On Efficient Constructions of Checkpoints

Yu Chen [1] **Zhenming Liu** [1] **Bin Ren** [1] **Xin Jin** [2]

Abstract

Efficient construction of checkpoints/snapshots is a critical tool for training and diagnosing deep learning models. In this paper, we propose a lossy compression scheme for checkpoint constructions (called LC-Checkpoint). LC-Checkpoint simultaneously maximizes the compression rate and optimizes the recovery speed, under the assumption that SGD is used to train the model. LC-Checkpoint uses quantization and priority promotion to store the most crucial information for SGD to recover, and then uses a Huffman coding to leverage the non-uniform distribution of the gradient scales. Our extensive experiments show that LC-Checkpoint achieves a compression rate up to $28\times$ and recovery speedup up to $5.77\times$ over a state-of-the-art algorithm (SCAR).

1. Introduction

Efficient construction of checkpoints (snapshots) has been increasingly important to deep learning research. In the arms race of developing more accurate models, researchers utilize heavier computing infrastructure and develop deeper and larger models. Without proper infrastructure support, the research process inevitably becomes fragile. For example, distributed computation fails from time to time, leading to the excessive need to re-train models (Qiao et al., 2018b). Diagnosing deep learning models also evolves to a complex procedure partly because that the community has a better understanding of deep learning models and produces more rules for "debugging" them. Some common errors include gradient explosion (Goodfellow et al., 2016), "divide by zero" (Ioffe & Szegedy, 2015), and dead activation. This calls for the need to construct "breakpoints," resembling those used in debugging computer programs, so that researchers can conveniently jump to the state right before the model "crashes" in the training.

[1]William & Mary, Williamsburg, Virginia, USA [2]Johns Hopkins University, Baltimore, Maryland, USA. Correspondence to: Yu Chen <ychen39@email.wm.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

Producing checkpoints frequently enables failed training process to restart with minimum wasted time, and serves as breakpoints for debugging models. So far the standard practice of constructing checkpoints is primitive. The most common practice is to save the model state directly, counting on that the backend system is sufficiently robust so that this operation does not become a bottleneck (Baylor et al., 2017). Attempts of partially storing model states are also examined (Qiao et al., 2018b) but these works usually focus on recovery speed, instead of directly tackling system issues.

The most pronounced technical challenge here is that deep models are usually large, so producing frequent checkpoints creates unmanageable burdens to both I/O and storage, even under modern distributed platforms (Abadi et al., 2016; Li et al., 2014; Low et al., 2012). Therefore, this leads to our question:

Research Q: *How can we compress model checkpoints?*

We specifically aim to design a *lossy* compressing scheme, addressing two criteria simultaneously. First, like standard compression problems, we need to maximize the compression rate. Second, the scheme needs to be optimized for the downstream application of *training*. When a model restarts from our lossy checkpoints, it needs to efficiently resume to the most recent state (e.g., restart from a failed process or reach the state preceding the crash).

Compression of model states is a new technical problem that requires addressing cross-cutting constraints from information theory, learning algorithm, and system design. We need to leverage statistical patterns encoded in the model state and factor in how the model states interact with a learning algorithm (more specifically, stochastic gradient type algorithms in the deep learning setting). This means neither standard lossy compression algorithms nor recently developed model compression algorithms (Han et al., 2015a; Courbariaux et al., 2015; Hong et al., 2016; Leng et al., 2018; Lin et al., 2016) directly work in our setting. Standard lossy compression algorithms aim to minimize reconstruction error but our end goal is to enable a learning algorithm to "quickly recover." Model compression techniques aim to transform a (static) model into a simpler one while ensuring the forecasts are not perturbed much whereas in our setting we need a reliable coding scheme that functions well throughout the

entire dynamic process of learning, which is an orthogonal and perhaps more challenging goal. In addition, our algorithm must be efficient and scalable so that it can be executed frequently.

Our solution. To achieve our aims, we focus on a delta-encoding scheme (Mogul et al., 1997), tracking only the information on the difference between two checkpoints. Under this scheme, we examine whether we can cut the least useful information (with respect to training) from the model state, and ensure that the remaining information is amenable for compression. A perhaps surprising message here is that ℓ_2-norm reconstruction error for the "delta" appears to be an ineffective metric for minimizing the recovery time. Instead, our algorithm first removes all the parameters with inconsequential updates, and then quantizes the remainder information. These strategies resemble those used in distributed training with the goal of minimizing communication cost (Alistarh et al., 2017). After we obtain the most significant information for portion of parameter updates, we represent them in suitable format and apply a Huffman coding to further compress these bits, so that the compression rate can be at the information theoretic limit. This strategy resembles recent techniques for model compression (Han et al., 2015a; Wu et al., 2016; Park et al., 2017; Zhou et al., 2017; Rastegari et al., 2016).

The contribution of this paper includes:

- Proposal of a fundamental research question on compressing model states for training recovery.
- Characterization of a family of compression schemes that can efficiently track the learning process, based on a stylized model we develop.
- Design of a lossy coding scheme with high-compression rate that integrates both classical compression techniques and recent ones developed for distributed learning and model compression.
- Optimization of training systems that minimizes the overhead of producing checkpoints on the fly.

Our extensive evaluation demonstrates that by simultaneously leveraging techniques from distributed training and model compression, our algorithm delivers a solution (called LC-Checkpoint, LC refers to Lossy Compression) with a compression rate of **up to 28x** and superior recovering time—achieving up to $5.77\times$ recovery speedup over a state-of-the-art algorithm (SCAR).

