CPR: UNDERSTANDING AND IMPROVING FAILURE TOLERANT TRAINING FOR DEEP LEARNING RECOMMENDATION WITH PARTIAL RECOVERY

Kiwan Maeng 1,2, Shivam Bharuka 1, Isabel Gao 1, Mark C. Jeffrey 1, Vikram Saraph 1, Bor-Yiing Su 1, Caroline Trippel 1, Jiyan Yang 1, Mike Rabbat 1, Brandon Lucia 2, Carole-Jean Wu 1

ABSTRACT
The paper proposes and optimizes a partial recovery training system, CPR, for recommendation models. CPR relaxes the consistency requirement by enabling non-failed nodes to proceed without loading checkpoints when a node fails during training, improving failure-related overheads. The paper is the first to the extent of our knowledge to perform a data-driven, in-depth analysis of applying partial recovery to recommendation models and identified a trade-off between accuracy and performance. Motivated by the analysis, we present CPR, a partial recovery training system that can reduce the training time and maintain the desired level of model accuracy by (1) estimating the benefit of partial recovery, (2) selecting an appropriate checkpoint saving interval, and (3) prioritizing to save updates of more frequently accessed parameters. Two variants of CPR, CPR-MFU and CPR-SSU, reduce the checkpoint-related overhead from 8.2–8.5% to 0.53–0.68% compared to full recovery, on a configuration emulating the failure pattern and overhead of a production-scale cluster. While reducing overhead significantly, CPR achieves model quality on par with the more expensive full recovery scheme, training the state-of-the-art recommendation model using Criteo’s Terabyte CTR dataset. Our results also suggest that CPR can speed up training on a real production-scale cluster, without notably degrading the accuracy.

1 INTRODUCTION
Recommendation algorithms form the core of many internet services. For instance, the algorithms enable products that suggest music on Spotify (Jacobson et al., 2016), videos on YouTube and Netflix (Covington et al., 2016; Gomez-Uribe & Hunt, 2015), mobile applications on Google Play-Store (Cheng et al., 2016), stories on Instagram (Medvedev, Ivan and Wu, Haotian and Gordon, Taylor, 2019), commercial products (Smith & Linden, 2017; Chui et al., 2018), or advertisements (Zhao et al., 2020). The impact of recommendation algorithms on user experience is tremendous. Recent studies show that a significant amount of content—60% of the videos on YouTube and 75% of the movies on Netflix that were viewed—come from suggestions made by recommendation algorithms (Chui et al., 2018; Underwood, 2019; Xie et al., 2018b). Thus, the industry devotes significant infrastructure resources to recommendation models—across computing clusters serving a wide variety of machine learning workloads, about 50% of training (Acun et al., 2021) and 80% of inference cycles are dedicated to recommendation at Facebook (Gupta et al., 2020b).

Over the past decades, a plethora of research has been devoted to the development of recommendation algorithms, from classical techniques (Van Meteren & Van Someren, 2000; Sarwar et al., 2001; Koren et al., 2009) to machine learning (ML) (He et al., 2017; Wang et al., 2016; Rendle & Schmidt-Thieme, 2010) and deep learning (DL) (Cheng et al., 2016; Naumov et al., 2019; Zhou et al., 2018; Zhang et al., 2019; Guo et al., 2017; Song et al., 2020; Weinberger et al., 2009; Gupta et al., 2020a). Domain-specific systems tailored to deep learning-based recommendations have also been designed to enable performant and energy-efficient execution (Zhao et al., 2020; 2019; Kalamkar et al., 2020; Nvidia, 2019; 2020b; Jouppi et al., 2017; Nvidia, 2020a; Ke et al., 2020; Hwang et al., 2020; Kwon et al., 2019).

State-of-the-art deep learning recommendation models consist of two major components: Multi-Layer Perceptron (MLP) and embedding layers (Emb), jointly trained to reach a target model quality (Naumov et al., 2019; Cheng et al., 2016). MLP layers are replicated across multiple nodes (trainers) and run in parallel, while Emb layers are sharded across embedding parameter server nodes (Emb PS) due to their large memory capacity requirement (Zheng et al., 2020). As the size and the complexity of recommendation models grow (Zhao et al., 2020; Lui et al., 2021), the scale of MLP trainers and Emb PS nodes increases quickly, that leads to growing failure rates of training jobs. By analyzing
a large collection of industry-scale recommendation training jobs in production datacenters with failures, we find that the mean-time-between-failures (MTBF) is 14–30 hours on average. Similar statistics were reported for other production-scale systems (Schroeder & Gibson, 2009; Kondo et al., 2010; Garraghan et al., 2014; Sahoo et al., 2004).

A common approach to handle failures for distributed model training is with checkpointing. A checkpointing system periodically saves the system state (a checkpoint) to persistent storage. At a failure, all nodes participating in the training load the last checkpoint, setting the model state back to a consistent, earlier version of the model. We refer to this baseline as full recovery. We observed that the overheads coming from checkpoints are non-negligible. Checkpoint-related overheads in full recovery can consume an average of 12% of the total training time. And, for the worst 5% training jobs, training time slowdown can be up to 43%. This 12% overhead can add up to a significant computational cost at scale. By analyzing 17,000 training jobs from a 30-day window, we observed that 1.156 machine-year worth of computation was spent solely for failure handling.

