Rehabilitating over Recomputing: A Novel Failure Recovery Method for Large Model Training

Zichen Wang[1], Hongliang Li[*,1,2], Jie Wu[*,3,4], Zhewen Xu[1], Hairui Zhao[1], Qi Tian[1], Haixiao Xu[5]
[1]College of Computer Science and Technology, Jilin University, China
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, China
[3]China Telecom Cloud Computing Research Institute, China
[4]Department of Computer and Information Sciences, Temple University, USA
[5]High Performance Computing Center, Jilin University, China
zichen22@mails.jlu.edu.cn, lihongliang@jlu.edu.cn, jiewu@temple.edu,
{zwxu20, zhaohr21, qitian23}@mails.jlu.edu.cn, haixiao@jlu.edu.cn

***Abstract*—Today, large DNN models are often trained in parallel and distributed environments, which inevitably incur high costs for fault tolerance. Existing solutions rely on periodic checkpoints to save training state in run-time and recomputing from the latest checkpoint to recover upon a failure. Despite recent efforts to reduce checkpoint cost, data transfer bandwidth can become a bottleneck, limiting checkpoint frequency, and consequently leading to high recomputing cost. This paper focuses on efficient fault tolerance for large model training and proposes Controlled Predicting-assisted Self-Recovery (CPSR). Instead of resource-consuming recomputation, it features a lightweight predictor, fed by routine checkpoints, to predict training state prior to a failure. The prediction error exhibits as a minor perturbation that can be self-corrected by the training process itself. We propose a quantified model of predicting-based recovery cost during rehabilitation and introduce a novel checkpoint interval problem that seeks to minimize the overall fault tolerance cost. We present a solution to compute the optimal checkpoint interval configuration for a given setting, balancing checkpoint and recovery cost. Extensive testbed experiment data demonstrate that CPSR reduces the recovery cost by 41.66% on average compared with state-of-the-art approaches, while introducing a small GPU memory footprint (less than 200MB).**

***Index Terms*—Fault Tolerance, Large Model Training, Self-correcting, Predicting-based Recovery**

I. Introduction

In recent years, DNN models have experienced rapid growth in terms of parameter size [1] and dataset volume [2]. For example, the PaLM large language model has 540 billion parameters [3], and is trained using a dataset of 780 billion tokens. Large model training is usually computationally intensive and time-consuming. OPT-175B is trained with 992 GPUs for more than two weeks [4] and DeepSeek-v3 used 2,048 GPUs for 50 days [5]. Various parallel schemes, for example, Data Parallel (DP) [6], Pipeline Model Parallel (PMP) [7–10], tensor parallel [11, 12], sequence parallel [13] and MoE parallel [14, 15], have emerged to facilitate large DNN model training jobs, especially in distributed environments. These training jobs inevitably suffer from high fault rates [4, 5]. Meta's LLaMA3 pretraining has been reported to have experienced 466 interruptions during a 54-day period [16].

[*]Corresponding author: lihongliang@jlu.edu.cn, wujie@chinatelecom.cn

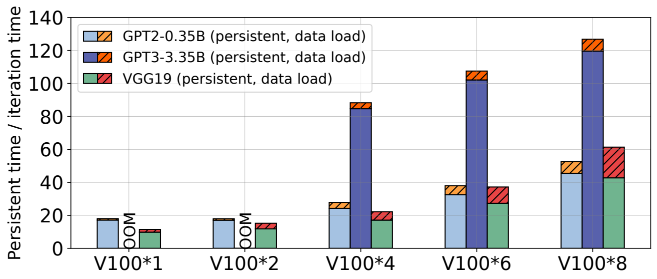
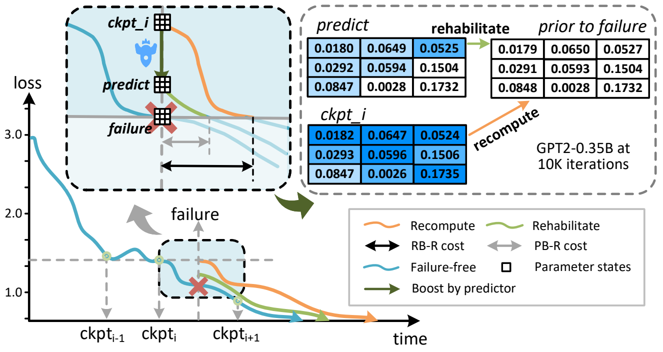


Figure 1. Performance impacts of bandwidth competition.

Checkpoint/recompute has emerged as the dominant fault tolerance method for training large models [17–20], but introduces a high cost of fault tolerance, including the checkpoint cost from training interruptions in run-time and the recovery cost caused by recomputing the lost progress upon a failure. Recent studies have focused on decoupling checkpoint operations from training workflow to reduce checkpoint cost. Specifically, recent checkpoint mechanisms [17, 19, 20] save the state of a training job by two steps: snapshot (GPU-to-CPU) and persistence (CPU-to-storage).

The latest snapshot methods significantly reduced training interruptions, but frequent checkpointing remains infeasible [20]. The checkpoint interval of large training jobs can be as long as 3 hours [21], leading to high recomputing cost in the event of a failure. For example, the recomputing cost for the OPT-175B training reached 178,000 GPU hours, 13% of the total amount [4]. This is mainly because the limited CPU-to-storage I/O bandwidth increases the persistence time [19, 20], especially when the model states become larger. Due to the high I/O contention among the tasks, the checkpoint time for a large training job can reach 10-15 minutes [4]. Fig. 1 shows a similar situation we experienced when multiple parallel tasks (GPT/CodeSearch) perform concurrent checkpoints on a server. Moreover, frequent data loading operations in modern training frameworks [22, 23] further aggravate I/O pressure, thus increasing persistent time (by 19.7%). Data bandwidth can become a bottleneck, restricting the checkpoint frequency.

This paper revisits the issue of failure recovery for large model training from a novel perspective. Recent studies sug-



predict

0.0180	0.0649	0.0525
0.0292	0.0594	0.1504
0.0847	0.0028	0.1732

prior to failure

0.0179	0.0650	0.0527
0.0291	0.0593	0.1504
0.0848	0.0028	0.1732

ckpt_i

0.0182	0.0647	0.0524
0.0293	0.0596	0.1506
0.0847	0.0026	0.1735

Figure 2. Boosted rehabilitation vs. recomputing recovery.

gested that the model training process is capable of tolerating minor perturbations, introduced by quantization [24, 25], lossy recovery [26, 27], or asynchronous model updating [28], etc. Instead of recovering the training state with expensive recomputing, we take advantage of the self-correcting characteristic [26] of model training and let it rehabilitate. We propose Controlled Predicting-assisted Self-Recovery (CPSR) that a) includes a lightweight predictor, fed by routine checkpoints, to quickly predict training state upon a failure, giving the recovery a boost start (as shown in Fig. 2), and b) incorporates a quantified model of the Predicting-Based Recovery (PB-R) cost to guide control of the PB-R process.