2. Our approach

We now describe our compression framework. We introduce a stylized model for the learning process to facilitate the analysis of the system design trade-off. Then we explain our design principles, determined by both the stylized model and our extensive experiments.

Our model. A "high-dimensional" vector $\mathbf{u} \in \mathbf{R}^n$ represents the model state. An iterative algorithm (e.g., stochastic gradient descent) is used to gradually move the model state vector $\mathbf{u}$ toward a local optimal point $\mathbf{u}^*$. Let $\mathbf{u}_t$ be the model state at the t-th round. In our stylized model, we assume $\mathbf{u}_t$ performs a (drifted) random walk that converges to $\mathbf{u}^*$. Specifically, we use the following process to model $\mathbf{u}_i$'s trajectory. Let $L = \|\mathbf{u}_0 - \mathbf{u}^*\|$.

$$\mathbf{u}_{t+1} = \mathbf{u}^* + \eta(\mathbf{u}_t - \mathbf{u}^*) + \epsilon_t, \tag{1}$$

where η and L jointly model the convergence rate of the algorithm, and ϵ_t is a random noise component to reflect the stochastic nature of SGD. When η is set to be a small constant, the model characterizes those algorithms that have linear convergence rate. When $\eta = (1 - 1/L)$, this model characterizes those algorithms whose convergence rates are $1 - 1/t$ (Boyd & Vandenberghe, 2004). While our model does not captures the detail of many SGD algorithms, because different SGD algorithms have different convergence rate, designing a unifying model that highlights design trade-offs requires us to make simplifying assumptions.

Our design principles. We next describe our design principles.

P1. Minimize irritation to SGD. When we design lossy compression scheme, a portion of information is inevitably lost, causing performance degradation to a learning algorithm. We find that we should not simply use ℓ_2 reconstruction error to measure degradation of SGD. This can be best illustrated by the stylized model. For simplicity, let $\mathbf{u}^* = 0$, so $\mathbf{u}_{t+1} = \mathbf{u}_t - ((1-\eta)\mathbf{u}_t + \epsilon_t)$. The delta term we want to compress is $((1-\eta)\mathbf{u}_t + \epsilon_t)$. When we use a lossy compression, it corresponds to adding an additional noise term that is a function of $\mathbf{u}_t$ and ϵ_t. So with the compression scheme, the new learning process becomes $\mathbf{u}_{t+1} = \mathbf{u}_t - ((1-\eta)\mathbf{u}_t + \epsilon_t + f(\mathbf{u}_t, \epsilon_t))$. Observing that as long as $\mathbb{E}[f(\mathbf{u}_t, \epsilon_t) \mid \mathbf{u}_t, \epsilon_t] = 0$, and $\mathrm{Var}(f(\mathbf{u}_t, \epsilon_t) \mid \mathbf{u}_t, \epsilon_t)$ is dominated (smaller than) by $\mathrm{Var}(\epsilon_t)$, then the convergence quality remains unchanged, by standard results from stochastic approximation (Lai, 2009; Kushner & Yin, 2003).

There are many constructs that satisfy the expectation and variance constraints. Let us consider an example of keeping the most significant bit of $((1-\eta)\mathbf{u}_t + \epsilon_t)$ by using standard randomized rounding (Alistarh et al., 2017). Because of the nature of the rounding algorithm, the expectation is 0. In addition, because the most significant bit is kept, the information loss in rounding will not be greater than $\| ((1-\eta)\mathbf{u}_t + \epsilon_t) \|_2 = O(\mathrm{std}(\epsilon_t))$ under a mild assumption that ϵ_t's standard deviation also scales proportionally to $\|\mathbf{u}_t\|$ over time. Therefore, this rounding scheme does not

affect the performance of the training algorithm. In general, the 1-bit encoding is a special case of quantization. A wide family of quantization schemes will satisfy the expectation and variance constraint. Our algorithm will explore this trade-off.

Note also when we minimize ℓ_2 reconstruction error, this corresponds to keeping top-k heaviest entries in $\mathbf{u}_{t+1} - \mathbf{u}_t$.

P2. Maximize redundancies in residual information. Our compression scheme also needs to ensure the information we keep exhibits large redundancy, as measured by entropy. This will enable us to use traditional coding schemes such as Huffman code to compress the data at the information theoretic limit.

The interplay between P1 and P2 highlights the unique structure of our compression problem. This can be best illustrated by a compression scheme called TOPN. This compression scheme keeps the largest elements in δ_t. We observe *(i)* while this scheme minimizes ℓ_2 reconstruction error, it *does not* have superior recovery time. Many other compression schemes that possess the aforementioned properties recover equally fast, as suggested by our stylized model. *(ii)* It is difficult to perform compression for the TOPN scheme. TOPN scheme usually needs to track 10% of all the entries in δ_t to be effective. The overhead of tracking the *locations* of these elements is surprisingly high. This is because in part that the vector is not sufficiently sparse so sparse matrix representation does not help.

Our solution, on the other hand, carefully complies P1 and circumvents the need to track the locations of the entries we keep and thus achieves significantly higher compression rate.