In this work, we propose to leverage partial checkpoint recovery to improve the efficiency and reliability of recommendation model training. Unlike full recovery, partial recovery restores a checkpoint only for the failed node, allowing all other trainer and Emb PS nodes to proceed without reverting their progress. Prior work showed that the iterative-convergent nature of ML training can successfully train the model around the inconsistencies partial recovery introduces to a certain degree (Qiao et al., 2019). However, we demonstrate in this paper that a naive application of partial recovery can harm the final model accuracy.

We identify that varying the checkpoint saving interval trades off the final model accuracy as well as the training time overhead in partial recovery—a unique, unexplored trade-off space. To our knowledge, this is the first work that conducts a thorough characterization study to understand this trade-off in the context of production-scale recommendation model training. From the characterization study, we formulate a metric, portion of lost samples (PLS), to navigate the design space.

Using PLS, we introduce Checkpointing with Partial recovery for Recommendation systems (CPR), the first partial recovery system that is customized for a large-scale production recommendation system training. With the user-specified PLS mapping to a certain accuracy target, CPR assesses the benefit of using partial recovery and selects a checkpoint saving interval to meet the target PLS with minimal overheads. CPR implements two additional optimizations, Most-Frequently Used checkpointing (CPR-MFU) and Sub-Sampled Used checkpointing (CPR-SSU), to further improve the accuracy when using partial recovery. CPR-MFU and CPR-SSU leverage an important observation—frequently accessed rows in the embedding table experience larger updates that have heavier effects when lost. CPR-MFU and CPR-SSU save updates of frequently accessed rows with higher priority under a constrained system bandwidth, thereby minimizing the degree of the model inconsistency introduced by failures.

We design, implement, and evaluate CPR on (1) an emulation framework using the open-source MLPerf recommendation benchmark (MLPerf, 2020) and (2) a production-scale cluster, training production recommendation models. Our evaluation results show that CPR effectively reduces overheads while controlling the accuracy degradation using PLS. On a framework emulating the production-scale cluster, we show CPR can reduce the checkpoint-related overheads by 93.7% compared to full recovery, while only degrading accuracy by at most 0.017%. Also, our results on a real-world production cluster demonstrate promising overhead reduction from 12.5% to a marginal 1%, without any accuracy degradation. The main contributions are:

- We provide a systematic analysis on the impact of partial recovery for recommendation model training.
- We introduce PLS, a metric that can be used to predict the effect of partial recovery on model accuracy.
- We propose CPR, a practical partial recovery system for immediate adoption in real-world training systems.

\section*{2 BACKGROUND AND MOTIVATION}

\subsection*{2.1 Deep Learning Recommendation Systems}

Figure 1 depicts the generalized model architecture for deep learning recommendation systems. There are two important feature types that are modeled in a recommendation system: dense and sparse features. Dense features represent continuous inputs that are used directly as inputs to the bottom MLP layer whereas sparse features represent categorical inputs, such as movies or books a user has liked. The sparse features are often encoded as multi-hot vectors, with only the indices mapping to a certain category set. Because of the large feature domain, the multi-hot vectors are sparse—a few liked items among millions of items.
Before being used, the sparse feature representation must go through embedding tables and be translated to a dense vector representation. The embedding tables can be viewed as lookup tables, where each row holds a dense embedding vector. Embedding vectors encode the semantics of each feature, and the distance between embedding vectors represents semantic relatedness. The hot indices in the sparse feature representation are used as lookup indices to retrieve a set of embedding vectors, which are then combined in a feature pooling layer by operations such as summation or multiplication. The combined embedding vectors and the output of the bottom MLP layer are aggregated in the feature interaction layer, where their similarity is calculated, e.g., with dot-product. Finally, the result is fed into the top MLP layer, which predicts the likelihood of user engagement for the input user-item pair, i.e., click-through-rate or CTR.

MLP layers are compute-intensive and can be on the order of MBs in size. To exploit data-level parallelism, an MLP layer is often replicated across multiple trainers and trained in parallel with different sets of data samples. Trainers synchronize their replicated parameters periodically, either through an MLP parameter server (Li et al., 2014; Zhang et al., 2015) or using point-to-point communication (Assran et al., 2019; Seide et al., 2014).

Emb layers, on the other hand, are memory intensive. The embedding tables of production-scale recommendation models are often in the order of several hundreds of GBs to TB (Zhao et al., 2020; Lui et al., 2021) in size and do not fit in a single-node training system (Acun et al., 2021). Thus, embedding table training exploits model-parallelism, where tables are partitioned across multiple Emb PS nodes and are jointly trained with all training data samples.

2.2 Checkpointing for Distributed Model Training

A common practice to handle failures in a distributed system is to periodically save a checkpoint, i.e., store a snapshot of the system state to persistent storage. Checkpoints hold system states necessary to restore the progress. For ML training, checkpoints usually include the model parameters, iteration/epoch counts, and the state of the optimizer. When any of the nodes fails, loading checkpoints for all the nodes (i.e., full recovery) reverts the system to the same state as when the checkpoint was saved.