As Recomputing-Based Recovery (RB-R), the proposed PB-R method also restarts from the latest checkpoint but does not require recomputation, therefore, has more potential for fast failure recovery. It introduces two challenges. a) How to accommodate the predictor training in the original training process, without introducing an additional device, is a key acceptance factor in real-world applications. b) Accurate prediction of the training state brings the training closer to where the failure occurred and, therefore, requires less recovery. However, it depends on a larger checkpoint sample size, which consequently increases the checkpoint frequency. How to trade off prediction accuracy and checkpoint cost is a key issue.

This paper makes the following key contributions:

- We propose a novel predicting-based failure recovery framework, CPSR, that incorporates predicting-assisted boost to the self-recovery of model training. It accommodates a lightweight predictor to support fast and accurate failure recovery in parallel and distributed environments.
- We introduce a quantified recovery cost model and a novel checkpoint interval problem that minimizes the overall fault tolerance cost. A solution is proposed to compute the optimal checkpoint intervals.
- Comprehensive data from testbed experiments based on real application setups demonstrate that CPSR reduces the recovery cost by 41.66% on average compared with state-of-the-art approaches.

II. Related Work

A. Consistent Fault Tolerance

Traditional fault tolerance approaches aim to restore an interrupted job to its exact state before a failure. The Checkpoint/restart method periodically saves the training state, usually includes parameter state and optimizer state, to persistent storage, and recomputes the loss process upon a failure.

CheckFreq [19] is among the first attempts to enable snapshot and persistent operations to be pipelined with the original training process. It significantly reduced snapshot cost, but was restricted by host memory bandwidth. To reduce checkpoint cost, Gemini [17] attempts to hide snapshot cost within communication gaps, but risks network contention and may degrade communication performance in parallel settings. JIT-Checkpointing [18] employs a just-in-time approach via a proxy-server architecture to isolate the host from process crashes, preserving the model states. However, its effectiveness is severely constrained by the type of failure (for example, ineffective against proxy-server failures). Despite recent efforts to reduce checkpoint cost, the limited bandwidth for data transfer restricts checkpoint frequency [29]. These approaches can incur prohibitive recovery costs [19] and even risk complete job failure when large-scale errors occur [17, 18].

B. Inconsistent Fault Tolerance

In parallel settings, a global checkpoint involves local checkpoints for each task, performed in synchronous [18] or asynchronous [19, 20] manners. In either way, a degree of global consistency is usually mandatory to complete the global checkpoint. Recent studies have shown that the iterative convergent training process has a self-correcting characteristic that it is capable of tolerating minor perturbations [24–28], inspiring a new trend of inconsistent fault tolerance approaches.

SCAR [26] was the first to demonstrate that minor inconsistencies of parameter states among parallel training tasks do not affect convergence properties or even the trajectory of an iterative training job, providing an important foundation for future studies. Differentiated checkpoint intervals are adopted among parallel tasks, without global consistency guarantees. Failures are restored by local checkpoints, avoiding expensive global rollback and recomputing. CPR [27] further proposed a partial recovery approach and explored the effect of inconsistent model states on the training accuracy empirically.

An effective quantitative model describes how the perturbation affects the training accuracy is missing. This makes inconsistent approaches difficult to control [27] and not easily accepted [26] by real-world applications.

C. Parallelism Techniques

A range of parallel schemes [6–8, 12, 30] have been proposed to deal with rapidly increasing dataset volumes and model sizes. Multiple orthogonal schemes can be used in a hybrid manner [6, 11, 12] to facilitate large model training in distributed environments. Specifically, two parallel schemes are important to implement the proposed CPSR.

Data Parallelism (DP) launches multiple replica instances (worker tasks) across devices. Each worker consumes a subset of data and participates in periodic global parameter updates, usually implemented by collective communication (e.g., AllReduce [31]), to ensure model convergence. Model

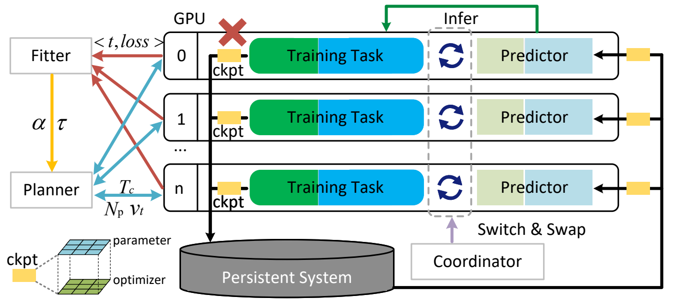


Figure 3. CPSR framework.

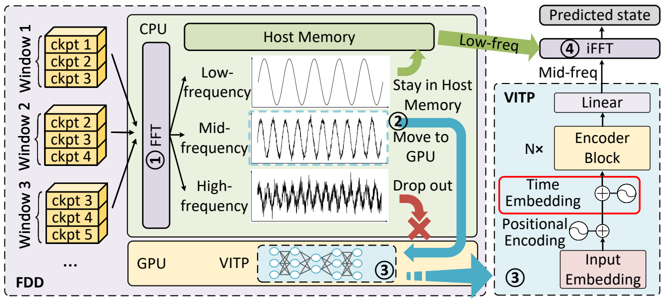


Figure 4. Predictor design.

state redundancy is introduced across workers within a DP group. This provides an opportunity to reduce the overhead of the proposed predictor. The prediction workload can be shared among predictor instances that reside with each original training task in a DP group, as examples in Section III-B.

Pipeline Model Parallelism (PMP) splits model layers into pipeline stages (tasks) distributed across multiple devices, but suffers from pipeline bubbles (idle time). GPipe [7] introduces micro-batching to improve training performance. DAPPLE [8] adopts the One Forward-One Backward schedule to reduce the memory footprint of activations. Inevitable bubbles in PMP provide an opportunity for a device to host additional tasks. Assisted by recent fast-switching approaches [10, 30, 32], it is possible to run additional lightweight training tasks without strongly interfering with the original training task.

III. CPSR Design

We propose a novel failure recovery framework, CPSR, to support fast recovery for large model training in parallel and distributed environments. Fig. 3 shows the overall design of CPSR. It consists of four modules, *predictor*, *coordinator*, *fitter*, and *planner*. The predictor is trained in run-time on the same device of each target training task, with periodic checkpoints. It infers the training state (parameter state and optimizer state) based on the last checkpoint to restart an interrupted task upon failure. The coordinator manages the device memory and the execution flows of the original training task and the predictor training task to reduce the predictor overhead. The fitter collects validation loss values of a training job over time and dynamically characterizes the properties (coefficients) of the loss function curve, indicating the convergence trajectory of the training.

A. Predictor

The predictor is designed to achieve both lightweight and precision. As shown in Fig. 4, the predictor consists of two core components: FFT-based Data Denoising (FDD) and Variable-Interval Temporal Predictor (VITP).

To avoid high-overhead training with large checkpoint samples (for example, the full OPT-175B model checkpoint can reach 2.5 TB), we propose a data denoising method FDD. It first applies a Fast Fourier Transform (FFT) to the checkpoint data sequence, mapping time-domain parameters to the frequency domain to separate components. High-frequency data (noise perturbations) are filtered out to reduce overhead, low-frequency data (slow-changing baseline states) are stored as baselines, and only mid-frequency data (stable evolutionary patterns) that account for 40% of the total volume are fed to the main DNN of the predictor. The proposed FDD procedure significantly reduces the volume of training data, while preserving prediction accuracy.

