IJRAR.ORG E-ISSN: 2348-1269, P-ISSN: 2349-5138

INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS (IJRAR) | IJRAR.ORG

An International Open Access, Peer-reviewed, Refereed Journal

Fault-Tolerant Distributed ML Frameworks for GPU Clusters: A Comprehensive Review

[1]Aditya Bhatia

[1]New York University, New York, USA

Abstract : Fault tolerance has become a critical requirement in distributed deep learning, especially on large GPU clusters. This paper reviews how modern deep learning frameworks achieve fault tolerance during training. I provide an overview of why fault tolerance is needed for multi-GPU training jobs that run for long durations, and discuss key challenges such as failure detection, checkpointing overhead, state synchronization, and maintaining efficiency. I then compare fault-tolerant mechanisms in popular frameworks – including TensorFlow, PyTorch (TorchElastic), Horovod, and Ray – highlighting their approaches to node failures, GPU faults, and network issues. A comparative analysis examines the strengths and trade-offs of parameter-server versus all-reduce strategies and static versus elastic training.

IndexTerms **- Fault-Tolerance, Distributed Machine Learning, TensorFlow, PyTorch, Ray, Horovod**

I. INTRODUCTION

In large-scale machine learning training, fault tolerance refers to the system's ability to continue training (or gracefully recover) despite failures of some components. In practical terms, a fault-tolerant training job can handle the loss of a node or GPU without having to restart from scratch. As deep learning models and datasets grow, training often runs on dozens or hundreds of GPUs for days or weeks. Such long-running, distributed jobs are susceptible to a variety of failures – from hardware faults to network glitches – that can otherwise interrupt training. Without fault tolerance, a single failure in a cluster could waste many hours of computation and delay model convergence.

Fault tolerance is therefore increasingly important for ML training at scale. Historically, high-performance computing jobs relied on periodic checkpoints and full restarts on failure. However, modern deep learning frameworks are incorporating more flexible mechanisms to reduce downtime. The goal is to make distributed training resilient: if a GPU server crashes or a network partition occurs, the training should resume with minimal loss of progress. This review targets an engineering audience, focusing on practical frameworks and recent research that make distributed deep learning on GPU clusters more robust.

II. THE IMPORTANCE OF FAULT TOLERANCE IN GPU CLUSTERS

Large GPU clusters running distributed training face non-trivial failure rates. The probability of at least one failure rises with the number of machines and the training duration. Empirical data from production clusters shows that for long-running jobs (e.g. 1000+ GPU-hours), failures are common – one study reported nearly 25% of 10,000-hour jobs experienced a failure due to node crashes or preemption. [1] In cloud environments, machines can be unreliable and jobs may be preempted (e.g. if using spot instances). Even in well-managed on-premise clusters, the sheer scale means hardware components (GPUs, NICs, power supplies) will fail occasionally. Network issues or transient outages can also disrupt the synchronous communication required in distributed training.

Without fault tolerance, a single node failure typically crashes the entire training job. This risk forces practitioners to either over-provision resources (to finish faster before a failure occurs) or to frequently save checkpoints. Frequent checkpointing mitigates lost work but incurs significant overhead. As jobs train for more epochs on more machines, the chance of failure increases, and relying only on checkpoints can become inefficient. In summary, fault tolerance is needed to ensure we can train large models reliably and cost-effectively, instead of repeating work after crashes. Real-world deployments treat fault tolerance not as a luxury but as a necessity for long-running ML workloads. [1][2]

III. Challenges in Fault-Tolerant ML on GPU Clusters

3.1 Checkpointing and Recovery

Saving model state to durable storage (disk or cloud) is the most common fault tolerance technique. However, writing out multi-gigabyte model checkpoints regularly can stall training. There is a trade-off between checkpoint frequency and overhead – saving every few minutes reduces re-computation on failure but can slow each iteration (Elastic Horovod — Horovod documentation). Efficient schemes like in-memory or asynchronous checkpointing are being explored to reduce this overhead [3][4]

3.2 State Synchronization

In synchronous training (e.g. data-parallel SGD with all-reduce), all GPUs must keep their model weights and optimizer states in sync. If one worker drops out, the others cannot proceed unless a mechanism exists to re-synchronize the state with a new group of workers. This requires rendezvous protocols to regroup remaining nodes or add a replacement node, and possibly roll back a few iterations to a consistent state. Ensuring that new or recovering workers have the right model parameters (either via broadcasting states or loading checkpoints) is non-trivial. [3]

