FlexiTrain: Dynamic Scaling for Fault-Tolerant Distributed Training Workloads

Kamyar Salahi
kamyar@stanford.edu
Stanford University
Stanford, California, USA

Matthew Ding
mattding@stanford.edu
Stanford University
Stanford, California, USA

Sathya Edamadaka
sath@stanford.edu
Stanford University
Stanford, California, USA

Abstract

Modern machine learning workloads involve training massive models with billions of parameters across tens of thousands of devices. Failures are inevitable at this scale due to the sheer number of devices and duration of training. Existing methods handle failures by periodically checkpointing the model state at various point throughout training. However, this approach to fault tolerance incurs additional overhead that can significantly reduce training throughput.

In this paper, we present FlexiTrain, a checkpoint-free framework for fault-tolerant, distributed model training. FlexiTrain gracefully handles machine failures by discarding the failed machines, redistributing their training data, and rescaling gradient updates among the remaining nodes. We find that FlexiTrain can improve training throughput by 20% with no observable loss on training quality. We empirically validate our approach by training a vision model and a large language model with simulated faults.

1 Introduction

Many recent advances in machine learning are due to significant increases in model size and compute, which have been shown to exhibit a power-law relation with model performance [4, 6]. As a result, modern state-of-the-art models often have parameters numbering into the tens of billions [1, 8]. Training these massive models requires a careful combination of data, tensor, and pipeline parallelism across distributed clusters with thousands of devices. At this scale, even infrequent individual failures can compound, leading to issues that impact training across the entire cluster. For instance, OPT-175, which was trained using 992 NVIDIA A100 GPUs across two months, reported 100 hardware failures during training, 35 of which were severe enough to require manual restart [14].

Existing fault tolerance methods are not tolerant to changes in the number of compute workers available during training. AntMan [13] and TGS [12] use improved scheduling algorithms and GPU-sharing to increase overall utilization. However, these systems work on static training jobs, where the number of data-parallel workers is fixed at allocation time and does not change. In addition, these systems are not designed to quickly recover from faults. In contrast, our proposal leverages device dynamism to maintain high utilization while also introducing greater fault-tolerance.

Another approach, Singularity [10] offers scheduling that is both elastic and fault-tolerant by taking frequent state snapshots and time-multiplexing accelerators. However, the Singularity system comes with high memory overhead and does not accommodate for scale-up jobs to utilize newly available resources. Reducing this high overhead and enabling dynamic scaling to leverage additional resources when they become available is important to scaling up data parallelism approaches to LLM-scale production environments.

These methods handle worker failures by periodically checkpointing model and optimizer states. This approach carries a significant overhead due to the massive size of modern models and the need to store them in a distributed fashion. Moreover, upon encountering a worker failure, training must restart from the last checkpoint, discarding all intermediate work done since the previous save point. Consequently, this strategy can slow down training by as much as 43% [7]. In the following section, we present simulations and theoretical analysis to quantify the potential throughput gains of an elastic scaling approach compared to the traditional checkpointing method.

In this work, we propose FlexiTrain, an elastic scaling framework that allows training to continue even if one or more workers stops responding during training. FlexiTrain is agnostic to worker type, accommodating a pool of any number of GPU and/or CPU workers. We first motivate our approach through theoretical throughput calculations, as well as simulations of inelastic and elastic training processes. We then discuss the design and implementation of FlexiTrain, addressing core assumptions and tenets of our implementation. We evaluate our system by training models for computer vision and natural language tasks while simulating worker failures. We compare the losses and test accuracies to models trained in a distributed manner without such failures. We show that FlexiTrain consistently improves throughput under worker failure conditions while not inducing any noticeable performance drop.

2 Motivation

2.1 Checkpointing Overhead

We model individual GPU failures as occuring with probability 1.5×10^{-5} in a span of an hour following MegaScale [5] wherein 100 failures took place on a cluster of 10,000 GPUs over the course of a month. Using these probabilities,

we note that the interarrival times of GPU failures are exponentially distributed, meaning GPU failure incidence can be modelled with a Poisson process with with failure incident rate of $\lambda = 0.15$ over the time period of an hour. We assume a checkpoint frequency of once every three hours and assume that loading from checkpoints takes 20 minutes following experimental details from prior large model literature [5]. We assume a cluster of 10,000 GPUs with 64 data parallel ranks.