The convergence rate of a training job, denoted by (v), generally exhibits a superlinear decrease over time. As a result, the training process can be divided into non-uniform phases with different convergence behaviors, according to v. This requires the predictor to have the ability to predict variable-interval time series. Traditional Transformer-based time series prediction models [33] demonstrate strong performance in fixed-interval scenarios, but cannot be directly applied to variable-interval settings. We design a Transformers-based model to handle irregular temporal spacing. VITP introduces a dynamic time embedding layer that encodes varying checkpoint intervals into the latent space. The time embeddings are additively combined with input features before position embedding, injecting temporal awareness.

Eventually, upon failure, the time domain prediction state from VITP is constructed with the resident low-frequency data from FDD via inverse FFT (iFFT) to form the prediction state, maintaining both short-term dynamics and long-term trends.

B. Coordinator

CPSR deploys a lightweight local predictor on each GPU, in charge of boosting the recovery of the local training task. The coordinator is designed to accommodate the predictor training task, along with the original training task, sharing time and space. The coordinator obtains the pipeline topology before training and applies fast task switching [30] to hide the predictor training task in the bubbles (outlined zone in Fig. 5-A of the original training pipeline. We demonstrate the effectiveness of the coordinator in Section VI-C3.

Hosting multiple training tasks in device memory can be difficult, as the volume of models is increasing rapidly [1, 3]. Recent studies on fast task switching [30, 32] utilize host memory as a shared warehouse of model states to increase concurrency among tasks, but frequent memory swapping [34] may still cause contention for the GPU data transfer bandwidth. We propose a lightweight predictor with a small memory footprint (<200MB) that is practical to reside in GPU memory with the original training task. The predictor only saves its local parameter and optimizer states after each

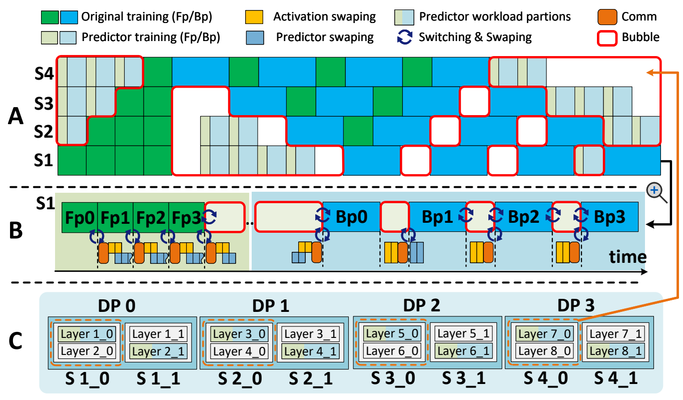


Figure 5. Coordinator design.

iteration, incurring negligible overhead. In circumstance where GPU memory is insufficient, the coordinator will offload the activations of the original training task to host memory after the forward pass and reload them before the backward pass. The activation offloading operations are paired with predictor uploads to overlap with inter-stage communications in PMP to reduce swapping overhead, as shown in Fig. 5-B. The data is further partitioned into smaller chunks, enabling fine-grained scheduling to maximize utilization of idle bandwidth.

Moreover, with the application of DP, the redundancy of model states in a DP group makes it possible to further divide the prediction workload. As illustrated in Fig. 5-C, a DP group of 2 tasks has 2 predictors, each is in charge of half of the total prediction workload. This will further reduce the predictor overhead, as demonstrated in Section VI-C3.

C. Fitter

The loss function curve is an important metric for assessing training convergence trends [35, 36]. It can usually be fitted online [37]. At the end of each training phase, the fitter is launched to approximate the loss descent trajectory (coefficients). Compared with considering all previous phases, we found that focusing on the recent time spans fits the true curve more precisely. The reason is twofold, 1) mainstream DNN optimizers [38] introduce stochasticity in gradient updates that alter the training trajectory, 2) certain learning rate schedulers [39, 40] tend to use non-uniform learning rates in different phases of the training process. We demonstrate the performance of the fitter in Fig. 12. The loss curve illustrates the relationship between training progress and training time, which helped to inspire the modeling of rehabilitation cost.

D. Planner

The CPSR does not enforce uniform T_c among tasks, since each task can have different parameter volumes that lead to different checkpoint and recovery costs. As the inconsistent fault tolerance approaches [26, 27], the planer computes optimal checkpoints for each task separately. Moreover, CPSR does not require a uniform T_c throughout the training process. The training state generally becomes stable when a model is near convergence. Thus, it becomes easier to predict the failed training state and recover when the training is in its late training phases than in its initial phases. Since T_c is independent of the convergence rate v, in practice, instead of computing an optimal T_c for the entire training process based on average v, the planner calculates the optimal T_c according to the proposed solution (Section V-C) for each separate phase.

TABLE I: Notations

Notation	Description	Notation	Description
T_s	checkpoint cost	T_c	checkpoint interval
T_r	recomputing-based recovery cost	T_p	predicting-based recovery cost
T_l	fault tolerance cost	λ	failure occurrence rate
i	checkpoint index	t	iteration index
N_p	number of parameters to be recovered	v_t	second-order moment of gradient at iteration t
ΔF	prediction deviation	ξ_t	expectation of the 2nd-order derivative
L	smoothness coefficient	μ	strong convexity coefficient
α, τ	coefficients for loss curve functions		

IV. Checkpoint Interval Problem Formulation

One key challenge in CPSR is how to decide the optimal T_c for each training task of a job. This section first briefly reviews the RB-R fault tolerance cost model and then proposes the PB-R cost model, based on which a novel checkpoint interval problem is introduced.

A. Recomputing-Based Recovery

The periodic checkpoint in run-time saves training states. When a failure occurs, the training states are rolled back to the last checkpoint to avoid a restart from the very beginning. As shown in Fig. 6, each checkpoint operation involves a cost of T_s, considering both snapshot and persistent, which is usually not a straightforward combination. T_r is the recovery cost (recomputing) to get the training states back before failure. T_c decides the maximum T_r. Frequent checkpoints lead to a high checkpoint cost, including execution stall, while infrequent checkpoints may cause a longer recomputing time. The trade-off of checkpoint and recovery cost is an essential issue that has been studied in different settings [17, 19, 20, 26, 29, 33, 41]. More notations are listed in Table I.

B. Predicting-Based Recovery Problem

PB-R method also relies on periodic checkpoints to save training states. The proposed predictor is trained in run-time with the checkpoints. Upon a failure, the predictor uses recent checkpoints as input to infer a training state that is close to that before the failure. This gives the recovery a start-up boost, at very little cost that can be omitted. The deviation between the predicted and the true training state is defined as *prediction deviation* (denoted by ΔF). It shows as a minor perturbation [26] of the model training process that can self-correct over time. The PB-R cost T_p is defined as the additional training required for a model to be rehabilitated, as shown in Fig. 6. It is inversely correlated with the accuracy of the predictor. A comprehensive model of T_p is needed to guide the control of the PB-R process.