3.3 Redundancy vs. Efficiency

Some fault tolerance strategies use redundancy – for example, running duplicate computations on backup nodes so that if one fails, a replica is already up-to-date. This can provide fast recovery but at high cost (essentially wasting compute in the normal case). [5] On the other hand, approaches without redundancy (checkpoint/restart or dynamic reconfiguration) avoid extra cost when failures are rare, but incur delay during recovery. Designing frameworks that balance reliability and computational efficiency is a key challenge.

3.4 Distributed Coordination

Detecting failures and reconfiguring the training job on the fly requires coordination among nodes. In a parameter-server architecture, if a worker dies the parameter server may detect a timeout and decide how to continue (e.g. ignore that worker or wait for a restart). In all-reduce training, collective operations will hang on a failed node, so an external coordinator is needed to abort or reform the communicators. Frameworks like PyTorch Elastic use a Rendezvous server (often backed by etcd or Kubernetes) to manage membership – all workers agree on the set of alive nodes and can collectively decide when to resume training. [6] The challenge is to perform this coordination quickly and robustly, especially under concurrent failures or network partitions. Vector clocks or consensus algorithms are sometimes employed to ensure a consistent view of which nodes are active

IV. Review of Existing Frameworks and Approaches

Modern deep learning frameworks have adopted different strategies to handle faults in distributed training. I will review how TensorFlow, PyTorch, Horovod, and Ray approach fault tolerance in GPU cluster environments

4.1 TensorFlow

TensorFlow's original distributed model (TensorFlow 1.x) used a parameter server architecture. Model parameters are hosted on dedicated parameter-server processes, and worker processes compute gradients. In asynchronous training mode, if a worker fails, the remaining workers can continue training using the parameter servers' state – essentially just one less gradient contributor for some steps. This offers a degree of fault tolerance: training doesn't halt for a lost worker (though convergence might slow slightly). TensorFlow jobs typically designate one worker as the "chief" to coordinate checkpoints; if the chief fails, TensorFlow can promote another worker to chief to continue, using the last saved checkpoint. Parameter servers themselves can be made fault-tolerant by replication. [1]

In TensorFlow 2.x, the default is to use synchronous training (e.g. tf.distribute.MirroredStrategy for multi-GPU or MultiWorkerMirroredStrategy for multi-node) which employs collective all-reduce. By default, these strategies do not seamlessly handle worker failures – a crash will cause the training to error out. The recommended practice is to combine this with frequent checkpointing and external orchestration: upon failure, restart the training job and have it load the last checkpoint (possibly skipping the already finished steps) [7][8]. TensorFlow leans on checkpoint/restart for fault tolerance in distributed training, rather than dynamic in-job recovery.

To summarize, TensorFlow's approach relies heavily on periodic checkpoints and (if possible) redundant parameter servers to recover from failures.

4.2 PyTorch and TorchElastic

PyTorch's core distributed training performs synchronous all-reduce and, like most MPI-style jobs, will fail if any process fails. To address this, Facebook introduced PyTorch Elastic (previously called TorchElastic) to enable elastic training. PyTorch Elastic is a library and set of primitives allowing a job to dynamically scale without restarting the entire process. [6]. The idea is that a training job can start with a minimum number of workers and, as resources become available, scale up to a maximum number – or conversely, if some workers die (node failure or are preempted), the job continues with fewer workers (down to the minimum).

Crucially, this happens transparently to the training loop: failed nodes are detected and replaced by new ones, and the PyTorch process group is reinitialized to include the new set of workers.

One implementation of PyTorch Elastic is the TorchElastic Controller for Kubernetes. This Kubernetes operator manages the life cycle of an elastic job: it might launch with, say, 4 out of a requested 8 workers if only 4 GPUs are free, and later scale up when more GPUs free up. If a node running one of the workers fails, the operator will spawn a new pod to replace it. The PyTorch job itself, running under torch.distributed.run, will coordinate to include the new pod in training. In summary, PyTorch (with TorchElastic) provides fault tolerance by elasticity: the ability to detect and replace failed nodes during training without a full job restart. This significantly improves resiliency and flexibility compared to static training jobs.