2.1.1 Simulation. We first model the throughput of our approach in simulation. In particular, over the span of one month, we simulate GPU failures in the timespan of 10 minute intervals. Upon a GPU failure, we assume that the data parallel rank of the GPU failure is inoperable until the next checkpoint has been taken. This means that a single failure will cause 156 GPUs to be idle. We compute throughput through the number of batch iterations that can be processed by our data parallel ranks in the time span of one month.

For the standard checkpointing strategy, we assume that a single failure stalls the entire training pipeline until the checkpoint can be reloaded. When a failure occurs in between two checkpoints, any batches processed in that time interval are not accounted in our final throughput calculation [11]. For both approaches, we compute throughput as a ratio of batch iterations completed to no fault case. Our results are shown below.

Using our simulations, we compute that a failure occurs on average 81 minutes into a checkpointing interval.

Type	Throughput
FlexiTrain (Ours)	99.85
Standard Checkpointing	80.18

Table 1. Simulation Results

2.1.2 Probabilistic Derivation. We compute a lower bound on the throughput of our method by calculating the expected number of failed data ranks at the end of a 3 hour checkpoint interval. As time passes, more data ranks may become unavailable. Since our throughput is measured as an integral over time, computing the expected number of GPUs failures at the end of the interval will provide us with a lower bound on throughput.

$$\int \text{AvailableGPUs}(t) \times \text{BatchSize}\, dt$$

$$\geq \int (\text{GPUCount} - \text{FailedGPUs}) \times \text{BatchSize}\, dt$$

Since we have modelled failure incidence as a Poisson processm, the amount of failures in a given time interval is Poisson distributed. Therefore, we can compute the expected number of available GPUs at the end of an interval using $X \sim \text{Poisson}(3\lambda)$. This yields an expected throughput of 99.30 which resembles our simulation results.

We now compute the throughput of the standard checkpointing method. Since failure interarrival times are exponentially distributed, $T \sim \exp(\lambda)$, we can compute the conditional expected time of failure within an interval, $E[T|T < 3] = 1.387$ or 83.2 minutes. This applies to arbitrary time intervals due to the memoryless property of an exponential distribution. We define *wasted time* as the total sum of time spent recovering from a failure. This includes the time from the last checkpoint spent training before the failure and the time spent retrieving the checkpoint during failure recovery. In the event of a failure, the amount of wasted computation is $E[T] + t_{\text{rev}}$. Using our previous calculations for conditional expected interarrival times, we find that 103.2 minutes are wasted in expectation given a failure. Given the 0.36 probability of a failure occurring within a 3 hour interval, we find an expected throughput of 79.36 In contrast, our method avoids the periodic checkpointing and wasted computation. Instead, our slowdown comes from computing the remainder of the batch on fewer workers. If workers rejoin (which we have not modeled here), however, our throughput approaches that of no GPU failures.

Type	Throughput
FlexiTrain (Ours)	99.30
Standard Checkpointing	79.36

Table 2. Mathematical Derivation

3 Design

Here, we discuss how FlexiTrain allows us to gracefully handle faults by continuing training without the failed device while scaling the data loading and parameter update operations accordingly. In turn, this reduces the lost time by restarting from checkpoints, which improves overall job completion time and throughput.

3.1 Worker Failure Case

In order to coordinate each aspect of FlexiTrain with all workers, a leader node is designated upon the beginning of training. All workers then join a (PyTorch and Python Multiprocessing) process group, which allows for direct communication between any pair of nodes. After synchronously waiting for all workers to set up and join the process group, the leader node starts sending out diagnostic communications to each of the other workers. After finding an average there-and-back communication time over all messages sent, a constant health check process is started. Specifically, each follower node synchronously sends a message to the leader node. They then wait for up to the number of workers times the average diagnostic communication time to hear a response back from the leader. On the leader side, it waits up