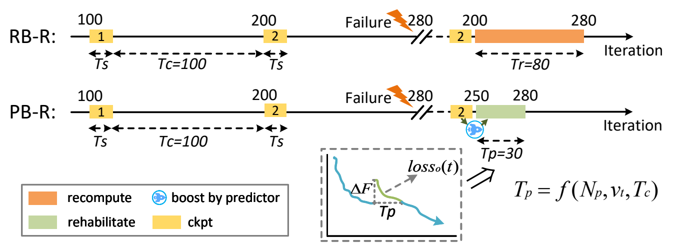


Figure 6. Fault tolerant cost modeling.

The PB-R cost T_p is influenced by three factors, the amount of model parameters affected by a failure, the speed of convergence at the moment and the distance to the projected destination. The number of parameters to be predicted (N_p) can be obtained based on the partition of the model. For example, N_p equals the number of parameters in an interrupted PMP stage (task). We use the loss curve, fit by the fitter in Section III-C, as an indication of the training convergence rate. The projected distance is constrained by T_c. Detailed modeling of T_p is proposed in Section V-B.

The richness of the dataset plays a decisive role in the generalization of the DNN model [2]. Similarly, for predictor training, we argue that appropriately increasing the sample size (number of checkpoints) can improve its accuracy, as empirically validated in our test (Fig. 7 and Fig. 8 in Section VI-A). A high checkpoint frequency can affect the training process, causing a higher checkpoint cost. However, a low checkpoint frequency can jeopardize the accuracy of the predictor and increase the difficulty of self-recovery or even make it infeasible. T_c is a critical controller for predicting-assisted self-recovery. This paper introduces a novel checkpoint interval problem to trade off the checkpoint and recovery cost.

Definition 1. Predicting-based Checkpoint Interval Problem (PCIP). *Given failure rate λ, checkpoint cost T_s, the amount of parameter affected by a failure N_p, and the second-order moment of the gradient v of a model, find a checkpoint interval T_c with minimum fault tolerance cost T_l.*

$$minimum \qquad T_l = \sum_{i=0}^{\infty}(iT_s + T_p)P(t)d_t \tag{1}$$

$$s.t. \qquad T_p = f(N_p, v_t, T_c) \tag{2}$$

$$P(t) = \lambda e^{-\lambda t}, \lambda = 1/T_f \tag{3}$$

$$F(y) \leq F(x) + \nabla(x)^T(y-x) + L\|y-x\|^2/2 \tag{4}$$

$$F(y) \geq F(x) + \nabla(x)^T(y-x) + \mu\|y-x\|^2/2 \tag{5}$$

The fault tolerance cost T_l consists of the total checkpoint cost and the accumulated PB-R cost for random failures. We assume a widely applied failure scenario in the literature that the occurrence of failures is random (a Poisson process) with a failure rate λ [37, 41], whose inter-arrival time t follows the density function given by Eq. 3. Eq. 4 and Eq. 5 express the smoothness [42] and strong convexity [43] of the DNN objective function F, where $x, y \in \mathbb{R}^d$.

C. Assumptions

In this paper, we make the following assumptions. First, failures are assumed to follow a Poisson distribution (Eq. 3), aligning with the literature [37, 41]. Second, the minor perturbation caused by the prediction deviation does not affect the original model convergence property and trajectory, which is theocratically proved in [26] and empirically validated in our experiment (Section VI-C1). Finally, we assume that the loss function of a model training job can be fitted [35–37].

V. Approach

The PCIP presents two main challenges. How to quantify the PB-R cost T_p? How to compute optimal checkpoint interval for each training phase with a small overhead? First, Section V-A proposes a model of the prediction deviation introduced by the predictor. Then the PB-R cost model T_p is presented in Section V-B, reflecting the additional training time required to eliminate the prediction deviation. Finally, Section V-C presents a solution for PCIP.

A. Prediction Deviation

The prediction deviation ΔF represents the difference in model quality, indicated by the variation of the loss value, between the predicted and the true training state. The *Smoothness* and *Strong Convexity* of the DNN objective functions (Eq. 4 and Eq. 5) provide the upper and lower bounds of ΔF caused by parameter perturbations (δx), as $\nabla(x)^T\delta x + \frac{\mu}{2}\|\delta x\|^2 \leq \Delta F \leq \nabla(x)^T\delta x + \frac{L}{2}\|\delta x\|^2$. L and μ correspond to the upper and lower bounds of the eigenvalue of the Hessian matrix [42, 43] of the function F. From the second-order Taylor expansion, there exists some $\theta \in [0, 1]$, $\Delta F = \nabla F(x)^\top \delta x + \frac{1}{2}\delta x^\top \nabla^2 F(x + \theta\delta x)$. In practice, a scalar curvature estimate ξ_t is often used to approximate the second-order term. For any δx, there exists $\xi_t \in [\mu, L]$ such that

$$\Delta F \;\approx\; \nabla(x)^T\delta x + \xi_t\|\delta x\|^2/2, \tag{6}$$

where ξ_t denotes the estimated average curvature along the update direction at iteration t.

The second moment of the gradient is commonly employed to approximate the Hessian diagonal based on the Lipschitz continuity assumption, i.e. $\mathbb{E}[g_t^2] \approx diag(\nabla^2 F(\theta))$, due to the high time complexity to compute second-order derivatives. We use the Exponentially Weighted Moving Average (EWMV) of squared gradients as an online estimator [38] that $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t{}^2$. An approximate form of ξ_t is as follows:

$$\xi_t = \sum_{j=1}^{N_p}[\beta_2 v_{t-1,j} + (1-\beta_2)g_{t,j}{}^2]/N_p, \tag{7}$$

where $v_{t,i}$ can be obtained from the optimizer state [38].

The proposed predictor is a time-series predicting model fed by checkpoints. Its accuracy is affected by the sampling interval (checkpoint interval). The projected difficulty of the prediction is positively correlated with T_c, as demonstrated later in Fig. 7. We establish the relationship between the accuracy of the predictor (indicated by its validation loss value) and the checkpoint interval as

$$loss_p(t) = e^{-\tau_1\frac{t}{T_c}+\tau_2}, \tag{8}$$

where τ_1 and τ_2 are coefficients of the loss function that can be fitted online [37].

Meanwhile, the predictor regressively learns the evolution pattern of parameter states and optimizer states over time. The loss function for regression problems [44–46] usually adopts the Mean Squared Error (MSE) as

$$loss_p(t) = \sum\nolimits_{j}^{N_p} (x_{j,t} - y_{j,t})^2 / N_p. \tag{9}$$

With Eq. 8, Eq. 9 and Eq. 6, we derive the relationship from T_c through δ_x to ΔF. The first term in Eq. 6 is omitted because it is usually much smaller than the second one (over 140 times in our experiment training GPT3-3.35B). The prediction deviation ΔF is related to T_c as

$$\Delta F = \xi N_p e^{-\tau_1 \frac{t}{T_c} + \tau_2} / 2. \tag{10}$$

B. Predicting-Based Recovery Cost

The prediction deviation incurs a minor perturbation that needed to be self-corrected by the training process with extra training time, introducing recovery cost T_p. We first construct a mapping of the prediction deviation ΔF and the estimated recovery time based on the trajectory of the validation loss value of the original training job. The loss value serves as a nonnegative metric to track training progress and assess model quality. Its function outlines the training progress over time and generally exhibits convexity and monotonic descent [35–37]. Research has shown that neural network algorithms exhibit linear or superlinear convergence rates of $O(\mu^k)$ [35]. Thus, we model the loss function as an exponential function,

$$loss_o(t) = e^{-\alpha_1 t + \alpha_2}, \tag{11}$$

where the coefficients α_1 and α_2 can be fitted by the fitter proposed in Section III-C. α_1 controls the curvature of the loss curve, while the position of the curve is controlled by α_2.

