

FastCheck retains only the latest checkpoint during training. Thus, only a single compressed checkpoint needs to be stored in memory during transmission. With 768 GB of memory per node in our setup, the additional memory footprint accounts for only a small fraction of total memory and thus incurs negligible memory pressure. Since both checkpointing and training data accesses exhibit poor locality, the additional memory usage has minimal impact on caching.

To evaluate training interference, we measure GPT-2 training time with and without concurrent checkpoint transmission. Training without checkpoint transmission takes 139365 s, while training with checkpoint transmission takes 139744 s, an increase of approximately 0.3%. This shows that FastCheck has negligible impact on training performance.

6 Related works

6.1 Optimization for checkpointing

Checkpointing is a mature technology widely applied in various scenarios, with traditional cases including high-performance-computing, distributed storage, and databases. These works aim to enhance checkpoint performance by reducing checkpointing/recovery latency. For DNN models, existing methods periodically save model states to remote persistent storage (Mohan et al., 2021; Danchev et al., 2023). However, checkpointing is limited by the bandwidth of persistent storage. To reduce checkpoint overhead, DeepFreeze (Nicolae et al., 2020) uses asynchronous checkpointing, yet it still stores checkpoints in remote persistent storage. CheckFreq dynamically adjusts checkpoint frequency, but slow access to persistent storage prevents high checkpointing frequency. To improve checkpointing performance and address the impact of critical failures, Gemini stores checkpoints in the CPU memory of another node, enabling a higher I/O bandwidth. Nevertheless, Gemini fails to leverage parallelism across multiple nodes. FastCheck divides checkpoints into several shards and distributes them across the CPU memory of multiple training nodes, further improving performance of checkpointing and recovery.

6.2 Data compression

Compression methods have long been a universal approach to reducing storage size. Lossy compression is primarily applied to images and videos (e.g., JPEG (Rao and Hwang, 1996), MP3 (Sterne, 2020)), wherein data precision is less critical. Traditional high-performance lossless compressors such as GZIP and Huffman coding (Huffman, 1952) excel in general scenarios; however, their designs target specific patterns and are not suited for checkpoint data due to the distinct characteristics in DNN models. Several studies have explored compressing checkpoints from model training. LC-Checkpoint (Chen et al., 2020) proposes a lossy compression scheme but compromises model accuracy. Works with Check-N-Run (Eisenman et al., 2022), Inshrinkerator (Agrawal et al., 2024), and QD-Compressor (Zhang et al., 2021) use quantization to reduce checkpoint size, yet quantization still degrades precision. Delta-DNN (Hu et al., 2020) incrementally compresses floating-point numbers between consecutive checkpoints but suffers from read amplification during recovery. Furthermore, Delta-DNN does not consider momentum. FastCheck proactively merges deltas to eliminate read amplification and uses index compression for momentum.

6.3 Recovery in parallel

Fault tolerance is important in storage systems. Numerous works have recovered lost data in parallel(Strati et al., 2025; Zhang et al., 2025). In erasure-coded storage systems, recovering original data requires reading data or parity blocks from multiple nodes. EC-Cache (Rashmi et al., 2016) accelerates recovery by parallelizing read requests across all nodes storing erasure-coded blocks. Some approaches reduce checkpointing overhead by overlapping computation with checkpointing operations through parallelism. Megatron-LM (Shoeybi et al., 2019) and PCcheck (Strati et al., 2025) implement parallel checkpoint accessing by dividing checkpoints into shards for parallel checkpointing/recovery. For critical failures, FastCheck uses parallel checkpointing and recovery to enhance performance, and we maintained the health status of all nodes for transmitting checkpoint shards to healthy nodes.

7 Conclusion

This paper presents FastCheck, a fast checkpointing and recovery framework for DNN training. By applying parallel transmission and tailored compression, FastCheck achieves significantly fast checkpointing and recovery for critical failures. Experiments on multiple DNN models show that FastCheck achieves up to 78.42% reduction in checkpointing time when compared to existing approaches. In future works, we plan to apply FastCheck into distributed KV cache systems, which can provide a memory pool constructed by the memories of multiple nodes. Training nodes need not consider whether the location of the checkpoints is local or remote.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62025203 and No. 62372419).