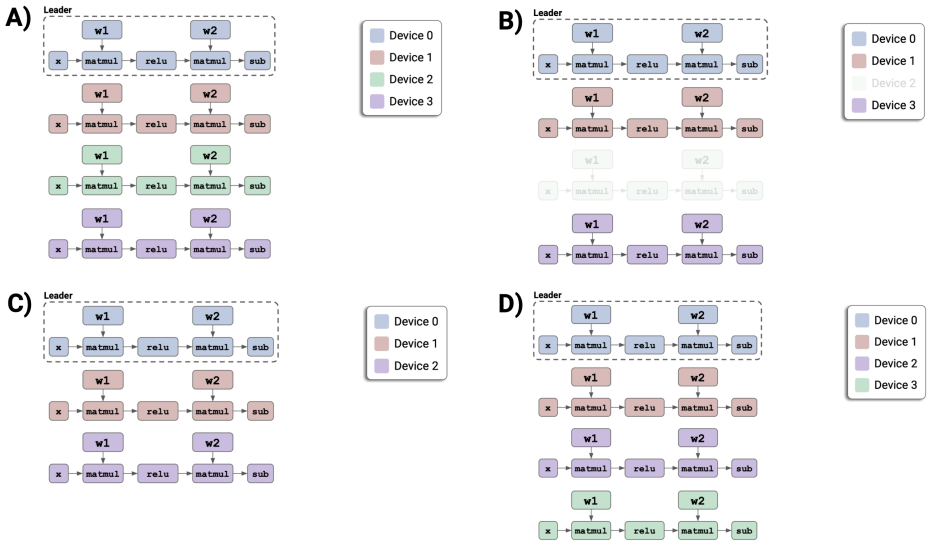

Figure 1. Demonstration of how worker failure and rejoining works in FlexiTrain. A) shows a starting configuration of 4 worker devices, the first of which is the leader that coordinates all health checks. B) shows device 2 failing. C) shows after the leader detects the failure and shrinks the process group accordingly, communicates the updated ranks associated with each node, and updates the world size associated with the entire process group. D) shows how if a worker comes back online, it can be added back into the process group after the latest checkpoint.

to the average diagnostic communication time for each of the follower nodes to send it a message. If it receives all of them, it will then send back out an acknowledge signal to each of the follower nodes and the health check process will continue. This signal consists of a bit vector corresponding to whether or not each of the workers in the process group has responded in time to the health check.

In the case that a worker fails and does not send a message back to the leader within the average diagnostic communication time, visualized in Figures 1A)-C), the leader will skip it and continue to listening for a message from each of the other follower nodes. Then, the leader node notes which nodes have failed and send the acknowledge signal to each of its workers. The data on each worker node that has failed is dropped. This is a valid assumption, as for large model training processes, enough data is present to not significantly decrease performance. All remaining nodes will delete their old process groups, create and join a new process group with the updated total number of workers ("world size") and their adjusted individual worker rank (identical to an MPI process rank). In this way, FlexiTrain can handle an arbitrary number of failures.

The communication overhead of this process is designed to be minimal given that the largest waiting periods (the leader worker waiting for signals from its followers and the follower workers waiting for acknowledge signals from the leader) are completely asynchronous. All synchronous operations are limited only by individual hardware constraints. If at any point during training the overhead caused by waiting for certain nodes exceeds the average diagnostic communication time (i.e. the node becomes too slow), it will be assumed to have failed and will be removed from the process group to prevent further delays. We note that the health check and resizing has no overhead in the fault-free case since our algorithm is only invoked when communication timeouts occur.

However, naively continuing the training process is not mathematically valid, as previously gradient updates were to be pooled together and used equally to adjust all parameters on all data parallel workers. If one worker fails and the other parameters are scaled, the gradient update is not normalized to the amount of data used to actually update the parameters. Instead, parameter updates are scaled by the current world size (instead of the initial one), therefore allowing for nodes to drop without incurring errors in gradient steps. Then, all scaled gradients from all workers are combined via an all reduce sum operation. This allows for distributed training that is accommodative of an actively changing number of workers. These hooks are called once the loss on each worker has been calculated and back propagated.

3.2 Worker Joining Process Group

In the event that workers become available, we can add them to the process groups by halting training, serializing a temporary checkpoint, and restarting training from that checkpoint to reach the state shown in Figure 1D). To minimize the cost of frequent stops, we propose batching available workers and only running the procedure once the number of

additional worker exceeds some threshold. We did not have time to fully implement the worker re-joining procedure in our implementation, but this design provides a simple and lightweight approach for scaling up training.

3.3 Model and Pipeline Parallelism Compatibility

Large language models are also usually too large to fit onto a single GPU, with billions of parameters. Instead, a combination of model parallelism (wherein different parts of models are split up onto different workers) or pipeline parallelism (where these model parallel workers are pipelined together to avoid throughput loss) and data parallelism is used. In a case with only model parallelism, the loss of a single worker node necessarily means the halting of the model, as the stage of the model associated with that node is not replicated elsewhere. However, in the case where there are additional nodes to which data batches are split up and sent, FlexiTrain can easily be implemented to thus accommodate for pipeline model parallelism. A schematic for an example implementation of this on CUDA GPUs (which we also include in our code on the `model-parallel-check` branch) is shown in Figure 2.