Furthermore, we develop a cost model to characterize the temporal degradation caused by perturbations, compared with the failure-free scenario. Suppose the failure-free convergent parameter state is x^*, and the parameter state at the k-th iteration is x_k. If the condition $F(x_k) - F(x^*) < \epsilon$ is satisfied, where F is the objective function and ϵ corresponds to the prescribed convergence tolerance, then x_k is considered to have converged. Suppose a predicting-based recovery occurs at time t, causing a loss variation ΔF. After recovery, the training converged at iteration k', and the loss function satisfies

$$loss_o(k') = \underbrace{[(e^{-\alpha_1})^t \cdot e^{\alpha_2} + \Delta F]}_{predicting-assisted} (e^{-\alpha_1})^{k'-t}. \tag{12}$$

The predicting-assisted term represents the training iterations before t and the prediction deviation caused by the prediction on restart. The rest of Eq. 12 describes the training process after the predicted boost start. Although it does not guarantee an identical convergence state, compared with that of failure-free training, our experimental results (Fig. 8) indicate that decent predictions (accurate to 10^{-4}) are sufficient to preserve the original convergence behavior. This aligns with recent studies on parameter quantification [25]. Therefore, the recovery cost model is

$$T_p = k' - k = ln[1 + e^{\alpha_1 t_e - \alpha_2} \cdot \Delta F] / \alpha_1. \tag{13}$$

Since $e^{\alpha_1 t_e - \alpha_2} \cdot \Delta F \in (0, 1)$, and it is typically a small value, applying the first-order Taylor approximation $ln(1+x) \approx x$ for small x, we derive the explicit form of Eq. 2 as

$$T_p = e^{\alpha_1 t_e - \alpha_2} \cdot (\xi N_p e^{-\tau_1 \frac{t}{T_c} + \tau_2}) / 2\alpha_1. \tag{14}$$

C. Optimal Checkpoint Scheme

Based on the PB-R cost T_c, we propose an optimal checkpoint scheme to minimize the fault tolerance cost in the training process. Combining with Eq. 14, we further formulate the objective of the PCIP as

$$T_l = \sum\nolimits_{i=0}^{\infty} \int_{i(T_c+T_s)}^{(i+1)(T_c+T_s)} (iT_s + \tfrac{e^{\alpha_1 t - \alpha_2} \cdot \Delta F}{\alpha_1}) \lambda e^{-\lambda t} d_t. \tag{15}$$

In practice [22], v undergoes only minor variations in a short period of time. Therefore, a training phase is divided when the accumulated variation of v exceeds a threshold. During each training phase, the temporal effects on ξ are ignored. At the start of each phase, ξ is approximated as a piecewise constant.

Theorem 1. *An optimal checkpoint interval that minimizes the cost of fault tolerance in the PCIP can be calculated as*

$$T_c = \frac{\tau_1}{B + \lambda\sqrt{A/T_s}}, \tag{16}$$

where $A = \xi N_p \tau_1 e^{\tau_2 - \alpha_2} / 2\alpha_1$ *and* $B = \alpha_1 - \lambda$.

Proof. By integrating and adding the resulting geometric series on Eq. 15, we obtain

$$T_l = \frac{Ts}{1 - e^{-\lambda(T_c+T_s)}} - \frac{\lambda \xi N_p e^{\tau_2 - \alpha_2}}{2\alpha_1 \theta} \cdot [1 - e^{n\theta(T_c+T_s)}], \tag{17}$$

where $\theta = \alpha_1 - \frac{\tau_1}{T_c} - \lambda$ and n represents the checkpoint numbers that is typically a large value.

Next we show that $1 - e^{n\theta(T_c+T_s)}$ converge to 1 as n increases. It can be divided into three terms according to θ. (1) $e^{n\alpha_1(T_c+T_s)}$ can be expressed as $e^{\alpha_2}/e^{-\alpha_1 n(T_c+T_s)+\alpha_2}$, which represents the ratio of maximum (initial) to minimum (convergence) loss values e^{α_2}/ϵ that is independent of T_c. (2) The overall trend of the loss function is monotonically decreasing ($\tau_1 > 0$), thus $e^{-n\tau_1/T_c(T_c+T_s)}$ is in [0,1]. (3) $n\lambda(T_c+T_s)$ represents the expectation of the number of failures, which typically ranges from hundreds to thousands in large model training [4], thus $e^{n\lambda(T_c+T_s)}$ dominates $e^{n\theta(T_c+T_s)}$. Because $1 - e^{n\lambda(T_c+T_s)}$ convergences to 1 as n increases, we simplify T_l by canceling out $1 - e^{n\theta(T_c+T_s)}$, and seek the minimum T_l by taking the derivative of T_l with respect to T_c as

$$\frac{d_{T_l}}{d_{T_c}} = \frac{-\lambda T_s e^{-\lambda(T_c+T_s)}}{[1 - e^{-\lambda(T_c+T_s)}]^2} + \frac{\lambda \xi N_p e^{\tau_2 - \alpha_2} \tau_1}{2\alpha_1 \theta^2 {T_c}^2} = 0. \tag{18}$$

In practice $T_s, T_c \ll Tf$, by approximating $1 - e^{-\lambda(T_c+T_s)}$ to $-\lambda(T_c+T_s)$, we obtain $T_s/[\lambda(T_c+T_s)]^2 = A/(BT_c - \tau_1)^2$, where $A = \xi N_p \tau_1 e^{\tau_2 - \alpha_2}/2\alpha_1$ and $B = \alpha_1 - \lambda$. □

VI. Evaluation

A. Resuming Training from a Failure

CPSR[1] is implemented as a plug-in module for PyTorch. The checkpoint module is developed based on the Checkfreq

[1]https://github.com/wangzc-HPC/CPSR.git

TABLE II: Evaluation workload

Model	Dataset	# Layer	# Hidden size	# of Params
VGG16	ImageNet	16	_	144M
GPT2-0.35B	CodeSearch	24	1024	350M
GPT3-3.35B	CodeSearch	16	4096	3.35B

snapshot [19]. No user intervention is required to apply CPSR, except for checkpoint storage configurations.

CPSR periodically monitors tasks' run-time status. Upon device-level failure, e.g., GPU-side training task failures, CPSR first restores the interrupted task based on the last checkpoint, then uses predictor to infer a training state as a boost-start, followed by the self-recovery process. For parallel training jobs, one key technical challenge is maintaining the communication context among tasks in the same job. We adopt a similar method of communication proxy [18] to maintain communication links, buffers, and message consistency, avoiding cascading errors caused by communication timeout. Upon node-level failure, e.g. multiple GPU-side failures and/or CPU-side failure, CPSR recovers each task independently, and reboots proxies and reestablish links if needed.