P3. Do not use random projections and/or sketches. Notably, we discover that sketch-based randomized projection techniques (e.g., Woodruff et al. (2014)) *harm* the compression. Roughly speaking, sketches compress information by projecting multiple numbers into one cell. While this could speed up query time, it only irritates the gradient descent algorithm in our setting. Consider a toy example in which $\mathbf{u}_t \in \mathbf{R}^2$ and the optimal point $\mathbf{u}^* = (0, 10)$. Let $\mathbf{u}_t = (5, 5)$ be the current state so the gradient is along the direction $(-1, 1)$. When we apply sketches (say CountMin sketches), it collapses the direction $(-1, 1)$ into a single point 0. When we make a query, the gradients for both coordinates are incorrect. Sketches are more useful when the entries in the gradient vector are heterogeneous and queries need to be answered at "line rate" (e.g., do not slow down the training Ivkin et al. (2019)). Here, when a model needs to be recovered from a checkpoint, the job is less time-sensitive. Therefore, even we face heterogeneous parameters, it is more effective to carefully disentangle crucial information from inconsequential ones than using arbitrary random projections.

Algorithm 1 LC-CHECKPOINT-BASED SGD

Input: $\mathbf{u}^*$, $\mathbf{u}_0$, η

1: Initialize $\tilde{\mathbf{u}}_0 = \mathbf{u}_0$.
2: **for** $t = 1$ **to** T **do**
3: Update model state: $\mathbf{u}_t = \mathbf{u}^* + \eta(\mathbf{u}_{t-1} - \mathbf{u}^*) + \epsilon$
4: Compute distance: $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1}$
5: Quantize δ_t: $\tilde{\delta}_t = \text{QUANTIZE}(\delta_t)$
6: Compress $\tilde{\delta}_t$ by Huffman coding and save to disk
7: Update checkpoint state: $\tilde{\mathbf{u}}_t = \tilde{\mathbf{u}}_{t-1} + \tilde{\delta}_t$
8: **end for**

Output: $\mathbf{u}_T$, $\{\tilde{\delta}_t \mid t \in [T]\}$

3. LC-Checkpoint-based SGD

We now describe our solution LC-Checkpoint (LC refers to Lossy Compression). See Figure 1 for a working example and Algorithm 1 for a workflow. For simplicity, we assume that our system maintains a checkpoint $\tilde{\delta}_t$ for each iteration. We slightly abuse δ_t to refer to both the compressed data and the real vector it represents. It is straightforward to downsample our operations to construct a checkpoint every k-iterations. Our solution consists of two major components.

C1. Approximate tracking by delta-coding. At each step, our system maintains an approximation $\tilde{\mathbf{u}}_t$ of the ground-truth state. We simply set $\tilde{\mathbf{u}}_t = \mathbf{u}_0 + \sum_{i \le t} \tilde{\delta}_i$, where $\mathbf{u}_0$ is the initial state of the model. Our system continuously maintains and updates $\tilde{\mathbf{u}}_t$ at the background (line 7 in Algorithm 1). Our major compression task is to properly track the "delta" between the approximate state and ground-truth. Specifically, the compression task for the t-th iteration is $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1}$. See ③ in Figure 1.

C2. Quantization and Huffman coding. This component compresses δ_t through two steps, *Step 1. Two-stage quantization.* We first perform an exponent-based quantization, and then a priority promotion operation. This operation intelligently drops inconsequential information between two consecutive states. *Step 2. Lossless compression by Huffman.* Finally, the quantized distance vector is further compressed using Huffman coding.

One can see that to reconstruct the model state at iteration t from the checkpoints, we may simply compute $\mathbf{u}_t = \mathbf{u}_0 + \sum_{i=1}^{t} \tilde{\delta}_t$.

In what follows, Section 3.1 discusses C2 and Section 3.2 discusses additional system-level optimizations.

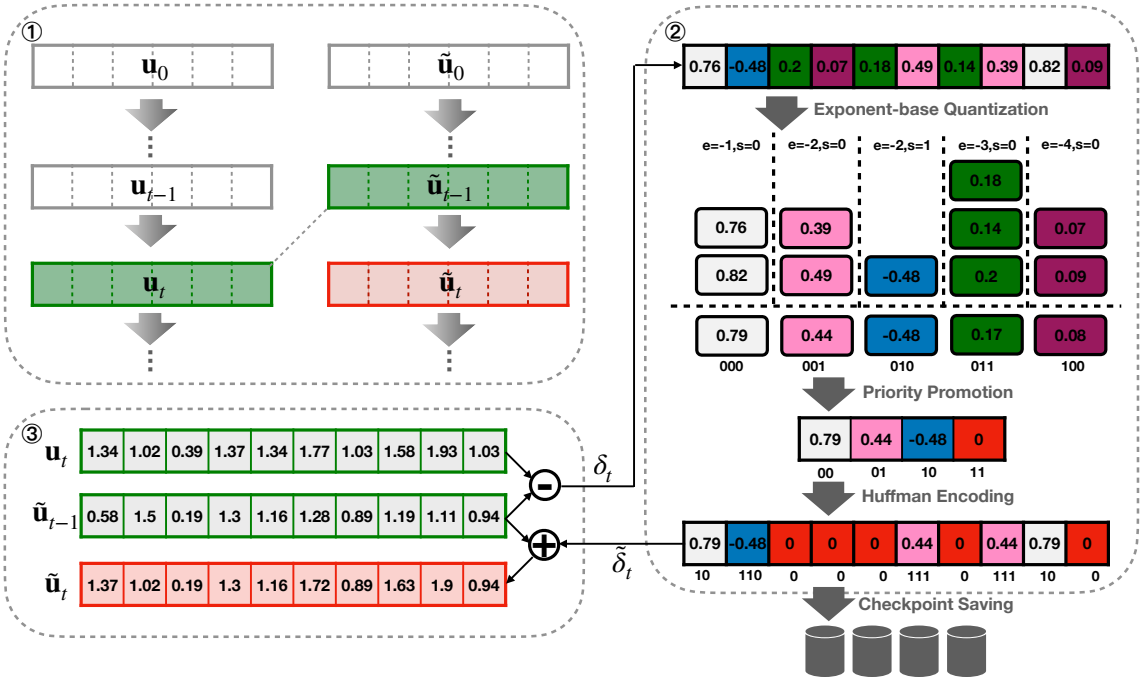


Figure 1. **LC-Checkpoint overview.**

3.1. Quantization and Huffman coding

3.1.1. Two-stage Quantization

LC-Checkpoint employs a novel two-stage pipeline to quantize δ_t, which consists of two main sub-steps: exponent-based quantization and priority promotion.