Because each worker node in each process group is cognizant of the current world size, the outputted data from all nodes (i.e. all healthy nodes) in the process group can easily be redirected to the next healthy node in the FlexiTrain distributed data parallel group corresponding to the next stage in the model pipeline. In addition, because the nodes used for different stages of the pipeline are independent of the other workers in their FlexiTrain data parallel group, they are able to send information directly using CUDA mechanisms without relying on torch-based multiprocessing methods, further reducing overhead.

Figure 2. Demonstration of how FlexiTrain is accommodative of model parallelism. A) shows how a 3-stage, pipelined model parallelism split of workers is distributed to a pool of 6 workers. Here, two workers are assigned to each stage of the model, in total forming 3 sets of distributed data parallel (DDP) groups (each for a stage of the model pipeline). B) shows when a worker in the first data parallel group fails. C) shows how the first FlexiTrain DDP group shrinks to the single working node, and now routes its output evenly to the next group, as set up in the model pipeline implementation.

4 Implementation

4.1 PyTorch Integration and Dataloader

FlexiTrain is built on top of `torch.distributed`, a distributed communication and worker management system built into PyTorch. All multiprocessing and parallelism methods are built on top of `torch.multiprocessing`, a wrapper around Python's multiprocessing library that allows for PyTorch `Tensor` objects to be passed as data in primitives like `send`, `recv`, and their asynchronous equivalents. In particular, Distributed Data Parallel (DDP) is a class that automates much of the worker management needed for data parallelism. However, it also restarts as soon as a single node fails at the previous checkpoint with the remaining healthy nodes. We implemented a module like DDP that is fully elastic. In addition, it is implemented agnostic to backend or worker type. As a result, it can work for pools of CPUs and/or GPUs, and has been tested on ARM and Intel architecture processors. In addition, we only use features of `torch.distributed` that are compatible with both NCCL and GLOO backends.

We chose to build on top of these frameworks to allow for easy integration into existing codebases, as completely new modules may take longer to become compliant and to be used in existing code. FlexiTrain can serve as a drop in replacement for the DDP class.

In order to enable our FlexiTrain model, we also provide archetypal elastic dataloaders. For the MNIST evaluation trials discussed in section 5, the provided dataloader employs a dynamic batching algorithm. In particular, whenever a worker fails, although the data on that node is dropped, the remaining batch sizes are rescaled to increase the total number of batches but keep the same batch size, preventing

out of memory errors on worker nodes while still distributing work evenly across all healthy workers.

4.2 Leader Failure Case

In the edge case where the leader node fails, a leader election process is held to select a new leader. Follower nodes can detect this if they don't receive an acknowledge signal (consisting the alive worker bit vector) before the timeout. If this is the case, each of the follower workers reduce their rank by 1 (since the rank 0 leader node has now died) and reinitialize each of their process groups with the new rank and a world size of 1 smaller than the total number of workers they knew of before. They iteratively continue this until a new leader node that is healthy reaches rank 0, and then all remaining failed nodes are automatically detected and removed once the next valid acknowledge signal check is completed.

5 Evaluation

We evaluate FlexiTrain on computer vision and natural language tasks and with varied model sizes. In particular, we examine whether the fault recovery procedure impacts the training process or model performance, given that the recovery procedure drops the batch of data being processed at the time of failure.

5.1 MNIST

We trained two 1.2M parameter convolutional neural network models from scratch on the MNIST dataset [2]. The models were trained using a learning rate of $1e-3$ and an effective batch size of 256 across four workers. Both models were trained for 14 epochs. The first model had a simulated failure rate of $1e-4$ resulting in two failures over the course of training. The other model had no failures. We found that when scaling up our model training to 10 workers, failure recovery would take up to 50 seconds. In general, we assume that a node has failed when communication is not received within an approximate 5 second timeout (or 25,000 $\times$ measured communication times at start).

Type	Throughput
FlexiTrain (Ours)	99.18
No Failures	92.98

Table 3. MNIST Experiments

We find slight degradation of test accuracy for our first model. We attribute some of this error to the shrinking batch size during training. Although our tests did not experiment with worker joining, we expect that this addition would result in faster training and better test performance. Furthermore, upon fault, the failed batch is dropped until the next epoch, resulting in slightly smaller training sizes and imbalanced training, which may slightly degrade performance.