B. Experimental Setup

We conducted comprehensive testbed experiments to evaluate the performance of CPSR. The results show that CPSR achieves 1.46× training throughput over state-of-the-art failure recovery approaches on average.

1) System Setups: The testbed consists of two platforms. Platform A has four servers with Intel XEON 6330 CPUs×2 and NVIDIA A30 GPU (24GB)×2 each, and platform B has one server with Intel XEON 8480+ CPU×2 and NVIDIA H100 GPU (80GB)×8. All servers use 12GB/s (PCIe4.0) GPU-to-CPU bandwidth. The CPU-to-storage bandwidths for the A30 server and the H100 server are 250MB/s and 2GB/S PCIe4.0, respectively. All servers run on 64-bit Ubuntu 20.04.5 LTS with CUDA toolkit V12.4, PyTorch 1.11.1, and NCCL v2.19.4. We use platform A for most of our tests, because it represents a typical distributed training environment, and it has relatively small GPU memory, which helps to demonstrate the effectiveness of CPSR under high GPU memory contentions. We combine both platforms to form a 16 GPU environment for our large-scale experiments in parallel training scenarios.

2) Baselines: We compare the proposed CPSR with three state-of-the-art checkpoint approaches: Pytorch checkpoint API [22], CheckFreq [19], and PCcheck [20]. The Pytorch checkpoint API is currently the first choice for many deep learning practitioners. CheckFreq is the first to enable pipelining snapshot and training operations, which significantly reduced the snapshot time. PCcheck further reduced snapshot cost through memory pinning, while enabling concurrent checkpointing to reduce persistent costs. Furthermore, changing the checkpoint interval T_c would have an impact on the recovery cost in RB-R methods. We designed ablation experiments forcing all approaches to use a uniform T_c in each case, eliminating T_c variance in recovery cost measurements. CPSR_FIX represents the approach of CPSR with fixed T_c.

3) Workloads: We evaluated CPSR with popular and representative large DNN models, including GPT-3, GPT-2 and VGG16. The number of layers, hidden sizes, and intermediate sizes of the workloads are tuned to fit our testbed. Table II shows the detailed configurations. As in most related work, the Adam [38] optimizer is used for all training tests.

C. Data and Analysis

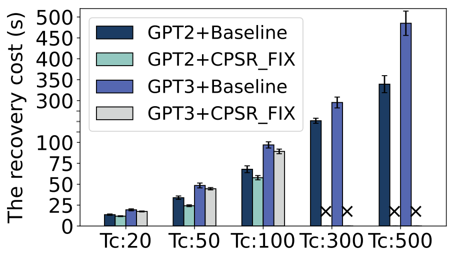
Based on empirical data from real-world training workloads [4], we set the system failure rate λ at 0.08 failure/hour (one failure every 12 hours). We demonstrate the fault tolerance performance of the CPSR in five aspects.

1) ***Recovery Cost:*** This experiment illustrates the recovery cost of CPSR. Since the PB-R cost is related to three variables, T_c, N_p and v, to evaluate the recovery performance of different checkpoint methods, we conducted experiments in three dimensions: checkpoint intervals, model parameter volumes and failure occurrence phases. All baselines are RB-R approaches that are based on recomputing to recover. Thus, we use vanilla as the representative of the RB-R approaches. Fig. 7 shows the recovery cost comparisons of different approaches. Since the other three approaches share an identical recovery mechanism, their recovery performance in Fig. 7 and 8 is collectively represented by baseline.

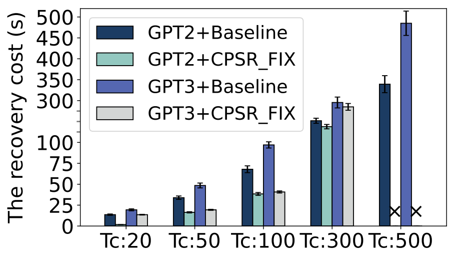
The first deciding factor for recovery performance is T_c, which also indicates the maximum recomputing cost. All RB-R approaches tend to choose small T_c to avoid a high recomputing cost, which is not usually feasible in practice. As illustrated, CPSR_FIX outperforms baselines for large T_c (20-100 training iterations) by 43%. This is because CPSR_FIX boosted the rehabilitation to recover without actual recomputation. In this T_c rage, it is possible to provide a decent sized dataset for the predictor. However, we also notice that when we push the limit and set T_c to ultra-large (300-500 iterations), CPSR_FIX fails to fully recover the model training process in some cases. The reason is twofold, 1) insufficient checkpoints brought the predicting accuracy of the predictor down; 2) significant prediction deviation makes it difficult for the model training to self-correct.

The second factor is the volume of the model parameters. We used two similar workloads (GPT2-0.35B and GPT3-3.35B) with different levels of parameter volumes (350 million and 3.35 billion). As illustrated, comparing results with the same T_c settings, CPSR_FIX performs better for the smaller model than for the larger model, by 21% on average. This is because a larger parameter volume N_p can cause a larger prediction deviation of the predictor. However, it is important to note that reducing T_c, leading to a larger sample size, also helps to increase the accuracy of the predictor. For example, the recovery cost of the GPT-3 model with $T_c = 50$ is 42% less than that of the GPT-2 model with $T_c = 100$.

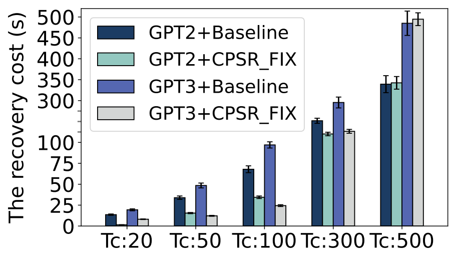
The third factor is the failure occurrence point. We highlight failures in three stages of the training lifecycle, the beginning stage (25% progress), the middle stage (50% progress), and the late stage (75% progress). CPSR_FIX reduces the recovery cost by 18.6%, 50.8%, and 55.4% on average, at different



(a) Failure at 25% training progress.



(b) Failure at 50% training progress.



(c) Failure at 75% training progress.

Figure 7. Recovery cost comparisons (training GPT2-0.35B).

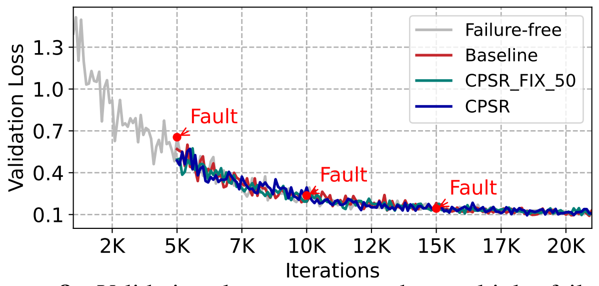


Figure 8. Validation loss curves under multiple failures. "CPSR_FIX_50" indicates using CPSR with a fixed checkpoint interval of 50 iterations.