Exponent-based Quantization. Recall that a floating point v is represented by $v = (-1)^s \times m \times 2^e$, where s is the sign, m is the mantissa, and e is the exponent. Recall that $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1} \in \mathbf{R}^n$ is a high-dimensional vector we aim to encode. Our exponent-based quantization works as follows: first, it partitions entries in δ into multiple buckets according to e and s, i.e., it assigns the elements with identical exponents and signs to the same bucket. Our crucial observation from extensive experiments is that entries in $\mathbf{u}_t$ usually drift towards the same direction, so δ_t typically have the same sign. Next, our algorithm represents each bucket by the average of maximum and minimum values in the bucket.

Figure 1 ② shows an example, in which, δ_t is quantized into five buckets (marked with five different colors). All entries in each bucket are then represented by a unique value.

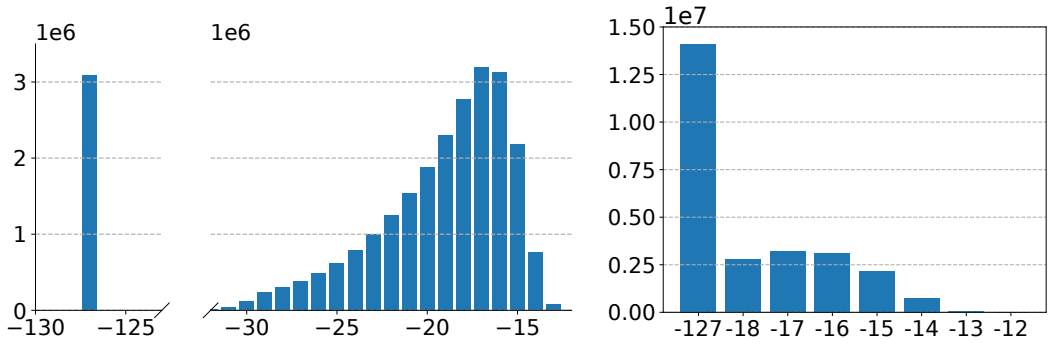
Indexing k buckets requires $\log_2 k$ bits. Because δ_t consists of n floating points, each of which uses b (e.g., $b \in \{32, 64\}$) bits, the compression rate is $r = \frac{nb}{n \log_2 k + kb}$.

For example, in Figure 1, δ has 10 elements (i.e., $n = 10$), each of which is represented by a single-precision floating point (i.e., $b = 32$). Thus, the original δ has nb, i.e., 320 bits in total. Exponent-based quantization uses 5 buckets (i.e., $k = 5$). Thus, after quantization, δ has $(10 \times \log 5 + 5 \times 32 = 190)$ bits. Therefore, the compressing rate (r) is 1.68 (i.e., $320/190$).

It is critical to control the number of buckets k to achieve an optimal compression ratio. Fortunately, the exponent-based bucketing can control $k \leq 2^9$ for single-precision floating point elements, and control $k \leq 2^{12}$ for double-precision. [1] Our evaluation results (Section 4.3) confirm that usually $k < 2^5$ suffices. Figure 2(a) plots the distribution of all elements' exponent parts in the last convolutional layer of AlexNet.

Priority Promotion. We further improve the compression ratio by limiting the number of buckets with a priority promotion approach. Our crucial observation is that when $\delta_{t,i}$ is excessively close to 0 (i.e., $\tilde{\mathbf{u}}_{i,t-1}$ is close $\mathbf{u}_{i,t}$), it is more effective to batch the updates (i.e., do not update the i-th entry of δ_t until it becomes substantial). Note also this is conceptually different from minimizing construction errors. Minimizing construction errors corresponds to exactly keeping track of the heaviest entries in δ_t, whereas we both remove excessively small entries and quantize large entries

[1] Single-precision floating point numbers use 8 bits to store e, and together with a sign bit—that is why $k \leq 2^9$. Similarly, double-precision numbers use 11 bits to store e.



(a) Exponent distribution of δ. (b) Exponent distribution of $\tilde{\delta}$ (3-bit promotion).

Figure 2. **The distribution of all elements' exponent parts in the last convolutional layer of AlexNet.** When e equals -127, the element value is 0. The x-axis denotes the exponent part value, and the y-axis indicates the count of elements with this value.

(as done in the previous step). Specifically, we propose x-bit priority promotion. It keeps $2^x - 1$ buckets with larger e only and merges the rest buckets into one with a unique value of 0. In other words, priority promotion updates $\tilde{\mathbf{w}}_i$ with a larger distance to $\mathbf{w}_i$ with a higher priority. It limits the index of buckets within x bits.

Figure 1 (Priority Promotion) uses 2-bit priority promotion to control the number of buckets under 4. It merges the green and purple buckets into a red one that is represented by a value 0. Indexing these buckets only needs 2 bits. Figure 2(b) gives a real example of 3-bit priority promotion for the last convolutional layer in AlexNet.