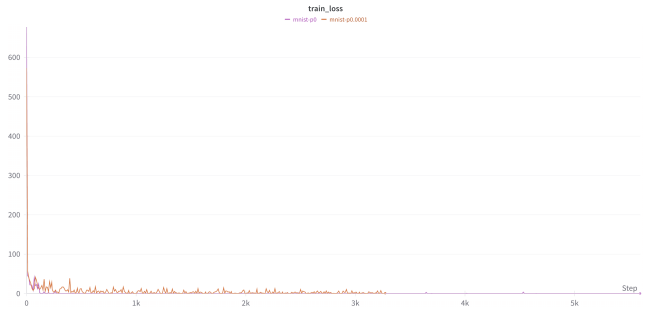

Figure 3. Training losses for our MNIST models. `mnist-p0` (red) was trained with no failures, while `llm-p0.0001` (purple) had two simulated failure. Both models displayed similar training loss curves and final validation losses.

Figure 4. Training losses for our two LLM models. `llm-p0` (pink) was trained with no failures, while `llm-p1e-5` (purple) had one simulated failure. Both models displayed similar training loss curves and final validation losses.

The remaining data for the failed node is shuffled amongst the remaining nodes by our distributed fault tolerant data loader. The training losses are shown in Figure 3.

5.2 LLM

We trained two 78M parameter language models from scratch on the TinyStories dataset [3]. The models have GPT2-style architectures, consisting of alternating pre-norm Transformer blocks and fully-connected Linear layers. We used the Llama3 tokenizer with a vocab size of 128256. The model hyperparams are batch size 128, context length 256, `d_model` 512, `d_ff` 2048, 4 transformer layers, 16 attention heads, and dropout 0.1. Both models trained for 5000 iterations on two NVIDIA H100 GPUs. The training losses are shown in Figure 4. The final validation losses are 1.864 for `llm-p0` (no failures) and 1.857 for `llm-p1e-5` (one recovered failure). Samples of generated output are shown in Appendix A.

6 Discussion

6.1 Training on Preemptible Instances

Our framework enables a significant cost-saving opportunity by allowing for large model training on preemptible cloud instances. This approach was previously impractical due to the high frequency of preemptions, which would require fully restarting training from the previous checkpoint. However, our framework seamlessly accomodates preemptions with minimal impact on the training process. This approach is particularly effective when the leader node is a non-preemptible instance, as this configuration streamlines the fault recovery mechanism. Building a proof of concept that demonstrates this applicability would be an effective next step.

6.2 Extension to Fully Sharded Approaches

Although our results primarily focus on the case of data parallel, our methods can be easily extended to support pipeline and tensor parallelism since the process groups and ranks across the model would not change, as discussed earlier. However, recent approaches such as ZeRO [9] or PyTorch's Fully-Sharded Data Parallel [15] leverage sharded weights, optimizer states, or gradients to enable larger models to be stored within GPU memory. Since our approach achieves fault-tolerance through redundancy and replication of models, it is not immediately applicable to fully sharded methods—a potential extension in the future. However, we believe that with shard replication and replica tracking, FlexiTrain could strike an excellent trade-off between training throughput and memory consumption.

7 Code Availability

The open source repository we have built for FlexiTrain is available for use at this Github link:

https://github.com/TheQuantumFractal/CS349D

Acknowledgments

We would like to thank the Stanford CS349D teaching team for their feedback on this project throughout the quarter.

References

[1] AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card* (2024).

[2] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[3] Ronen Eldan and Yuanzhi Li. 2023. TinyStories: How Small Can Language Models Be and Still Speak Coherent English? arXiv:2305.07759 [cs.CL]

[4] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs.CL]

[5] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[6] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG]

[7] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C. Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2020. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. arXiv:2011.02999 [cs.LG]

[8] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[9] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[10] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. 2022. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv:2202.07848 [cs.DC]

[11] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[12] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[13] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[14] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

[15] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

Figure 5. Comparison of output generated by our language models. `llm-0` (left) was trained on 2 GPUs with no failures, while `llm-1e-5` (right) had a GPU failure halfway through training. Both models were trained from scratch on the TinyStories dataset.

A LLM Generated Output

h

Figure 5 shows a side-by-side comparison of text generated by our language models, alongside the perplexity attained by the model's own weights. We find that both trained models produce similar output and perplexity values, despite the `llm-1e-5` model sustaining a GPU failure during training. For more details about the model architecture and training process, see Section 5.2.