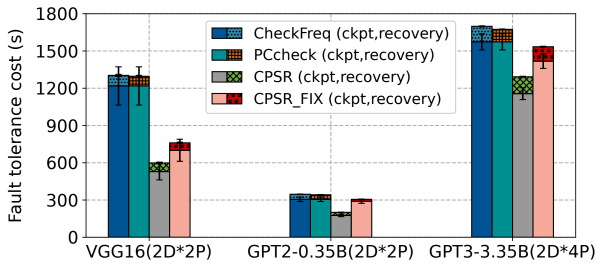


Figure 9. Fault tolerance cost comparisons. "D" and "P" indicate parallel degree of DP and PMP, respectively. 2D*2P means 2 DP groups and 2 PMP states.

stages, respectively. It performs better in the middle and late stages than at the beginning of training. With the progression of training, the predictor gradually converges (around 800 iterations). Meanwhile, the parameter states of the training model become increasingly stable as the training progresses. This reduces the prediction deviation in CPSR_FIX and, in turn, reduces the recovery cost.

Please note that the upper limit of feasible T_c in CPSR increases as training progresses and training states become stable. This is another reason why CPSR chooses the optimal T_c for different training phases separately, rather than using a uniform T_c throughout the training process.

Fig. 8 presents the loss reduction trajectories during the recovery process for different checkpoint approaches training GPT2-0.35B. It indicates that after failure, the model instance recovered by CPSR has a better model quality (lower loss value) than the baseline in 58.9% of the time.

In summary, CPSR supports large T_c and large models with billion-level parameters, aligned with current application scenarios [20]. It performs better in the middle and late stages of a training than in its early stage. It reduces the recovery cost by 41.66% on average, compared with baselines.

*2) **Total Fault tolerance Cost:*** Fig. 9 illustrates the total fault tolerance cost comparisons. CPSR achieves the lowest fault tolerance cost compared with the baselines (37.5%, and 36.9% lower than CheckFreq and PCcheck, respectively).

The checkpoint interval settings are critical to the overall cost of fault tolerance. The proposed CPSR can calculate an optimal checkpoint interval without user input. However, the baselines, CheckFreq and PCcheck all require user inputs. PCcheck requires users to specify the checkpoint cost control parameter (denoted q in [20]). CheckFreq requires a user-specified checkpoint cost, e.g., 3.5% of the Job Completion Time (JCT) in [19]. We found that this setting triggered a high checkpoint frequency, exceeding the CPU-to-storage bandwidth capacity in our testbed, and causing severe training process blockage. Give enough data bandwidth, the baselines prefer high checkpoint frequency to reduce recomputing cost. Therefore, we configure the checkpoint cost of PCcheck and CheckFreq to 2% of the JCT, within the data transfer bandwidth limit, so the checkpoint frequency is the highest and concurrent persistent operations will not cause blockage of the training process.

CPSR reduces recovery cost by eliminating recomputing. It has a better recovery ability that does not require a high checkpoint frequency as a baseline, as shown in Table III. Meanwhile, the optimizer states tend to be more stable in the latter phases of the training process than in the early phases. CPSR is able to capture this momentum (v) to reduce the checkpoint frequency in the latter phases. CPSR maintains a lower checkpoint cost, while incurring almost the same recovery cost as CPSR-FIX. This advantage is more prominent in long training durations.

Moreover, it is worth mentioning that the recovery costs of all approaches are more stable than checkpoint costs; this is because checkpoint costs can be affected by bandwidth contention in both snapshot and persistent procedures. As shown by the error bars in Fig. 9, the recovery cost of CPSR is as stable as that of recomputing-based approaches.

*3) **Parallel Training Scenario:*** We employ the 1F1B pipeline scheduling strategy for PMP training. It is a widely applied scheme [8, 30] for large model training. The number

TABLE III: Checkpoint interval T_c and JCT comparisons.

Model	CheckFreq	PCcheck	CPSR	JCT
VGG16	23	21	53	61h10min
GPT2-0.35B	19	17	33	7h41min
GPT3-3.35B	36	32	49	10h55min

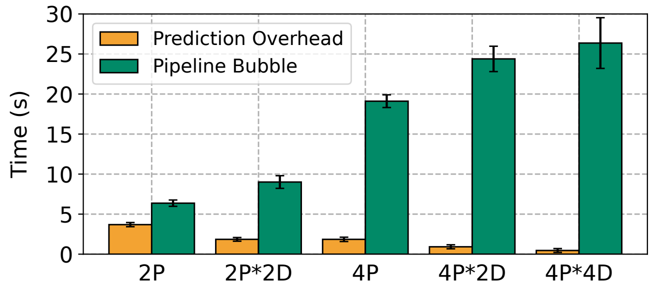


Figure 10. Training overhead of predictor and pipeline bubble between consecutive checkpoints.

of micro-batches is set to the number of pipeline stages.

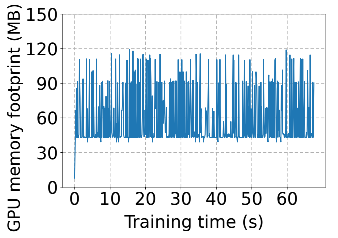
Fig. 10 illustrates the overhead of training the predictor, compared with pipeline bubbles between consecutive checkpoints, training the GPT2-0.35B model. It shows that the overhead of the predictor is small enough that it can be hidden in the pipeline bubbles of the original model training process. As the number of pipeline stages increases, more bubbles occurred. In the meantime, with the parallel degree of DP increased, the overhead of training the predictor is divided in a DP group, therefore the per-task overhead decreases. This makes it easier to overlap the overhead with the pipeline bubbles. As a result, CPSR is suitable for large-scale parallel training with PMP and/or DP, aligning with the current trend.

*4) **Memory Utilization and Switching Cost:*** CPSR ships with a lightweight predictor that has small memory footprints and is more likely to reside on the same device as the original training task without memory swapping. We divide multi-dimensional tensors into samples according to the present token size in a serialized manner.

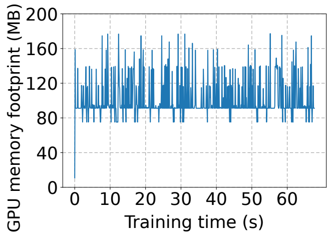
Fig. 11 shows the GPU memory footprint of the predictor during GPT2-0.35B and GPT3-3.35B model training. We tested different token size configurations and get similar results of less than 200MB of GPU memory. We also considered scenarios where GPU memory is strictly limited and the predictor state need to be swapped out of the GPU memory to make space for the original training task. The activations swapping time for token size of 1024 is 0.011 second, much smaller than the forward propagation (0.151 seconds) or the back propagation (average 0.237 seconds) of GPT2-0.35B. For token size 4096, swapping takes 0.018 seconds and is still far lower than the forward (0.245 seconds) or backward propagation (1.552 seconds) for GPT3-3.35B. It is possible to hide the swapping overhead in communication idle time.

As a result, the proposed predictor is a lightweight model with small memory footprint (less than 200MB) that can reside in GPU memory with the original training task, or quickly swapped to host memory with little overhead.

*5) **Loss Curve Fitting:*** Fig. 12 demonstrate the performance of the fitter with two examples (GPT2 and VGG16

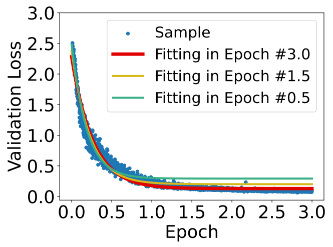


(a) Token size of 1024 (GPT2-0.35B).