3.1.2. Huffman Coding

Finally, observing the number of elements in each bucket is highly non-uniform in most learning processes, we use Huffman coding (Van Leeuwen, 1976) to further compress the bucket. For example, Figure 2(a) plots the distribution of all elements' exponent parts in the last convolutional layer of AlexNet. This distribution shows a skewed behavior,

thus more suitable for Huffman coding. Our crucial observation is that priority promotion *further aggravates* the skewness of this distribution (Figure 2(b)), thus marrying quantization with Huffman coding produces more than "sum of parts" benefits. Our later evaluation validates it (Section 4.3).

3.2. System Optimizations

LC-Checkpoint also comprises several novel system-level optimizations as follows:

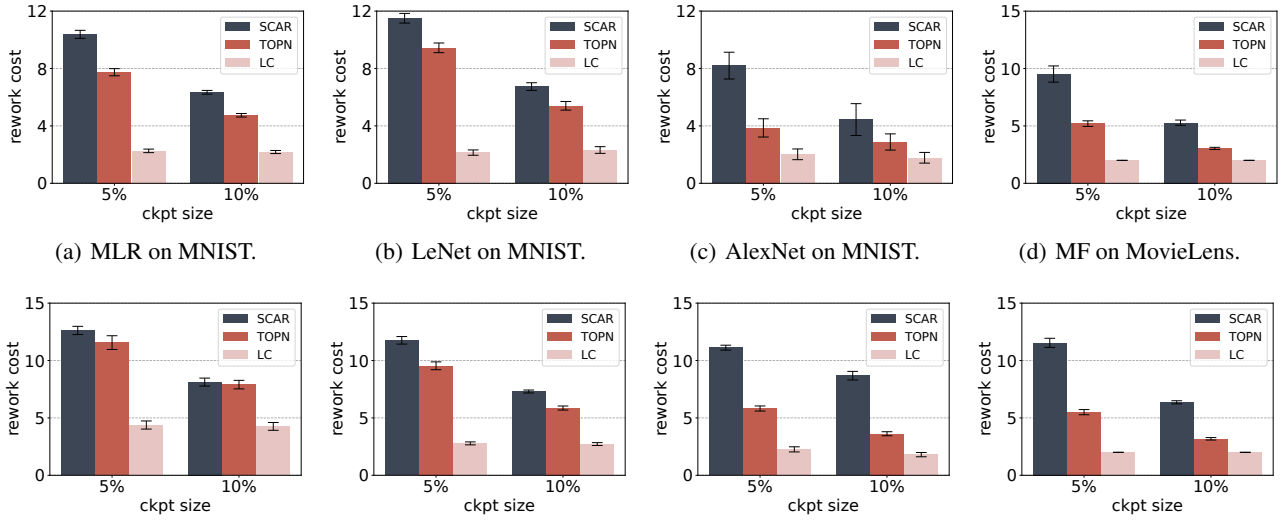
- **Asynchronous Execution:** Because only the first step of LC-Checkpoint depends on the model state, the rest steps can run simultaneously with the next iteration of SGD computation. This asynchronous (non-blocking) execution significantly reduces the checkpoint overhead, and mitigates the blocking of model execution.
- **Checkpoint Merging:** To further reduce the recovery time, LC-Checkpoint employs a helper process to merge multiple checkpoints into *super-step* ones, periodically. In case of any system crash, LC-Checkpoint uses these *super-step* checkpoints for recovery.
- **Huffman Code Table Caching:** The number of buckets may stay the same from one iteration to another, specifically after priority promotion. Thus, it is possible to reuse the Huffman code table (with only a simple sort of buckets according to the number of entries in each bucket) among different iterations without any rebuilding. LC-Checkpoint comprises a lightweight cache to store the Huffman code table for each buckets count.

4. Experiments

This section evaluates LC-Checkpoint on four typical ML applications with three benchmark datasets, and compares it with previous efforts (SCAR Qiao et al. (2018b) and a TOPN mechanism as mentioned in Section 2) on recovery (rework) cost, compression ratio, and execution overhead, demonstrating the superiority of LC-Checkpoint.

4.1. Methodology

Evaluation Objective: This evaluation has four main objectives: (1) comparing LC-Checkpoint' recovery (rework) cost with previous work; (2) evaluating the compression benefits brought by different approaches mentioned before; (3) specifically, validating the effectiveness of priority promotion; (4) confirming that LC-Checkpoint incurs low overhead by an experiment case study. Our work is mainly compared with two state-of-the-art efforts: SCAR (Qiao et al., 2018b) and a TOPN mechanism. SCAR partitions



(a) MLR on MNIST. (b) LeNet on MNIST. (c) AlexNet on MNIST. (d) MF on MovieLens.

(e) MLR on FashionMNIST. (f) LeNet on FashionMNIST. (g) AlexNet on FashionMNIST. (h) MF on Jester.

Figure 3. **Rework cost comparison among LC-Checkpoint, SCAR, and TOPN.** The x-axis indicates the ratio of the compressed checkpoint size over the full checkpoint size. The y-axis shows the rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 50 times.

the parameters and updates one partition in each iteration to reduce the checkpoint size. The TOPN mechanism only updates the parameters with the top-n largest distances to the previous iteration. The TOPN checkpoint is stored in a compressed sparse row (CSR) format.