There are four major overheads when using full recovery: (1) checkpoint saving overhead ($O_{save}$), (2) checkpoint loading overhead ($O_{load}$), (3) lost computation ($O_{lost}$), and (4) rescheduling overhead ($O_{res}$). Checkpoint saving/loading overhead refers to the time spent on saving/loading the checkpoint. Lost computation is the amount of computation executed between the last checkpoint and a failure. Because the intermediate results were not saved, the same computation has to be re-executed. Rescheduling overhead is the time spent for the cluster scheduler to find alternative, available nodes to take over the role of the failed nodes (Basney & Livny, 2000; Yoo et al., 2003).

With an average node failure period $T_{fail}$ and the checkpoint saving interval $T_{save}$, a system’s total overhead $O_{total}$ can be represented roughly as the following formula:

$$O_{total} \approx O_{save} \frac{T_{total}}{T_{save}} + (O_{load} + \frac{T_{save}}{2} + O_{res}) \frac{T_{total}}{T_{fail}}$$

The first term ($O_{save} \frac{T_{total}}{T_{save}}$) represents the checkpoint saving overhead, calculated by multiplying the overhead of saving a checkpoint $O_{save}$ with the number of saving throughout training $\frac{T_{total}}{T_{save}}$. Similarly, the second, third, and fourth terms represent the overhead of checkpoint loading, lost computation, and rescheduling, multiplied by the number of failures ($\frac{T_{total}}{T_{fail}}$). Note that $O_{lost} = \frac{T_{total}}{T_{save}}$, assuming uniform failure probability. The formula assumes each overhead is small compared to $T_{total}$. Otherwise, e.g., the number of checkpoint saving would have to be calculated as $\frac{T_{total} + O_{total}}{T_{save}}$ instead of $\frac{T_{total}}{T_{save}}$. With knowing the system parameter $O_{save}, O_{load}, O_{res}$, and $T_{fail}$, the optimal checkpoint saving interval $T_{save}$ that minimizes $O_{total}$ can be calculated: $T_{save,full} = \sqrt{2O_{save}T_{fail}}$.

In a recommendation model training, Emb PS nodes account for most of the checkpoint-related overhead. Unlike MLP layers that are small and replicated across trainers, embedding tables are large and partitioned across multiple nodes. Thus, saving embedding tables is slow and requires coordination. Conventional checkpointing strategies (Koo & Toueg, 1987), therefore, are inefficient for handling Emb PS failures—the optimization focus of this work.

2.3 Partial Recovery

As an alternative to full recovery, the concept of partial recovery was proposed in recent work (Qiao et al., 2019). A distributed system with partial recovery only loads the checkpoint for the failed node, while keeping the progress of the remaining nodes. Unless the iteration/epoch count is lost, partial recovery does not revert the progress, eliminating the need to re-execute computations. The overhead of partial recovery does not contain the lost computation:

$$O_{total,paur} \approx O_{save} \frac{T_{total}}{T_{save}} + (O_{load} + O_{res}) \frac{T_{total}}{T_{fail}}$$

The performance benefit, however, comes at the expense of potential model quality degradation, because partial recovery introduces state inconsistencies across nodes. Prior work (Qiao et al., 2019) proposes to compensate for the accuracy loss with training for additional epochs, although there is no guarantee on eventual model quality recovery.

In fact, training for additional epochs does not always recover the model quality for recommendation model training,
because recommendation models are prone to model overfitting when trained with more than one epoch (Zhou et al., 2018). We show a motivational training scenario showing that partial recovery for recommendation model training can lead to irrecoverable accuracy degradation. In Figure 2, failures (red cross) during training were handled by partial recovery (orange, dashed). With partial recovery, the best accuracy is far lower than that without failures (blue, solid). Additional epochs do not close the accuracy gap, because recommendation models overfitted after the first epoch. The experimental setup for Figure 2 is discussed in Section 6.

**Unexplored design trade-off.** The accuracy degradation of partial recovery can be potentially mitigated by saving checkpoints more frequently. The relationship reveals a new trade-off space for partial recovery to explore—in partial recovery, changing the checkpoint saving interval trades off the training time overhead and model accuracy. The role of the checkpoint saving interval for partial recovery is very different from that of full recovery, where the optimal value is simply \( T_{\text{save, full}} = \sqrt{2} T_{\text{fail}} \). Understanding the trade-off space is essential for the practical adoption of partial recovery on real-world applications.

### 3 Understanding Failures for Production-Scale Training

Nodes in a large-scale training can fail for various reasons: hardware failures (Wang et al., 2017; Reagen et al., 2018; Birke et al., 2014; Narayanan et al., 2016; Dean & Barroso, 2013), system failures (e.g., out-of-memory), user errors (e.g., bug in the code), and maintenance failures (e.g., kernel update) (Chen et al., 2014). While the failure probability of each node may be low, as the number of participating nodes increases, the likelihood of failure becomes higher.

#### 3.1 Distributed Recommendation Training Failures

Failures are common in distributed systems. Prior studies show that the mean-time-between-failures (MTBF) for distributed systems are usually within the order of several hours: 2–27 hours for the high-performance computing cluster of the Los Alamos National Lab (Schroeder & Gibson, 2009); 0.06–58.72 hours from the Google cloud trace log (Garraghan et al., 2014); 0.08–16 hours for a large-scale heterogeneous server cluster in IBM Research Center (Sahoo et al., 2004); 3–23 hours for a multi-tenant GPU clusters for DNN training at Microsoft (Jeon et al., 2019).