4.2 Horovod

Horovod is a distributed deep learning framework originally developed by Uber for easy scaling of TensorFlow, Keras, and PyTorch training. It uses efficient all-reduce (NCCL or MPI) to average gradients across workers. In its initial versions, Horovod assumed a fixed world size; if any worker failed, the typical outcome was the job would crash (similar to MPI).

However, in version 0.20, Elastic Horovod was introduced to add fault tolerance and elasticity. Elastic Horovod allows the number of workers to dynamically increase or decrease at runtime without requiring a job restart or manual resume from checkpoint. Practically, this means Horovod jobs can continue training with a different number of processes when one or more workers are lost or new ones are added. [9] [10].

The biggest difference when moving from normal distributed training to elastic training is the need to track and synchronize state among the workers as workers are added or removed from the job. To use Elastic Horovod, the training script is augmented to use Horovod's elastic API. The framework provides a hvd.elastic.run decorator and a hvd.elastic.State object to encapsulate all states that need synchronization. In the event of a failure at an arbitrary point (e.g., during an allreduce), Horovod can roll back all workers to the last committed state to recover consistency.

The Horovod+Ray integration is another interesting approach: instead of managing host discovery via SSH or MPI, you can use Ray (discussed below) to launch an elastic Horovod job. Uber's system uses Horovod on Ray to achieve autoscaling and fault tolerance in their deep learning platform. [9]

4.3. Ray

Ray is a general-purpose distributed computing framework that is not specific to deep learning but is often used to scale Python ML workloads. Ray's core is built with fault tolerance in mind: it provides lineage-based fault tolerance for tasks and actors, meaning it can recompute or restart tasks if a node fails, using the recorded dependencies (lineage) to rebuild lost results. [11]

One way Ray is used for deep learning is via higher-level libraries like Ray Train. With Ray Train, users write a training function (for TensorFlow, PyTorch, or Horovod) and Ray takes care of executing it in a distributed fault-tolerant way. If one of the training workers fails, Ray can start a new one and, depending on the training setup, resume from the latest checkpoint. For example, when using Horovod on Ray, the system can detect a lost worker and invoke Horovod's elastic mechanism to continue. [9]

If using parameter servers or other paradigms on Ray, one could implement actors for parameter servers that are restarted upon failure (perhaps loading state from a checkpoint or from other actors). Ray's actor abstraction, which is a stateful worker, can be made robust by Ray's ability to restart actors on failure.

In summary, Ray itself ensures that the orchestration and task execution layer is fault-tolerant: a failed task will be re-executed, and a failed worker actor can be restarted on another node. This does not automatically solve ML-specific state consistency (that still requires saving model weights), but it provides a solid foundation. Using Ray, one can build higher-level fault-tolerant training frameworks (and indeed Ray's own libraries do so). The key advantage is that Ray can combine task parallelism and actor-based models, so it can support parameter-server style (actors as servers) or all-reduce style (tasks representing collective operations) with resilience. Ray is increasingly being adopted to manage distributed training because of these flexible fault tolerance features and its ability to dynamically scale cluster resources.

V. COMPARATIVE ANALYSIS

5.1 Parameter-Server vs All-Reduce

The parameter server architecture (as in early TensorFlow) inherently allows asynchronous training, which can continue through worker failures – surviving nodes just carry on, and new workers can join by pulling latest parameters. However, asynchronous updates introduce staleness that can affect model convergence, and parameter servers can become bottlenecks for large clusters.

All-reduce (used by Horovod, PyTorch DDP) is very efficient for synchronous SGD but traditionally has no tolerance for failure (all nodes must participate in each step). The recent elastic all-reduce solutions bridge this gap by allowing dynamic world size, essentially marrying all-reduce performance with parameter-server-like flexibility. The trade-off is additional complexity in coordination and slight overhead for syncing state when changes occur.

5.2 Static vs Elastic Job Management

TensorFlow's native approach (without specialized support) is static: you allocate a fixed set of N workers, and if any fails, you stop and restart the whole job (after fixing the cluster or letting the scheduler restart the process). This is simple but leads to downtime and extra epochs of training from the last checkpoint. PyTorch Elastic and Horovod Elastic, on the other hand, implement in-job elasticity: the job adapts on the fly. Elastic jobs avoid full restarts, which is a big win for long jobs on transient cloud nodes (e.g., spot instances). They do require that the training loop be robust to changes (e.g., shuffling data differently if worker count changes, ensuring no duplicate processing or missed data when a new worker joins mid-epoch) .In practice, elastic frameworks often restrict changes to epoch boundaries or require deterministic data partitioning schemes.