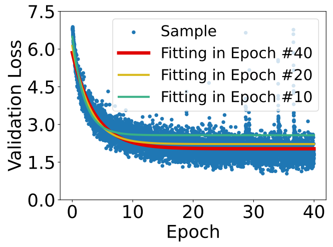


(b) Token size of 4096 (GPT3-3.35B).

Figure 11. GPU memory footprints of the predictor.



(a) Fitted loss function of GPT2-0.35B.



(b) Fitted loss function of VGG16.

Figure 12. Loss curve fitting.

models). The fitter is launched at the end of each training phase, when the accumulated variation of the convergence rate (v) exceeds the threshold of 5%, around 1000 and 2000 iterations in the experiments, respectively. As illustrated, the fitter can accurately capture the trajectory of the loss function online, with RMSE of 0.0824 and 0.3749, respectively. Note that the fitting result may vary because the optimizer continuously adjusts its optimization direction to reduce the loss.

VII. Conclusion

This paper revisits the failure recovery of large model training from a novel perspective. Instead of recomputing the lost process from the last checkpoint, we propose a predicting-based recovery framework, CPSR, that takes advantage of the self-correcting characteristic of the DNN training process. It features a predictor that is located on the same device as the original training task and fed by routine checkpoints in run-time. Upon a failure, the predictor predicts a training state close to before the failure to give the recovery a boost, reducing the recovery cost. We introduce a novel optimal checkpoint interval problem to guide predicting-assisted self-recovery and propose a solution. Extensive testbed data are presented to demonstrate the performance of CPSR. It reduces the recovery cost by 41.66% on average, taking 39% fewer checkpoints compared with state-of-the-art approaches, and causing a small memory footprint of less than 200MB.

VIII. Acknowledgment

This work was supported by the Natural Science Foundation of Jilin Province (Grant 20230101062JC), and the National Natural Science Foundation of China (Grant 62272190).

References

[1] J. Achiam, Adler *et al.*, "Gpt-4 technical report," *ArXiv*, 2023.

[2] C. Schuhmann, R. Beaumont, R. Vencu *et al.*, "Laion-5b: An open large-scale dataset for training next generation image-text models," *Advances in neural information processing systems (NeurIPS)*, 2022.

[3] A. Chowdhery, Narang *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research (JMLR)*, 2023.

[4] S. Zhang, S. Roller, N. Goyal *et al.*, "Opt: Open pre-trained transformer language models," *ArXiv*, 2022.

[5] A. Liu, B. Feng *et al.*, "Deepseek-v3 technical report," *ArXiv*, 2024.

[6] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[7] Y. Huang, Y. Cheng, A. Bapna *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems (NeurIPS)*, 2019.

[8] S. Fan, Y. Rong, C. Meng *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPopp)*, 2021.

[9] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[10] H. Zhao, Q. Tian, H. Li, and Z. Chen, "Flexpipe: Maximizing training efficiency for transformer-based models with variable-length inputs," in *USENIX Annual Technical Conference (ATC)*, 2025.

[11] D. Narayanan, M. Shoeybi, J. Casper *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC)*, 2021.

[12] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *ArXiv*, 2019.

[13] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," *ArXiv*, 2021.

[14] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research (JMLR)*, 2022.

[15] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, "Fastmoe: A fast mixture-of-expert training system," *ArXiv*, 2021.

[16] LlamaTeam, "The llama 3 herd of models." [Online]. Available: https://ai.meta.com/research/publications/the-llama-3-herd-of-models

[17] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. E. Ng, and Y. Wang, "Gemini: Fast failure recovery in distributed training with in-memory checkpoints," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2023.

[18] T. Gupta, Krishnan *et al.*, "Just-in-time checkpointing: Low cost error recovery from deep learning training failures," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2024.

[19] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained dnn checkpointing," in *USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[20] F. Strati, M. Friedman, and A. Klimovic, "Pccheck: Persistent concurrent checkpointing for ml," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[21] B. Workshop, T. L. Scao, A. Fan *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *ArXiv*, 2022.

[22] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems (NeurIPS)*, 2019.

[23] J. Rasley, Rajbhandari *et al.*, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of ACM International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, 2020.

[24] A. Agrawal, Reddy *et al.*, "Inshrinkerator: Compressing deep learning training checkpoints via dynamic quantization," in *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2024.

[25] J. Lin, Tang *et al.*, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[26] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *International Conference on Machine Learning (ICML)*, 2019.

[27] K. Maeng, S. Bharuka, I. Gao *et al.*, "Understanding and improving failure tolerant training for deep learning recommendation with partial recovery," *Proceedings of Machine Learning and Systems(MLSys)*, 2021.

[28] Y. Chen, X. Sun, and Y. Jin, "Communication-efficient federated deep learning with layerwise asynchronous model update and temporally weighted aggregation," *IEEE Transactions on Neural Networks and learning systems (TNNLS)*, 2019.

[29] Z. Huang, H. Nie, H. Jia, and others., "Flowcheck: Decoupling checkpointing and training of large-scale models," in *Proceedings of the European Conference on Computer Systems*, 2025.

[30] H. Zhao, H. Li, Q. Tian, J. Wu, M. Zhang, Z. Xu, X. Li, and H. Xu, "Arraypipe: Introducing job-array pipeline parallelism for high throughput model exploration," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2025.

[31] L. Zhang, S. Shi, X. Chu, W. Wang, B. Li, and C. Liu, "Dear: Accelerating distributed deep learning with fine-grained all-reduce pipelining," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2023.

[32] A. Devkota, R. V. W. Putra, and M. Shafique, "Switchmt: An adaptive context switching methodology for scalable multi-task learning in intelligent autonomous agents," *ArXiv*, 2025.

[33] M. Jin, S. Wang, L. Ma *et al.*, "Time-llm: Time series forecasting by reprogramming large language models," *ArXiv*, 2023.

[34] Y. Li, A. Phanishayee, D. Murray, J. Tarnawski, and N. S. Kim, "Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers," in *International Conference on Very Large Data Bases (VLDB)*, 2022.

[35] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *Proceedings of Symposium on Cloud Computing (SoCC)*, 2017.

[36] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys)*, 2018.

[37] H. Li, Z. Wang, H. Zhao, M. Zhang, X. Li, and H. Xu, "Convergence-aware optimal checkpointing for exploratory deep learning training jobs," *Future Generation Computer Systems (FGCS)*, 2025.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ArXiv*, 2014.

[39] R. Ge, S. M. Kakade, R. Kidambi, and P. Netrapalli, "The step decay schedule: A near optimal, geometrically decaying learning rate procedure for least squares," *Advances in neural information processing systems (NeurIPS)*, 2019.

[40] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *ArXiv*, 2016.

[41] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, 1974.

[42] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[43] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM review*, 2018.

[44] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[45] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2018.

[46] G. Zhang, B. E. Patuwo, and M. Y. Hu, "Forecasting with artificial neural networks:: The state of the art," *International journal of forecasting*, vol. 14, no. 1, pp. 35–62, 1998.