We observed a similar trend in the MTBF with a large collection of recommendation training workflows from the production-scale clusters. From the logs of 20,000 training jobs running on fleets of Intel 20-core 2GHz processors connected with 25Gbit Ethernet, similar to Zheng et al. (2020) and Kalamkar et al. (2020), we collected the time-to-failure data. We excluded training runs without failures in the statistics. Figure 3a plots the survival probability of a training job over time. We overlaid a fitted gamma distribution on top of the observed survival probability data and extrapolated the failure probability, shown in Figure 3b.

The median-time-between-failure (corresponding to \( y = 0.5 \) in Figure 3a) was 8–17 hours and the MTBF was 14–30 hours, similar to statistics from prior work (Schroeder & Gibson, 2009; Kondo et al., 2010; Garraghan et al., 2014; Sahoo et al., 2004). Jobs with more nodes failed more quickly, with the MTBF decreasing linearly with the increasing number of nodes. Similar to prior work on modeling hardware failures (Wang et al., 2017), the observed training failures followed a gamma distribution closely, with an RMSE of 4.4%. The gamma distribution fits the best compared to other commonly-used heavy-tailed distributions, e.g., Weibull (Runne, 2008), exponential (Balakrishnan, 2018), and log-normal (Weisstein, 2010). The derived failure probability was close to uniform, except near the beginning of training (Figure 3b). The much higher failure probability near the beginning is likely related to user errors, e.g., erroneous configuration leading to instant failure.

#### 3.2 Checkpoint Overhead Analysis for Reliable Recommendation Training

We quantified the impact of the four checkpoint-related overheads from Section 2.2 in a production-scale training.
cluster. Similar to the failure analysis, we inspected 17,000 training jobs that ran for more than 10 hours over a 30-day period. Figure 4 shows the checkpoint-related overhead breakdown. The four overhead categories added up to an average of 12% of the total training time. We estimated the wasted machine-time due to training failures by multiplying the time wasted with the number of nodes. Even though the average overhead of 12% may seem small, the implication is dire: the total overhead of the 17,000 training jobs summed up to 1,156 machine-year worth of computation.

Figure 4 shows that the overhead is not dominated by a single source. The major source of overhead for training jobs experiencing fewer failures comes from checkpoint saving (8.8% for p75), while training jobs with more frequent failures suffered from lost computation (13.2% for p90) and rescheduling (23.3% for p95). High rescheduling overhead near the tail happens when the cluster is heavily utilized with additional queuing delay. The diverse sources of overhead pose a dilemma to full recovery. To optimize for checkpoint saving overheads, a full recovery system must save checkpoints less frequently. However, to optimize for lost computation, the system must save checkpoints more frequently. Motivated by the dilemma, we explore an alternative solution, partial recovery. Next section describes the proposed system, CPR, that applies partial recovery to recommendation model training.

4 THE CPR SYSTEM

CPR is a lightweight design to improve the efficiency and reliability of distributed recommendation model training with partial recovery. In order to understand the performance–accuracy trade-off space of partial recovery, we define a new metric—portion of lost samples (PLS)—which is a function of checkpoint saving interval, the failure rate of the system, and the number of Emb PS nodes. Our empirical analysis shows that PLS strongly correlates with the final model accuracy. Based on this observation, a user selects a target PLS corresponding to the degree of accuracy degradation that is tolerable. Then CPR selects a checkpoint saving interval to achieve the target PLS. When the selected interval brings too much overhead, CPR simply falls back to full recovery. To improve the accuracy for CPR further, we introduce two optimizations, CPR-MFU and CPR-SSU, that prioritize saving embedding vectors with large changes. Figure 5 provides the design overview for CPR.

4.1 Portion of Lost Samples (PLS)

PLS represents the portion of the training data samples whose effect on the model was lost due to a failure. We empirically show that PLS has a high correlation to final model accuracy and can be used to trade-off performance and accuracy. Let $S_{total}$ denote the number of total samples, $S_i$ the number of samples processed up to $i$-th iteration, and $N_{emb}$ the number of Emb PS. The PLS at iteration $i$ is:

$$PLS_i = \begin{cases} 
0, & \text{if } i = 0 \\
PS_{i-1} + \frac{S_i - S_{last,chkpt}}{S_{total} N_{emb}}, & \text{if failure at } i \\
PS_{i-1}, & \text{otherwise}.
\end{cases}$$

(3)

$s_{i} - s_{last,chkpt}$ represents the portion of the lost samples among total samples, $N_{emb}$ in the denominator accounts for the fact that the lost information from a node failure is roughly $1/N_{emb}$ with $N_{emb}$ nodes. As verified below in Section 6.5 with measurement data and analysis, the final model quality is linearly correlated with the final PLS value of the recommendation training system. Using this relationship, a CPR user can provide a target PLS corresponding to the accuracy degradation they are willing to tolerate. CPR selects the checkpoint saving interval so that the expected PLS of the system meets the target PLS. The expected PLS can be calculated from the checkpoint saving interval and the failure frequency:

$$E[PLS] = \frac{0.5 T_{save}}{T_{fail} N_{emb}}$$

(4)