Author contributions

Yun Teng designed the research. Shipeng Hu and Zhiyue Li processed the data. Yun Teng and Guangyan Zhang drafted the manuscript. Haidong Tian and Rui Chang helped organize the manuscript. Yun Teng and Dawei Sun revised and finalized the paper.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve language. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

Agrawal A, Reddy S, Bhattamishra S, et al., 2024. Inshrinkerator: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. Proceedings of the 2024 ACM Symposium on Cloud Computing, p.1012-1031. https://doi.org/10.1145/3698038.3698553

Cai W, Chen H, Zhuo Z, et al., 2022. Flexible supervision system: a fast fault-tolerance strategy for cloud applications in cloud-edge collaborative environments. IFIP International Conference on Network and Parallel Computing, p.108-113. https://doi.org/10.1007/978-3-031-21395-3_10

Chen Y, Liu Z, Ren B, et al., 2020. On efficient constructions of checkpoints. *arXiv*, . https://doi.org/10.48550/arXiv.2009.13003

Chorey S, Sahu N, 2024. Rapid Recover Map Reduce (RR-MR): Boosting failure recovery in Big Data applications. *Journal of Integrated Science and Technology*, 12(3):773-773. https://doi.org/10.62110/sciencein.jist.2024.v12.773

Chowdhery A, Narang S, Devlin J, et al., 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(1):113.

BLOOM Chronicles, 2022. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md,

Danchev V, Nikoulina V, Laippala V, et al., 2023. Bloom: A 176b-parameter open-access multilingual language model, . https://doi.org/10.48550/ARXIV.2211.05100

Deutsch P, 1996.

Eisenman A, Matam KK, Ingram S, et al., 2022. Check-N-Run: A checkpointing system for training deep learning recommendation models. 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), p.929-943.

Gill P, Jain N, Nagappan N, 2011. Understanding network failures in data centers: measurement, analysis, and implications. Proceedings of the ACM SIGCOMM 2011 Conference, p.350-361. https://doi.org/10.1145/2018436.2018477

Han S, Mao H, Dally WJ, 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:151000149*, .

Hu Q, Ye Z, Wang Z, et al., 2024. Characterization of large language model development in the datacenter. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), p.709-729.

Hu Z, Zou X, Xia W, et al., 2020. Delta-dnn: Efficiently compressing deep neural networks via exploiting floats similarity. Proceedings of the 49th International Conference on Parallel Processing, p.1-12. https://doi.org/10.1145/3404397.3404408

Huang G, Liu Z, Van Der Maaten L, et al., 2017. Densely connected convolutional networks. Proceedings of the IEEE conference on computer vision and pattern recognition, p.4700-4708. https://doi.org/10.1109/CVPR.2017.243

Huawei, 2024. FusionCube A3000. https://support.huawei.com/enterprise/zh/distributed-storage/fusioncube-a3000-pid-261115115

Huffman DA, 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098-1101. https://doi.org/10.1109/JRPROC.1952.273898

Jeon M, Venkataraman S, Phanishayee A, et al., 2019. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. 2019 USENIX Annual Technical Conference (USENIX ATC 19), p.947-960.

Krizhevsky A, Sutskever I, Hinton GE, 2017. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84-90. https://doi.org/10.1145/3065386

Li W, Chen X, Shu H, et al., 2024. Excp: Extreme llm checkpoint compression via weight-momentum joint shrinking. *arXiv preprint arXiv:240611257*, . https://doi.org/10.48550/arXiv.2406.11257

Lian X, Jacobs SA, Kurilenko L, et al., 2025. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. 2025 USENIX Annual Technical Conference (USENIX ATC 25), p.1519-1534.