ML Applications and Datasets: LC-Checkpoint is evaluated on four typical ML applications: Multinomial Logistic Regression (`MLR`), LeNet-5 (`Lenet`) (LeCun et al., 1998), `AlexNet` (Krizhevsky et al., 2012) and Matrix Factorization (`MF`). The first three applications are trained on `MNIST` (LeCun et al., 1998) and `FashionMNIST` (Xiao et al., 2017) datasets. The last one, `MF` is trained on `Jester` (Goldberg et al., 2001) and `MovieLens10M` (Harper & Konstan, 2015).

Platforms and Evaluation Configurations: Our experiments are conducted on a multi-core server with an Intel Xeon Gold 6138 Skylake CPU with 40 cores, each running at 2.0 GHz, and 192 GB DDR4 memory. The training is performed on a Tesla P100 GPU with 16GB High-bandwidth Memory (HBM).

4.2. Recovery/Rework Cost Comparison

This section evaluates the recovery (or rework) cost of LC-Checkpoint, particularly comparing it to SCAR (Qiao et al., 2018b) and a TOPN mechanism[2].

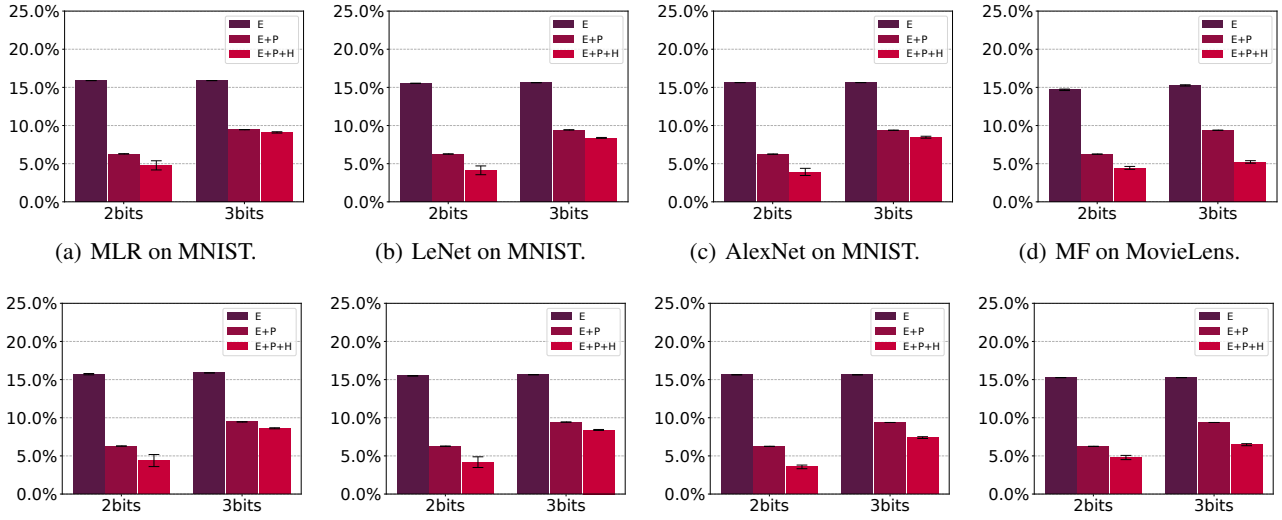
To evaluate their rework costs fairly, we use the same checkpoint size (update size) for all three methods. Two checkpoint sizes are tested: 5% and 10% of the full checkpoint size[3]. These checkpoint sizes can be set directly for SCAR and TOPN. However, LC-Checkpoint's size is determined by the data distribution and thus changed dynamically. To address this issue, LC-Checkpoint employs 2-bit and 3-bit priority promotion that control its checkpoint size at 5% and 10%. Figure 4 reports more details of LC-Checkpoint's checkpoint size information.

Figure 3 compares the rework cost of three methods, SCAR, TOPN, and LC-Checkpoint, showing that LC-Checkpoint incurs the lowest rework cost for all ML applications and datasets among them. For the 5% checkpoint test case, LC-Checkpoint outperforms SCAR by $2.88\times$-$5.77\times$, and TOPN by $2.17\times$-$4.06\times$, respectively. With 10% checkpoint size, LC-Checkpoint outperforms SCAR by $1.9\times$-$4.82\times$, and outperforms TOPN by $1.52\times$-$2.17\times$, respectively.

In addition, comparing two checkpoint sizes (5% v.s. 10%), LC-Checkpoint results in more stable rework cost as the checkpoint size decreasing. For example, decreasing the checkpoint size from 10% to 5%, LC-Checkpoint has a negligible rework cost increase on `LeNet` with `MNIST` (Figure 3(b)) and `AlexNet` (Figure 3(c), 3(g)). It does not have any rework cost change for other cases. In contrast, SCAR and TOPN increase $1.6\times$ rework cost on average as the checkpoint size changing from 10% to 5%.

[2]Rework (or recovery) cost is defined as the number of iterations from $\tilde{\mathbf{u}}_t$ to $\mathbf{u}_t$. All methods share the same SGD computation cost for each iteration.

[3]Full checkpoint stores all model parameters after a specific iteration.



(a) MLR on MNIST. (b) LeNet on MNIST. (c) AlexNet on MNIST. (d) MF on MovieLens.

(e) MLR on Fashion-MNIST. (f) LeNet on Fashion-MNIST. (g) AlexNet on Fashion-MNIST. (h) MF on Jester.