We briefly describe the derivation. Expected PLS is the number of expected node failures times the expected PLS increase on each node failures. With the time at $i$-th iteration of $t_i$, the total training time of $T_{total}$, and the expected value for all $i$, $E_i$, the expected PLS increases on node failures is:

$$E[\Delta PLS] = E_i [\frac{S_i - S_{last,chkpt}}{S_{total} N_{emb}}] = E_i [\frac{S_{last,chkpt}}{S_{total} N_{emb}}] =$$
When the checkpoint saving interval is too small to reap any benefit, CPR can potentially benefit from adopting SCAR. However, SCAR is impractical to implement in industry-scale recommendation training systems. Tracking updates to the embedding tables of several TBs in size requires the same order-of-magnitude memory capacity, at most requiring as much memory as the model itself. Furthermore, selecting the top \( rN \) most changed vectors has a time complexity of \( O(N \log(N)) \), scaling poorly with increasing \( N \).

Instead of tracking the updates directly, we propose to only track the access frequency. Figure 6 shows the strong correlation between the access frequency and the size of the update to embedding vectors, measured after 4096 iterations for the Kaggle dataset (Criteo Labs, 2014) (evaluation details in Section 6). The correlation coefficient is high, at 0.983, meaning that the access frequency is an excellent proxy to the magnitude of the embedding vector update. Based on this observation, we propose time- and memory-efficient alternatives over SCAR: CPR-MFU and CPR-SSU.

### CPR-MFU
CPR-MFU saves the Most-Frequently-Used (MFU) \( rN \) out of \( N \) parameters on every \( rT_{\text{save}} \), with \( r < 1 \). A 4-byte counter is allocated for each vector in the embedding table to track the access frequency. The typical size of an embedding vector ranges from 64–512 bytes (Naumov et al., 2019), making the memory overhead of the counter 0.78–6.25% of the size of the embedding tables. This is much smaller compared to the 100% memory overhead of SCAR. When an embedding vector is saved, its counter is cleared. The time complexity, however, is the same with SCAR, being in the order of \( O(N \log(N)) \).

### CPR-SSU
CPR-SSU further improves the time and memory overhead of CPR-MFU. CPR-SSU Sub-Samples Used (SSU) embedding vectors and keeps a list of vectors that were ever accessed from the subsampled data points, of size \( rN \). If the list overflows, CPR-SSU randomly discards the overflowing entries. The idea of CPR-SSU is that the subsampling will act as a high-pass filter, giving vectors with more frequent accesses a higher likelihood of staying in the list. Because CPR-SSU only requires a list of size \( rN \), the memory overhead is \( r < 1 \) times that of CPR-MFU. With \( r = 0.125 \), the memory overhead becomes 0.097–0.78%.

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Mem (rel. to emb tbl)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAR</td>
<td>( \approx O(N \log(N)) )</td>
<td>100%</td>
</tr>
<tr>
<td>MFU</td>
<td>( \approx O(N \log(N)) )</td>
<td>0.78 – 6.25%</td>
</tr>
<tr>
<td>SSU</td>
<td>( \approx O(N) )</td>
<td>0.097 – 0.78%</td>
</tr>
</tbody>
</table>

Table 1. Time and memory overhead of SCAR (Qiao et al., 2019), CPR-MFU, and CPR-SSU. Memory overhead is shown for embedding vectors of size 64–512 bytes, with \( r = 0.125 \).
of the embedding tables. CPR-SSU only needs to keep a non-duplicate list of size \( rN \), which has a time complexity in the order of \( O(N) \). Table 1 summarizes the overhead of SCAR, CPR-MFU, and CPR-SSU.

5 Experimental Methodology

We evaluated CPR in two different settings: (1) a framework that emulates the characteristics of the production-scale cluster, and (2) a real production-scale cluster.

5.1 Emulation Framework

The emulation framework allows a fast evaluation of CPR using a small model and a small dataset, while emulating the failure/overhead characteristics from the production cluster. For emulation, we implemented and trained CPR on top of the DLRM recommendation architecture (Naumov et al., 2019), a standard reference provided by MLPerf (Wu et al., 2020; MLPerf, 2020). We trained DLRM using two datasets of different sizes, the Criteo Kaggle (Criteo Labs, 2014) and Terabyte datasets (Criteo Labs, 2013). The hyperparameters of DLRM differed depending on the dataset. For Kaggle, we use 64-byte embedding vectors, a 4-layer Bottom MLP of \((13 \times 512, 512 \times 256, 256 \times 64, 64 \times 16)\), and a 3-layer Top MLP of \((432 \times 512, 512 \times 256, 256 \times 1)\). For Terabyte, we use 256-byte embedding vectors, a 3-layer Bottom MLP of \((13 \times 512, 512 \times 256, 256 \times 64)\), and a 4-layer Top MLP of \((1728 \times 512, 512 \times 512, 512 \times 256, 256 \times 1)\). DLRM was trained on a single machine with two NVIDIA V100 GPUs attached to a server with 20 CPUs and 64GB memory. Using a single node does not affect the accuracy of DLRM because the implementation is fully synchronous.