Lin J, Gan C, Han S, 2019. Tsm: Temporal shift module for efficient video understanding. Proceedings of the IEEE/CVF international conference on computer vision, p.7083-7093. https://doi.org/10.1109/TPAMI.2020.3029799

Liu Z, Mao H, Wu CY, et al., 2022. A convnet for the 2020s. Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, p.11976-11986. https://doi.org/10.1109/CVPR52688.2022.01167

Maeng K, Bharuka S, Gao I, et al., 2021. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems*, 3:637-651.

MLPerf, 2020. MLPerf Training Results v0.7. https:// github.com/mlperf/training_results_v0.7

Mohan J, Phanishayee A, Chidambaram V, 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. 19th USENIX Conference on File and Storage Technologies (FAST 21), p.203-216.

Nicolae B, Li J, Wozniak JM, et al., 2020. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), p.172-181. https://doi.org/10.1109/CCGRID49817.2020.00-76

Nishtala R, Fugal H, Grimm S, et al., 2013. Scaling memcache at facebook. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), p.385-398.

Ongaro D, Ousterhout JK, 2014. In search of an understandable consensus algorithm. *draft of october*, .

Paszke A, Gross S, Massa F, et al., 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Qi G, Li Z, Wu C, et al., 2025. Eccheck: Enhancing in-memory checkpoint with erasure coding in distributed dnn training. 2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS), p.36-46. https://doi.org/10.1109/ICDCS63083.2025.00033

Radford A, Wu J, Child R, et al., 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Rao KR, Hwang JJ, 1996. Techniques and standards for image, video, and audio coding. Prentice-Hall, Inc.

Rashmi K, Chowdhury M, Kosaian J, et al., 2016. EC-Cache: Load-Balanced, Low-Latency cluster caching with online erasure coding. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), p.401-417.

Shoeybi M, Patwary M, Puri R, et al., 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:190908053*, . https://doi.org/10.48550/arXiv.1909.08053

Simonyan K, Zisserman A, 2014. Very deep convolutional networks for large-scale image recognition. *arXiv*, . https://doi.org/10.48550/arXiv.1409.1556

Sterne J, 2020. MP3: The meaning of a format. Duke University Press.

Strati F, Friedman M, Klimovic A, 2025. Pccheck: Persistent concurrent checkpointing for ml. Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, p.811-827. https://doi.org/10.1145/3669940.3707255

Tan C, Jin Z, Guo C, et al., 2019. NetBouncer: Active device and link failure localization in data center networks. 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), p.599-614.

Tang X, Zhai J, Yu B, et al., 2017. An efficient in-memory checkpoint method and its practice on fault-tolerant hpl. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):758-771. https://doi.org/10.1109/TPDS.2017.2781257

Meta AI LLaMA Team, 2024. The Llama 3 Herd of Models. https://ai.meta.com/research/publications/the-llama-3-herd-of-models/

Tiwari D, Gupta S, Rogers J, et al., 2015. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), p.331-342. https://doi.org/10.1109/HPCA.2015.7056044

Wang Z, Jia Z, Zheng S, et al., 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. Proceedings of the 29th Symposium on Operating Systems Principles, p.364-381. https://doi.org/10.1145/3600006.3613145

Xiao G, Lin J, Seznec M, et al., 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. International conference on machine learning, p.38087-38099.

Zhang B, Ainsworth S, Mukhanov L, et al., 2025. Parallaft: Runtime-based cpu fault tolerance via heterogeneous parallelism. Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, p.584-599. https://doi.org/10.1145/3696443.3708946

Zhang S, Wu D, Jin H, et al., 2021. Qd-compressor: A quantization-based delta compression framework for deep neural networks. 2021 IEEE 39th International Conference on Computer Design (ICCD), p.542-550. https://doi.org/10.1109/ICCD53106.2021.00088

Zhang S, Roller S, Goyal N, et al., 2022. Opt: Open pre-trained transformer language models. *arXiv*, . https://doi.org/10.48550/ARXIV.2205.01068

Zhang T, Liu K, Kosaian J, et al., 2023. Efficient fault tolerance for recommendation model training via erasure coding. *Proceedings of the VLDB Endowment*, 16(11):3137-3150. https://doi.org/10.14778/3611479.3611514