Figure 4. **The compression ratio with different compression methods.** The x-axis denotes the bits count used in priority promotion, and the y-axis is the ratio of the checkpoint size after compression over the one before compression. E, P, H denote "exponent-base quantization", "priority promotion", and "Huffman coding", respectively.

4.3. LC-Checkpoint Compression Effect Breakdown

This section evaluates and analyzes the compression effect of different approaches mentioned before, exponent-base quantization (`E`), priority promotion (`P`), and Huffman coding (`H`). Figure 4 reports the compression ratios with 2-bit and 3-bit priority promotion. With all compression approaches, the ultimate checkpoint sizes (`E+P+H`) are all below 5% with 2-bits, and below 10% with 3-bits over the uncompressed full checkpoint, i.e., the compression rates are above $20\times$ and $10\times$, respectively.

Exponent-base quantization yields a compression ratio of 85% on average. It proves that the exponent parts of all parameters in δ span across a small range of all values that can be represented by single precision floating-point. 15% also indicates that the bucket number $k < 2^5$, because the average bucket number can be estimated as $k = 2^{(32\times 15\%=4.8)}$, where 32 is the width of single precision floating-point. Priority promotion brings 9.26% extra compression ratio on average for 2-bit and 6.23% for 3-bit. For most cases, priority promotion with smaller bits yields more benefits for Huffman coding except MF (Figure 4(d), 4(h)). This is because MF's parameters are sparse, thus Huffman coding can reach a sufficient compression ratio without aggressive priority promotion. Across all models (and datasets), Huffman coding brings 2% extra compression ratio with 2-bits priority promotion, and 1.6% with 3-bits one on average.

4.4. The Effectiveness of Priority Promotion

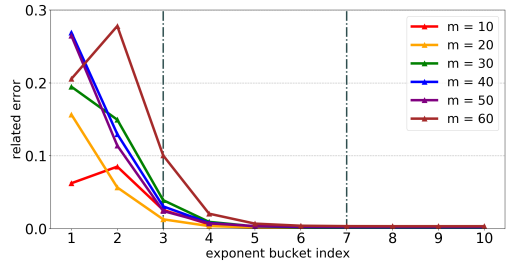
This section further discusses the effectiveness of priority promotion. It aims to prove that priority promotion is able to save the majority of high priority parameters. We prove it by showing the exponent buckets result in a larger impact on the model state when their represented unique values are further from 0 (i.e., e is larger).

Assume δ is calculated from one state $\mathbf{u}_\theta$ to another for m iterations. Then, δ_m^i is created by setting the parameters in the i-th exponent bucket to 0. The ground truth is calculated as $V_{gt} = L(\mathbf{u}_\theta + \delta_m)$ where $L(x)$ denotes the loss function. Then the relative error is calculated as:

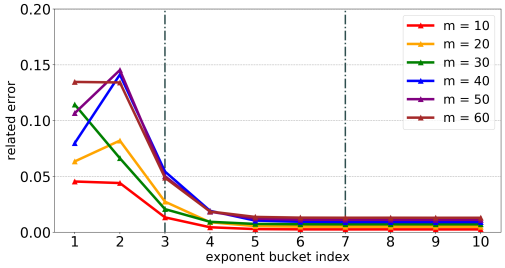
$$E_m^i = \frac{\left\| V_{gt} - L(\mathbf{u}_\theta + \delta_m^i) \right\|_2}{V_{gt}} \tag{2}$$

Figure 5 reports the result of `MLR` with $m = 10n, n \in [1, 6]$. Both datasets (`MNIST` and `FashionMNIST`) on varied m prove that the elements in the buckets with the top-n largest distance impact more on the model (denotes as a higher relative error when the bucket represented value is set to 0).

In addition, it is possible to preserve all *important* buckets with only a small number of index bits. For example, using 2-bit priority promotion (4 buckets with the last bucket storing 0) can easily preserve the most important buckets, and using 3-bit (8 buckets) can preserve all effective buckets. This result explains why priority promotion can compress the checkpoint with negligible accuracy loss.



(a) MLR on MNIST.



(b) MLR on FashionMNIST.

Figure 5. **Evaluation on the priority of each exponent bucket.** The x-axis denotes the id of the exponent bucket that is deleted. The y-axis shows the relative error to the ground-truth.

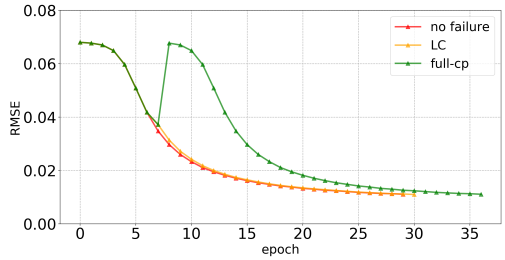


Figure 6. **MF on MovieLens25M.** The x-axis denotes the iteration and the y-axis is the model's RMSE (Root Mean Square Error).

4.5. A Case Study on LC-Checkpoint's Overhead

This section evaluates LC-Checkpoint's execution overhead and overall impact on the model execution using a case study, i.e., training `MF` on `MovieLens25M` (Harper & Konstan, 2015) dataset. Each iteration costs 91 seconds on average. LC-Checkpoint employs 3-bit priority promotion, resulting in a checkpoint size below 10% (of the uncompressed full checkpoint size). Default approach creates a full checkpoint every 10 iterations. A failure is triggered at the 7-th iteration.

Figure 6 reports the result. LC-Checkpoint only incurs one extra iteration than the normal execution without any failure to convergence, and saves 6 iterations compared to the full checkpoint method, i.e., saving 546 seconds execution time. LC-Checkpoint introduces only less than 4 seconds (i.e., around 4%) overhead for each iteration, which is negligible.