We ran training for a single epoch using all the data samples and reported the final test receiver operating characteristic area under curve (AUC). AUC is less susceptible to unbalanced datasets and is a standard metric for evaluating DLRM (MLPerf, 2020). Training for a single epoch is common for DLRM (Naumov et al., 2019; Mattson et al., 2020a;b), because DLRM suffers from overfitting if the same training data is revisited.

Failure and overhead emulation. Because the emulation runs much faster than production-scale training, we project the failure/overhead characteristics from Section 3 down to account for the training time difference. We emulate a 56-hour training job for simplicity: the average number of failures for a 56-hour training was exactly 2. We inject 2 failures randomly, as the failure probability is nearly uniform for the real-world cluster (Section 3.1). A failure clears 50%, 25%, or 12.5% of the embedding tables and triggers partial recovery, emulating 50%, 25%, or 12.5% of the total Emb PS failures. We linearly scale down the checkpoint-related overheads and the checkpoint saving interval.

Strategies. We implemented and compared full recovery, baseline partial recovery, CPR-vanilla, CPR-SCAR, CPR-MFU, and CPR-SSU. Full recovery uses the optimal checkpoint saving interval \( T_{\text{save}} = \sqrt{2O_{\text{save}}T_{\text{fail}}} \). The baseline partial recovery naively uses the same interval. CPR calculates the checkpoint saving interval from the target PLS. We compare four different variants of CPR: CPR-vanilla calculates the checkpoint saving interval from the target PLS without additional optimizations. CPR-SCAR implements the SCAR optimization from prior work (Qiao et al., 2019), which imposes significant memory overhead. CPR-MFU/SSU applies our memory-efficient MFU/SSU optimizations. For CPR-SSU, we use a sampling period of 2. We only apply SCAR/MFU/SSU optimizations to the 7 largest tables among 26 tables, which take up 99.6% (Kaggle) and 99.1% (Terabyte) of the entire table size, respectively. For the 7 tables, we save checkpoints 8 times more frequently but, at most, only 1/8 of the parameters compared to full recovery (i.e., \( r = 0.125 \)). Other tables are always fully saved.

5.2 Production-scale Cluster

The evaluation on the production-scale cluster used 20 MLP trainers and 18 Emb PS nodes. Each node consists of Intel 20-core, 2GHz processors connected with 25Gbit Ethernet, similar to (Zheng et al., 2020). We trained the model for a total of 50 hours, during which 5 failures were injected, with each failure on any four randomly selected Emb PS nodes. To simply test the effect of partial recovery, we mimicked the behavior of partial recovery by switching part of the checkpoints to an older version and triggering full recovery right after saving a checkpoint. Except for the nodes whose checkpoints were switched, loading checkpoints would not revert the model, having the same effect as partial recovery.

6 Evaluation Results

6.1 Emulation Results: Training Time and Accuracy

We ran full recovery (Full.), baseline partial recovery (Part.), and different variants of CPR on our failure emulation framework, which closely emulates our production-scale cluster’s failure rate and the checkpoint saving overhead. For the variants of CPR, we used target PLS = 0.1. Figure 7 summarizes the result for both Kaggle and Terabyte datasets.

CPR reduces the training time. Compared to full recovery, CPR reduces the checkpoint-related overhead by 93.7% and 91.7% for Kaggle and Terabyte, respectively. The speedup can be broken down into two factors. Elimination of the lost computation reduces the overhead for Kaggle from 8.5% to 4.4% and for Terabyte from 8.2% to 4.4% (Figure 7, Full. vs. Part.). PLS-based checkpoint saving interval selection additionally brings down the 4.4%
CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery

We varied the target PLS for CPR-vanilla used a target PLS from 0.025 to 0.1, which resulted in using checkpoint saving interval of 2–8 hours. Full recovery saved checkpoints every 2 hours. Because the production-scale training did not report AUC, we report the training loss instead. Note that unlike AUC, the training loss is lower the better.

Figure 8 summarizes the result. The training loss only increased by 0.0007 from full recovery to CPR-vanilla with PLS=0.1. Meanwhile, the overhead decreased significantly with CPR-vanilla with PLS=0.1, from 7.83% to 0.708%. Most of the overhead reduction (5%) came from the elimination of lost computation. 2.12% reduction came from saving checkpoints less frequently. The limited number of data points suggests a possible benefit of using CPR in a production environment. We did not evaluate CPR-MFU/SSU, because the accuracy was already good.

In addition, we studied how much hot embedding vectors change throughout the training process. We observed that most embedding tables converged quickly with marginal changes in the cosine distance when compared to full recovery. This can potentially explain why CPR achieves a similar model accuracy. However, a few embedding tables changed more drastically. The distinct convergence behavior of the individual embedding tables indicates further optimization opportunities in the context of CPR with, for example, hybrid failure handling strategies.