Horovod Elastic and TorchElastic have demonstrated that for many workloads, the overhead in failure-free operation is minimal – just an occasional state sync and a background heartbeating – making them attractive for production.

5.3 Overhead and Complexity

There is a spectrum between doing nothing (no fault tolerance, simplest code, but brittle) and doing full redundancy (complex and resource-heavy but seamless). TensorFlow's checkpoint/restart is conceptually simple but can become cumbersome with very frequent failures (imagine restarting 5 times in a day-long job). Horovod/PyTorch elastic approaches add some API surface and require an external coordination service (etcd or Kubernetes master) – this adds deployment complexity, but once set up, they handle failures quite gracefully.

The choice often depends on the use case: for extremely critical jobs where any delay is unacceptable (e.g., a model training on a tight deadline), one might tolerate extra resource usage (redundancy) to avoid even a second of downtime. Conversely, cost-sensitive training (like research experiments on cloud) might prefer elastic solutions that use zero extra GPUs, accepting a short pause on failure.

VI. Conclusion

Fault tolerance in distributed deep learning has evolved from a niche concern to a mainstream requirement as training workloads scale up. This review covered how current frameworks approach the problem: from TensorFlow's reliance on checkpoint/restart and parameter server redundancy to the elastic training capabilities of PyTorch and Horovod that can seamlessly recover from node losses. We discussed the inherent challenges such as checkpoint overhead, synchronization of state, and balancing redundancy with efficiency.

For engineers building or using large GPU clusters, these fault tolerance techniques offer ways to improve the reliability of training jobs – reducing wasted computation and enabling use of volatile resources like spot GPUs. However, there is no one-size-fits-all solution, and each approach comes with trade-offs in complexity and performance. The comparative analysis shows that the choice of framework and fault tolerance method should be informed by the specific failure modes expected (e.g. frequent preemptions vs. rare hardware faults) and the tolerance of the training process to delays or staleness.

In practice, many large-scale training systems combine strategies. For example, a training job might use Horovod Elastic (so it can continue through single node failures), but also periodically checkpoint to guard against a rare event of all nodes failing or a power outage. Or a parameter-server based trainer might use backup servers and also do periodic checkpoints to cloud storage for disaster recovery. The frameworks discussed are converging towards giving users a toolbox of fault tolerance mechanisms – from which an appropriate level of protection can be chosen based on cluster reliability and performance needs.

Looking ahead, I expect deeper integration of fault tolerance into ML frameworks. Future systems will likely make fault handling more transparent to users, allowing data scientists to focus on models without worrying about cluster hiccups.

VII. Acknowledgment

I would like to express my gratitude to the team of TensorFlow, PyTorch, Horovod, Ray for providing detailed documentation, which I used extensively to write this paper.

This work was inspired by a prior survey on fault-tolerance in distributed optimization and machine learning. [12]

References

[1] Scaling Distributed Machine Learning with the Parameter Server Mu Li, et al (2014) USENIX.
[2] Elastic Deep Learning with Horovod on Ray | Uber Blog Travis Addair et al. (2021).
[3] Horovod Documentation. Elastic Horovod.
[4] Fault-Tolerant Hybrid-Parallel Training at Scale with Reliable and Efficient In-memory Checkpointing Yuxin Wang et. al.
[5] ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation, Swapnil Gandhi et. al. (2024)
[6] AWS: Fault tolerant distributed machine learning training with the TorchElastic Controller for Kubernetes Mike Stefaniak et. al. (2020)
[7] Tensorflow Core Documentation, Migrate the fault tolerance mechanism
[8] Tensorflow Core Documentation, Distributed training with TensorFlow
[9] Elastic Deep Learning with Horovod on Ray | Uber Blog
[10] Elastic Horovod | Hovord Documentation
[11] Ray: A Distributed Framework for Emerging AI Applications. Philipp Moritz (2018) et. al. USENIX
[12] A Survey on Fault-tolerance in Distributed Optimization and Machine Learning Shuo Liu (2021)