5. Related Work

Fault-tolerance is a key fundamental support for ML systems. Li et al. (Li et al., 2014) propose a runtime parameter replication approach for recovery. Tensorflow (Abadi et al., 2016) employs periodic checkpoint to save the model state. Other efforts like (Harlap et al., 2017; Qiao et al., 2018a) aim to support strong consistency semantics. In contrast, our work relaxes the consistency guarantee of checkpoint based on the self-correcting behavior of ML applications. With a set of lossy compression mechanisms, our work can afford high frequent checkpoints, resulting in low rework cost and fine-grained model state recovery. Similarly, Qiao et al. (Qiao et al., 2018b) also propose a fault-tolerant solution (SCAR in our evaluation) based on weak consistency by partially updating parameters. SCAR is potential to store redundant information during checkpointing according to our evaluation, and our work aims to eliminate such redundancy by selectively saving the distance between two states.

Model compression has been proposed to reduce model storage space and accelerate model execution time, simultaneously. Weight pruning and weight quantization are two important categories of model compression.

Some popular weight pruning techniques closely related to our work are summarized as follows. Guo et al. (Guo et al., 2016) present a dynamic network surgery approach with on-the-fly connection pruning to reducing the network complexity. Dai et al. (Dai et al., 2019) combine the growth and the pruning phases in training to generate compact DNN architectures. Han et al. (Han et al., 2015b) design Deep Compression, a model compression approach by combining pruning, quantization, and Huffman coding. Mao et al. (Mao et al., 2017) carefully explore the impact of varied pruning granularity on model accuracy and propose a coarse-grained weight pruning approach. All effort above aims to prune model weights without compromising accuracy. Different from them, our work eliminates the redundancy between two checkpoints and reduces the rework cost during recovery by designing a reliable coding scheme working throughout the entire dynamic process of learning.

Weight quantization is also widely used for model compression. BinaryConnect (Courbariaux et al., 2015) introduces the binary weight for replacing multiplication by addition and subtraction. Binarized Neural Networks (Courbariaux et al., 2016) also use binary weights and activations to accelerate computation. Park et al. (Park et al., 2017) propose a clustering method based on weighted entropy for weight quantization. Leng et al. (Leng et al., 2018) formulate quantization as an optimization problem and solve it by ADMM. Our approach also employs quantization to reduce the bits of

parameters by designing a novel exponent-based quantization technique. Moreover, our approach emphasizes filtering the parameters with a new priority promotion method.

6. Conclusion and Future Work

This paper presents LC-Checkpoint, the **first** checkpoint scheme based on lossy compression to achieve the maximal compression rate and efficient recovery simultaneously. It employs a novel two-stage quantization method consisting of exponent-based quantization and priority promotion to identify and store the most critical information for SGD to recover, and leverages Huffman coding to further benefit from the non-uniform distribution of gradient scales. Our evaluation demonstrates that LC-Checkpoint achieves a compression rate up to $28\times$ and recovery speedup up to $5.77\times$ over the state-of-the-art algorithm (`SCAR`).

In the future, we plan to generalize LC-Checkpoint by relaxing the assumption of SGD and equipping it with the capability of selecting checkpoint compression rates dynamically according to model and data changes.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work is supported in part by NSF grants CRII-1755646, CRII-1755769, OAC-1835821, CNS-1813487 and CCF-1918757, a Google Faculty Research Award, and an AWS Machine Learning Research Award.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pp. 1709–1720, 2017.

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1387–1395, 2017.

Boyd, S. and Vandenberghe, L. *Convex optimization*. Cambridge university press, 2004.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Dai, X., Yin, H., and Jha, N. K. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. Eigentaste: A constant time collaborative filtering algorithm. *information retrieval*, 4(2):133–151, 2001.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Guo, Y., Yao, A., and Chen, Y. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems*, pp. 1379–1387, 2016.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015b.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 589–604, 2017.

Harper, F. M. and Konstan, J. A. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

Hong, M., Luo, Z.-Q., and Razaviyayn, M. Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. *SIAM Journal on Optimization*, 26(1):337–364, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Ivkin, N., Rothchild, D., Ullah, E., Stoica, I., Arora, R., et al. Communication-efficient distributed sgd with sketching. In *Advances in Neural Information Processing Systems*, pp. 13144–13154, 2019.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Kushner, H. and Yin, G. G. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

Lai, T. L. Martingales in sequential analysis and time series, 1945–1985. *Electronic Journal for history of probability and statistics*, 5(1), 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Leng, C., Dou, Z., Li, H., Zhu, S., and Jin, R. Extremely low bit neural network: Squeeze the last bit out with admm. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 583–598, 2014.

Lin, D., Talathi, S., and Annapureddy, S. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pp. 2849–2858, 2016.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*, 2012.

Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 181–194, 1997.

Park, E., Ahn, J., and Yoo, S. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5456–5464, 2017.

Qiao, A., Aghayev, A., Yu, W., Chen, H., Ho, Q., Gibson, G. A., and Xing, E. P. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 631–644, 2018a.

Qiao, A., Aragam, B., Zhang, B., and Xing, E. P. Fault tolerance in iterative-convergent machine learning. *arXiv preprint arXiv:1810.07354*, 2018b.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pp. 525–542. Springer, 2016.

Van Leeuwen, J. On the construction of huffman trees. In *ICALP*, pp. 382–410, 1976.

Woodruff, D. P. et al. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4820–4828, 2016.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.