6.3 Sensitivity Study: PLS

To evaluate the effect of different target PLS, we varied the target PLS between 0.02, 0.1, and 0.2 and present the resulting accuracy and overhead. We only show the result of CPR-vanilla and CPR-SSU from the Kaggle dataset for brevity; other configurations showed a similar trend. Figure 9 summarizes the result. For both CPR-vanilla and CPR-SSU, varying PLS effectively traded off accuracy and performance. For CPR-vanilla, increasing the target PLS from 0.02 to 0.2 decreased the overhead from 2.9% to 0.3%, while degrading accuracy from AUC=0.8028 to AUC=0.8021. For CPR-SSU, the degradation was much lower. CPR-SSU experiences a marginal AUC decrease from AUC=0.8028 to AUC=0.8027 for the same speedup.

**CPR maintains reasonable accuracy.** With optimizations, CPR was able to mostly achieve accuracy on par with full recovery, losing at most only 0.0002 test AUC with optimizations. For both full recovery and baseline partial recovery, the test AUC was 0.8028/0.7977 for Kaggle/Terabyte dataset. CPR-vanilla trades off accuracy with performance. While reducing the overhead to a marginal 0.53–0.68%, the AUC for CPR-vanilla decreased to 0.8025/0.7974, for Kaggle/Terabyte dataset (0.04% degradation). CPR-SCAR/MFU/SSU improves accuracy, making CPR much more practical. For Kaggle, they all reached a test AUC on par with that of full recovery. For Terabyte, CPR-SCAR/MFU/SSU achieved AUC=0.7976/0.7976/0.7975 (0.011/0.012/0.017% degradation), respectively. While using less memory (Table 1) CPR-MFU/SSU achieved accuracy similar to that of CPR-SCAR.

6.2 Production-scale cluster Results: Training Time and Accuracy

We evaluated full recovery and CPR-vanilla on a production-scale cluster. We injected 5 failures uniformly throughout training that failed 4 of the 18 Emb PS nodes randomly. (4.4%) overhead to 0.53% (0.68%) for Kaggle (Terabyte), respectively (Figure 7, Part. vs. CPR).

![Figure 7](image7.png)

**Figure 7.** CPR reduces the checkpoint-related overhead over full recovery by 93.7% (Kaggle) and 91.7% (Terabyte) on a setup emulating the production cluster, while achieving similar accuracy.

![Figure 8](image8.png)

**Figure 8.** CPR’s training loss increased only by 0.0007 from full recovery to CPR-vanilla with PLS=0.1, while the overhead was reduced from 7.8% to 0.71% on a production-scale setup.

![Figure 9](image9.png)

**Figure 9.** CPR trades off performance and accuracy.

We varied the target PLS for CPR-vanilla used a target PLS from 0.025 to 0.1, which resulted in using checkpoint saving interval of 2–8 hours. Full recovery saved checkpoints every 2 hours. Because the production-scale training did not report AUC, we report the training loss instead. Note that unlike AUC, the training loss is lower the better.

Figure 8 summarizes the result. The training loss only increased by 0.0007 from full recovery to CPR-vanilla with PLS=0.1. Meanwhile, the overhead decreased significantly with CPR-vanilla with PLS=0.1, from 7.83% to 0.708%. Most of the overhead reduction (5%) came from the elimination of lost computation. 2.12% reduction came from saving checkpoints less frequently. The limited number of data points suggests a possible benefit of using CPR in a production environment. We did not evaluate CPR-MFU/SSU, because the accuracy was already good.

In addition, we studied how much hot embedding vectors change throughout the training process. We observed that most embedding tables converged quickly with marginal changes in the cosine distance when compared to full recovery. This can potentially explain why CPR achieves a similar model accuracy. However, a few embedding tables changed more drastically. The distinct convergence behavior of the individual embedding tables indicates further optimization opportunities in the context of CPR with, for example, hybrid failure handling strategies.
6.4 Sensitivity Study: Failures

We also varied the number of failures and the portion of lost nodes on each failure. We fixed the target PLS to 0.02. We varied the number of failures between 2, 20, and 40. 20 and 40 failures represent a hypothetical case where the system experiences 10–20× more failures. Such a setup can represent a scenario of off-peak training, a training that only uses idle resources and gets suspended whenever a higher priority job arrives (e.g., Amazon Spot (Amazon, 2020)). On each failure, we varied the portion of the Emb PS nodes failed between 12.5–50%. We only plot the overhead; the accuracy was similar across all experiments. The overhead is normalized to the overhead of full recovery for simple comparison. Again, we only selectively show full recovery and CPR-SSU, trained with Kaggle dataset. Omitted data points showed a similar trend. The configurations CPR found as not beneficial to run a partial recovery are marked in a red hatch. We still plot what the overhead would have been like had CPR run partial recovery in such setups.

Figure 10 shows that CPR correctly estimates the benefit of using partial recovery. The overhead of the setup CPR predicted as not beneficial to run partial recovery (red hatch) was all higher than that of full recovery. Figure 10 also shows that CPR’s speedup becomes smaller when failures occur more frequently or when more nodes fail at once. CPR is less effective with more frequent failures because the checkpoint saving interval of partial recovery \((2(\text{PLS})N_{\text{emb}}T_{\text{fail}})\) decreases faster with decreasing mean-time-to-failure, compared to full recovery \((\sqrt{2O_{\text{save}}T_{\text{fail}}})\).

