Poster: Swift: Expedited Failure Recovery for Large-scale DNN Training

Yuchen Zhong\(^1\), Guangming Sheng\(^1\), Juncheng Liu\(^2\), Jinhui Yuan\(^2\), Chuan Wu\(^1\)
\(^1\)The University of Hong Kong, China, \(^2\)OneFlow Inc., China
\{yczhong,gmsheng, cwu\}@cs.hku.hk, {liujuncheng, yuanjinhui}@oneflow.org

Abstract
As the size of deep learning models gets larger and larger, training takes longer time and more resources, making fault tolerance critical. Existing state-of-the-art methods like CheckFreq and Elastic Horovod need to back up a copy of the model state in memory, which is costly for large models and leads to non-trivial overhead. This paper presents Swift, a novel failure recovery design for distributed deep neural network training that significantly reduces the failure recovery overhead without affecting training throughput and model accuracy. Instead of making an additional copy of the model state, Swift resolves the inconsistencies of the model state caused by the failure and exploits replicas of the model state in data parallelism for failure recovery. We propose a logging-based approach when replicas are unavailable, which records intermediate data and replays the computation to recover the lost state upon a failure. Evaluations show that Swift significantly reduces the failure recovery time and achieves similar or better training throughput during failure-free execution compared to state-of-the-art methods without degrading final model accuracy.

CCS Concepts: • Computing methodologies → Distributed artificial intelligence.

Keywords: Distributed DNN Training; Failure Resilience

1 Introduction
Large deep neural networks (DNNs) have recently been shown to improve model performance [2], but training these models is prone to failures due to the use of many machines (e.g., hundreds of GPU machines) and long training time (e.g., days to months). Currently, the most common method for fault tolerance in deep learning frameworks is global checkpointing, which periodically saves the entire model state (i.e., parameters and optimizer states) and restarts from the latest checkpoint in the event of a failure. Depending on the checkpointing frequency, this often results in several hours of lost computation time [8]. CheckFreq [9] achieves more frequent checkpoints by splitting the operation into two phases: first, the model state is copied into the GPU memory, called a snapshot, or to the CPU memory if the GPU memory is insufficient; next, the snapshot is written to the disk asynchronously. Elastic Horovod [1], a framework for elastic training, adopts a similar approach but without the second phase. The reason is that Elastic Horovod assumes distributed data-parallel training, where each worker maintains a replica of the model state; during failure recovery, one of the surviving workers broadcasts the snapshot to other workers, and all workers restart training from the snapshot. Taking a snapshot is necessary for Elastic Horovod to prevent a corrupted state: if a failure occurs during the model update, the surviving workers are in an awkward situation - some parameters are updated while the others are not. We identify this problem as the crash-consistency problem. However, we found that both methods can slow down training due to the overhead of snapshotting, as shown in Figure 1.

This paper studies a better failure resilience design for distributed DNN training that significantly reduces the recovery overhead without affecting training throughput and final model accuracy. Swift uses a combination of replication-based recovery and logging-based recovery to achieve this goal. We implement Swift in PyTorch [10] and the code is publicly available at https://github.com/jasperzhong/swift.

2 Swift Design
First, Swift uses a novel method called update-undo that resolves model state inconsistencies caused by the failure and thus enables replication-based recovery using replicas of the model state in data parallelism without creating additional snapshots. Second, Swift proposes logging-based recovery to achieve expedited failure recovery in pipeline parallelism.

2.1 Update-undo
Many update operators in optimizers like SGD and Adam [6] are mathematically invertible, meaning that there is an inverse operator that can reverse the operation of the original operator. For example, linear operators like element-wise addition and scalar multiplication are all invertible. However, some optimizers involve non-linear operators, such as the LAMB optimizer [13] which scales gradients with the L2 norm of the parameters. In this case, it is necessary to save the L2 norm as a scalar for recovery purposes. In the
We propose logging-based recovery for pipeline parallelism. We utilize the surviving workers to assist in the recovery of the failed workers. By logging the intermediate results of all micro-batches, we can use data-parallel training based on the logged data to expedite the recomputation of lost states. To further improve the recovery process, parallel recovery is made possible by logging inter-group communication (i.e., the sender rather than the receiver logs the message to ensure that the intermediate data needed for recovery is not lost upon a failure. Logging is done asynchronously in the background using a dedicated CUDA stream. A queue is set up for each worker. The worker keeps pushing outgoing tensors into the queue during training, while another background thread keeps reading tensors from the queue and doing the logging. In addition, we perform logging during the bubble time in synchronous pipeline-parallel training. In this way, logging is off the critical path. Note that global checkpointing is still used to limit the logging size because all logging files are obsoleted after a global checkpointing.

In the event of a failure, the surviving upstream workers flush the queue of uncompleted logging tasks when detecting a failure in the training job. The upstream workers then upload their logging files to global storage (e.g., HDFS). The replacement workers for the failure workers then download the necessary logging files from the global store, load the latest checkpoint, and replay previously received tensors from the logging file in the exact order of their timestamps. If necessary, the surviving workers will undo the update. This method allows for a more limited scope of recovery, focusing on the local computation graph on the failed machine rather than the entire computation graph compared to pure global checkpointing, resulting in faster recovery. Note that logging requires the computation to be deterministic (i.e., the same input leads to the same output).

**Parallel recovery.** To further improve the recovery process, we utilize the surviving workers to assist in the recovery of the failed workers. By logging the intermediate results of all micro-batches, we can use data-parallel training based on the logged data to expedite the recomputation of lost states.

**Selective logging.** We next investigate a trade-off between the storage space and the recovery time with selective logging. Our idea is to group machines and log inter-group communication but not intra-group communication. The original approach is a particular case where each machine forms a group. If one machine in a group fails, training on the entire group of machines needs to be rolled back from the latest checkpoint, and the recovery time will be longer.

### 3 Evaluation
We experiment on 16 DGX-2 machines, each equipped with eight 32 GB V100 GPUs connected via 40Gbps Ethernet. We compare the performance of SWIFT with CheckFreq and Elastic Horovod for training a scaled-up version of the Wide-ResNet-50 model [14] (base channel size 320, 1.23 billion parameters) on the ImageNet dataset [12] using data parallelism, and with synchronous logging (i.e., saving a tensor before sending it) and global checkpointing for training a BERT-128 model [3] (128 transformer layers, 1.11 billion parameters) on the Wikipedia dataset [3] using pipeline parallelism. We simulate a failure by killing one machine at iteration 100. Figure 1 shows that SWIFT’s replication-based recovery significantly reduces recovery time by 98.1% compared to CheckFreq and Elastic Horovod for Wide-ResNet-50. Figure 2 shows that SWIFT’s logging-based recovery achieves similar throughput while reducing recovery time by up to 76.3% compared to global checkpointing for BERT-128. In addition, SWIFT demonstrates no loss of accuracy in end-to-end finetuning tasks for BERT-Large on the SQuAD dataset [11] and ViT-Base/32 [4] on the CIFAR-100 dataset [7], compared to its failure-free counterparts.

**Simulation study.** We calculate the expected end-to-end training time using traces collected in our experiments. We inject failures uniformly during training, assuming a 17-hour median-time-between-failure [8]. Our results show that SWIFT can speed up end-to-end training for Wide-ResNet-50 and BERT-128 by 1.16x and 1.10x, respectively.

### Acknowledgments
This work was supported in part by the Major Scientific Research Project of Zhejiang Lab (No.2019KD0AD01) and grants from Hong Kong RGC (HKU 17204619, 17208920).
References