6.5 PLS and Accuracy

CPR relies on the linear relationship between the PLS and the final model accuracy. To evaluate the relationship, we generated runs with 1–32 random failures, each clearing 6.25–50% of the embedding vectors. We also randomly selected a checkpoint saving interval so that the expected PLS falls near 0–1. We applied partial recovery without any optimization and plotted the final accuracy degradation compared to the non-failing case.

Figure 11 shows the strong linear relationship between PLS and Accuracy.

6.6 Partial Recovery Scalability Analysis

To study the scalability of CPR, we analytically estimated the overhead of full recovery and CPR using Equation 1 and Equation 2. To conjecture how the rate of node failures would increase, we assumed two different models: (1) linearly decreasing mean-time-between-failure (MTBF) with
an increasing number of nodes, which was the behavior observed from Section 3.1, and (2) assuming that each node has an independent failure probability $p$. The second model leads to an MTBF equation in the form of $\frac{1}{1-(1-p)^n}$, which deviates from the linear behavior seen from the production cluster. Still, we consider this model due to its simplicity.

Figure 13 plots the result. For both of the failure models, CPR showed better scalability than full recovery, where the overhead actually decreased with an increasing number of nodes. For both cases, full recovery saw an increasing overhead with the increasing number of nodes. CPR scales better with an increasing number of nodes because, although the probability of observing a failure increases, the portion of the updates lost decreases with the number of nodes. Full recovery loads all the checkpoints even if only a small fraction of the model is lost, resulting in worse scalability.

7 **ADDITIONAL RELATED WORK**

7.1 **Prior Work on Checkpointing**

**Checkpointing for non-ML applications.** Checkpointing is a common technique used in data centers to handle failures (Chandy & Lamport, 1985; Koo & Toueg, 1987). Traditional checkpointing saves a globally consistent state across all the participating nodes and use full recovery to ensure correct behavior (Chandy & Lamport, 1985), which is often expensive. Many optimizations orthogonal to CPR have been proposed to speed up checkpointing, including using multi-level hierarchy (Moody et al., 2010; Bautista-Gomez et al., 2011), adding asynchronous operations (Nicolaie et al., 2019), leveraging memory access patterns (Nicolaie & Cappello, 2013; Carbone et al., 2015), or simultaneously using logging (Wang et al., 2019). These works aim to support arbitrary workloads and are complementary.

Intermittent computing (Ransford et al., 2011; Jayakumar et al., 2014; Maeng & Lucia, 2018; Maeng et al., 2019; Maeng & Lucia, 2019; 2020; Hester et al., 2015; Hester & Sorber, 2017; Lucia & Ransford, 2015; Van Der Woude & Hicks, 2016; Hicks, 2017; Ma et al., 2015; Choi et al., 2019), a field enabling compute on an energy-harvesting device with frequent failures, has also widely adopted checkpointing. However, these works focus on a single-node system.

**Checkpointing for distributed ML training.** Several distributed training systems implement checkpointing (Chilimbi et al., 2014; Narayanan et al., 2019; Cipar et al., 2013). Orpheus (Xie et al., 2018a) incrementally saves a checkpoint by breaking the model into a running sum of decomposed vectors, from which the original model can be recalculated. DeepFreeze (Nicolaie et al., 2020) improves the checkpointing efficiency by introducing multi-level storage, sharding the work across nodes, and overlapping compute with checkpoint saving. While some of the prior works reduce checkpoint-related overhead by leveraging ML-specific characteristics, they do not use partial recovery like CPR. SCAR (Qiao et al., 2019) is the first system that explores the benefit of partial recovery. CPR additionally studies the trade-off of partial recovery that SCAR neglected and proposes memory-efficient optimizations.

8 **CONCLUSION AND FUTURE WORK**

Training a recommendation system requires a fleet of machines due to its high compute and memory demand. With the ever-increasing number of participating nodes, the training process experiences more and more frequent failures. We studied the failure characteristics and the resulting overheads and observed that traditional full recovery adds unnecessary checkpoint saving overhead and lost computation.

We propose CPR, a system leveraging partial recovery to reduce the checkpoint-related overheads for recommendation system training. CPR selects checkpoint saving interval based on the user-specified target PLS, maximizing performance while maintaining reasonable accuracy. CPR also implements low-overhead optimizations that further reduce the accuracy degradation. We show that CPR can effectively eliminate checkpoint-related overhead with partial recovery while suppressing significant accuracy degradation.

Partial checkpoint recovery after a failure perturbs the training process. Consequently, when training with CPR it may be beneficial to use more robust distributed training methods, such as those designed to handle more adversarial Byzantine failures (Yin et al., 2018; Chen et al., 2018). We leave this line of investigation to future work.

**ACKNOWLEDGEMENTS**

The authors would like to thank Hsien-Hsin Sean Lee, Kim Hazelwood, Udit Gupta, David Brooks, Maxim Naumov, Dheevatsa Mudigere, Assaf Eisenman, Kiran Kumar Matam, and Abhishek Dhanotia for the discussion and feedback.
REFERENCES


CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery


CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery

on Architectural support for programming languages and operating systems, pp. 159–170, 2011.


Van Der Woude, J. and Hicks, M. Intermittent computation without hardware support or programmer intervention. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 17–32, 2016.


CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery


