DQRM: Deep Quantized Recommendation Models

Anonymous Author(s)

ABSTRACT

Large-scale recommendation models are currently the dominant workload for many large Internet companies. These recommenders are characterized by massive embedding tables that are sparsely accessed by the index for users and item features. The size of these 1TB+ tables imposes a severe memory bottleneck for the training and inference of recommendation models. In this work, we propose a novel recommendation framework that is small, powerful, and efficient to run and train, based on the state-of-the-art Deep Learning Recommendation Model (DLRM). The proposed framework makes inference more efficient on the cloud servers, explores the possibility of deploying powerful recommenders on smaller edge devices, and optimizes the workload of the communication overhead in distributed training under the data parallelism settings. Specifically, we show that quantization-aware training (QAT) can impose a strong regularization effect to mitigate the severe overfitting issues suffered by DLRMs. Consequently, we achieved INT4 quantization of DLRM models without any accuracy drop. We further propose two techniques that improve and accelerate the conventional QAT workload specifically for the embedding tables in the recommendation models. Furthermore, to achieve efficient training, we quantize the gradients of the embedding tables into INT8 on top of the well-supported specified sparsification. We show that combining gradient sparsification and quantization together significantly reduces the amount of communication. Briefly, DQRM models with INT4 can achieve 79.07% accuracy on Kaggle with 0.27 GB model size, and 81.21% accuracy on the Terabyte dataset with 1.57 GB, which even outperform FP32 DLRMs that have much larger model sizes (2.16 GB on Kaggle and 12.58 on Terabyte). All the code and experiments are open-sourced anonymously at: https://anonymous.4open.science/r/Deep_Quantized_Recommendation_Model_DQRM-6B4D

CCS CONCEPTS

• **Computing Systems** → *Efficient Large-scale Neural Network Training*; *Distributed Training*; • **Networks**;

KEYWORDS

Neural Networks, Recommendation Systems, Quantization, Efficiency Distributed Training, Data Parallelism, Regularization

ACM Reference Format:
Anonymous Author(s). 2023. DQRM: Deep Quantized Recommendation Models. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the widespread adoption of Internet services, personalization becomes a critical function of the services provided by the current Internet giants. Every service user's unique taste and preferences need to be catered to. With billions of Internet users present today, the need for recommendation models is crucial. According to [8], over 79% of the entire Meta's cloud ML inference cycles are spent on the inference of various sizes of recommendation models. By Amdahl's law, a slight boost in recommendation model efficiency can yield a massive performance boost, which can be highly beneficial to large Internet companies. On the other hand, if part of the inference workload can be migrated to edge devices, Internet service providers can save precious cloud resources, while the users can have less of their personal data sent to the cloud environment, which strengthens their personal data privacy. Both the company and the end users benefit. To achieve less costly recommenders, in this work, we propose a deep quantized recommendation model (DQRM) framework, where the models are both more efficient in inference on the cloud environment, and possible to fit on edge devices. Moreover, to address the necessity for periodic retraining in recommenders, the proposed framework is optimized for model training in the cloud-distributed environment.

Specifically, designing deep learning based recommendation models is challenging, because of the necessity to both process dense and sparse inputs. Successful previous works [17, 23] utilize massive embedding tables, each corresponding to a sparse feature category. Although embedding tables are proven to learn sparse features well, they are usually massive, and easily take up many GB, and in some cases even TB in DRAM memory. The massive model size makes memory the prime bottleneck in both training and inference. This severe problem motivates a huge effort in shrinking recommendation model size [3, 6, 7, 13, 15, 21]. In our work, we base our model framework on the Deep Learning Recommendation Model (DLRM) [17], the state-of-the-art recommendation model for Click-Through-Rate (CTR) ([26]) prediction (as in Figure 1 (a)), and we apply quantization and sparsification to enhance its efficiency.

Based on experiments and analyses, we find that DLRM models suffer from severe overfitting issues, with testing accuracy dropping steeply after the first one or two epochs of training. We propose to heavily quantize the DLRM model into ultra-low INT4 precision, and we show that Quantization-aware Training (QAT) can greatly reduce the memory and computational costs while imposing a strong regularization effect to benefit DLRM training. As a result, we achieve comparable or even better testing accuracy than the full-precision models. Besides, our experiments indicate that conventional QAT can be highly inefficient in recommendation model quantization. To deal with this issue, we propose two techniques that reduce additional memory copies of large embedding tables during QAT and avoid the costly traversal of large tensors in memory to compute quantization scales. Our proposed methods significantly reduce time costs.

Besides the model size and inference costs, the training of recommendation models is also crucial in real-world applications. Recommenders are usually trained in highly distributed cloud environments, where the training time is dominated by heavy inter and intra-node communications overhead. In this work, we propose

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

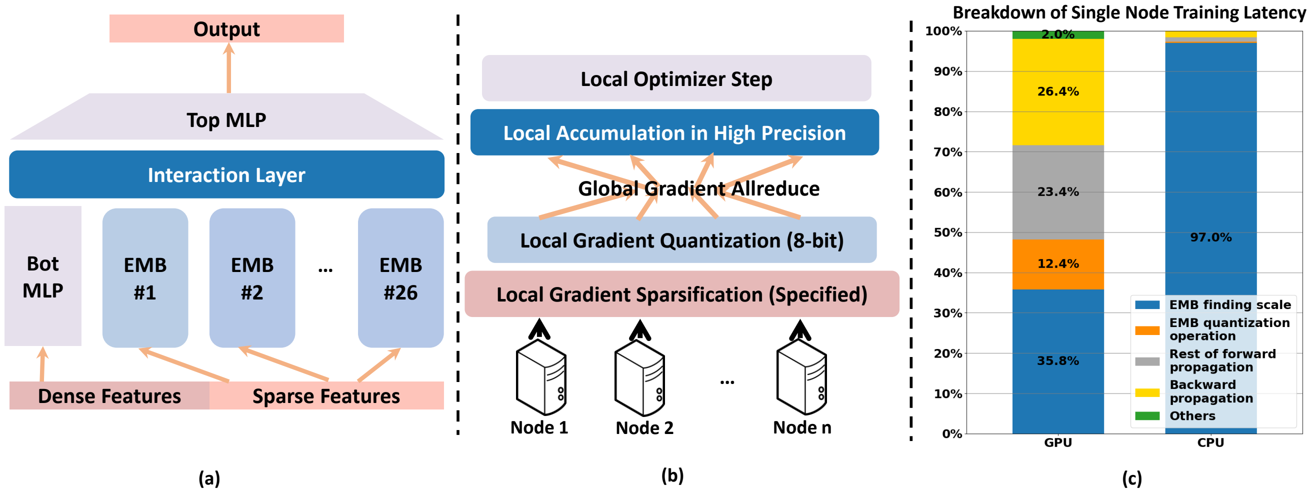


Figure 1: (a) shows the state-of-the-art large-scale recommendation model architecture. The model contains two types of layers: Embedding tables and MLP layers. (b) Our framework builds on top of specified sparsity and add quantization to achieve additional gradient compression ratio. (c) shows a breakdown of the single-machine training time running DQRM in INT4; the majority of the training time on the GPU platform (left) is spent on finding the quantization scales, and even more so on the CPU node (right) (Figure best viewed in color).

to apply communication sparsification and quantization jointly to shrink the workload. We show that using well-supported specified sparsification of gradients can compress the communication by three orders of magnitude. On top of that, we further quantize the communication from the massive embedding tables into INT8, shrinking the remaining size of communication by one-half during training, with a negligible accuracy degradation on the large Criteo Terabyte dataset. We summarize and show the entire backpropagation of the proposed framework in Figure 1 (b).

The contributions of this work are summarized as follows:

- We propose to apply ultra-low precision quantization to alleviate the overfitting of DLRMs meanwhile achieving smaller model sizes,
- We introduce two novel techniques to improve the speed of QAT on recommendation systems,
- We achieve efficient training by jointly applying sparsification and quantization on communications,
- Our DQRM models with INT4 achieve an 8× reduction in model size, while obtaining 79.071% accuracy (0.148% higher than DLRM FP32 baseline) on the Kaggle dataset, and 81.210% accuracy (0.045% higher than DLRM FP32 baseline) on the Terabyte dataset.

2 METHODOLOGY

2.1 Quantization-aware Training

In this work, we utilize uniform symmetric quantization. When quantizing a floating point number x, we calculate a mapping between x and an integer number I between $[-2^{b-1}, 2^{b-1}-1]$. Formally, quantization is defined as:

$$I = Q(x,k) = \frac{clip(x,-\alpha,\alpha)}{S}$$

where Q is the quantization function and S is the quantization scale. In this work, we use uniform quantization where S is determined using the following formula:

$$S = \frac{\alpha}{2^{b-1}-1}$$

In both of the above equations, the term α is the clipping range of quantization, usually determined by the minimum and the maximum of the distribution of the model weights, X. In this work, we directly use the maximum of $abs(max(x))$ and $abs(min(x))$ to compute the clipping range α. In experiments, we found that the minimum and the maximum of the embedding weights are constantly growing, widening the range of clipping continuously. A more detailed study can be found in Appendix C.2. Percentile clipping adds additional overhead in embedding table traversal, which, as we show later, is a big challenge for quantizing embedding tables. On the other hand, the trained clipping range may be sub-optimal for the moving target.

We are aware of previous works [7] that use a non-uniform codebook quantization for quantizing embedding tables. Although codebook quantization theoretically can find a better fit for an arbitrary weight distribution, without complex implementation support, it slows down the process of quantization. This time cost is tolerable in Post-Training Quantization (PTQ) which [7] is based on, but it is indeed significant in our work which uses Quantization-aware Training (QAT). During every iteration of the fine-tuning process in conventional QAT, a copy of full precision weights is always stored, which is updated by the gradients in the back-propagation. After updating the full precision weights from the previous iteration, the model is quantized again based on the full precision model from the previous iteration. However, when training large-scale DLRM models in particular, having both the quantized and the full

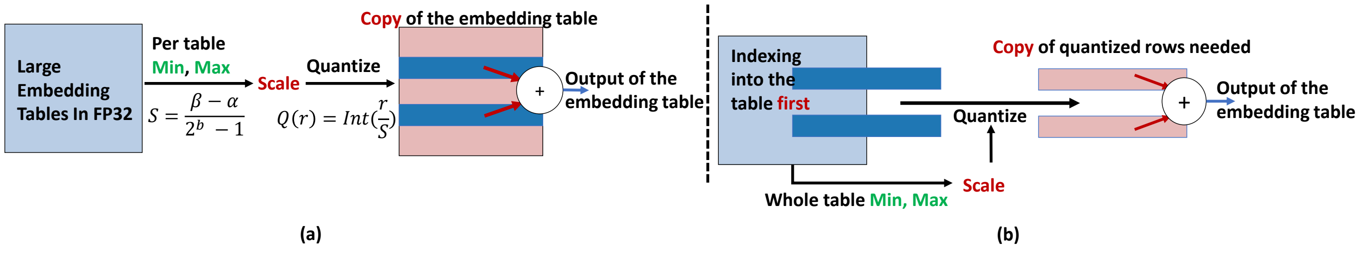

Large Embedding Tables In FP32
Per table Min, Max
Scale
$S = \frac{\beta - \alpha}{2^b - 1}$
Quantize
$Q(r) = Int(\frac{r}{S})$
Copy of the embedding table
+
Output of the embedding table
(a)
Indexing into the table first
Copy of quantized rows needed
Quantize
+
Output of the embedding table
Whole table Min, Max
Scale
(b)

Figure 2: (a) Conventional QAT method, where the entire set of weights of the embeddings are copied and quantized. As embedding accesses are very sparse, this method is wasteful as it processes unused entries and worsens the memory bottleneck in QAT. (b) Our method to avoid the massive copy is by first performing the functional part of each table and then performing copying and quantization. In this way, only part of the weights that are actually used is copied and quantized, utilizing the memory more efficiently (Figure best viewed in color).

precision model copies introduces additional memory costs, which Section 2.2 addresses.

2.2 Reducing Memory of Unused Weights

Unlike other popular neural network architectures such as CNNs or transformers, embedding tables occupy over 99% of the entire model size for DLRM-style recommenders. From these giant embedding tables, previous works [6, 15] have shown that training large-scale recommendation models is usually memory-bound instead of compute-bound. Additionally, we find that when running the conventional QAT workflow on DLRM näively, the existing memory bottleneck is exacerbated even further.

As shown in Figure 2 (a), conventionally, all the weight tensors inside convolutional layers or linear layers contribute to the computation in subsequent layers of the model. Therefore, the entire weight tensor in the quantized layer needs to be quantized. As mentioned previously, preserving a copy of full precision weights is required in QAT. Therefore, we now have a full-precision and a low-precision quantized shadow copy of the same tensor. If such a method is näively applied to embedding table quantization, this entire giant embedding table needs to be copied in memory, which is often unnecessary because of the fundamentally different sparse workflow where not all the embedding weights contribute to the subsequent stages of computation.

Instead, we propose to selectively run the sparse functional portion of the embedding layer first, indexing into the embedding tables and retrieving the embedding vectors that are actually required in the following layers in the current iteration. Then, only quantize the embedding vectors that are needed. In the backward pass, the gradient can still update the full precision embedding tables. The key is that throughout the entire iteration, only active embedding table entries are copied, and depending on the batch size and the power law distribution within the training dataset, it is usually more than three magnitudes smaller than the number of embedding vectors in the embedding tables on average.

2.3 Periodic Update to Find Quantization Scale

The giant embedding tables in DLRM-like large-scale recommendation models also pose another big challenge to the conventional quantization workflow. In quantization, the first step is usually finding the quantization scale. Since we use uniform quantization and determine the clipping range for quantization based on embedding table weights on that iteration, the maximum and the minimum values of the entire embedding table must be found during the forward pass of the QAT. Given the large embedding table size, this step is very costly. Shown in Figure 1 (c), we run a breakdown of training time under the single-node training setting. We profile both single GPU and single CPU node. We train the model under the Kaggle dataset settings on a single Nvidia A5000 GPU. Interestingly, without quantization in forward propagation, backward propagation dominates the single-node total training time: 5.798 ms for backprop vs 2.737 ms for forward. Contrastively, weight quantization significantly enlarged forward time and outweighs the backprop. Shown in the left pillar, only yellow portion signifies the backprop, while the rest all comprises the forward. Especially, finding the scale of large embedding tables occupies more than one-third of the entire training time. The long time is due to the difficulties of large tensor traversal.

We also run experiments of Terabyte dataset on a single Intel(R) Xeon(R) Platinum 8280 CPU node. We found that such a problem significantly magnifies on the CPUs. Finding the quantization scale for large DLRM models under the Terabyte dataset settings can take more than 1000 ms per iteration of QAT, while occupying the majority of the training time. The Terabyte dataset have embedding tables that are ten times larger than those on Kaggle dataset. We found that periodic update of the quantization scale works effectively in solving the huge lag of computing the quantization scale. Periodic updates both amortize the huge quantization overhead to hundreds of iterations and are found not to hurt model convergence. In fact, periodic update in embedding table quantization can even boost the performance slightly. Detailed experiments and evaluations are presented in Section 4.1.

2.4 Distributed Training with Gradient Quantization

With DQRM, we can train powerful click-through-rate models in low-precision efficiently. We find out that, however, when training using low-precision data types, DLRM converges much slower than using standard single-precision floating point. DQRM usually requires a few more epochs to converge during training. Also, even though DQRM time is significantly reduced compared to the initial unoptimized implementation, it still has additional compute overhead in quantization, taking slightly longer than normal training. To make DQRM more efficient and competitive in training time compared with normal training, we further optimize the communication workload that occurs in every iteration.

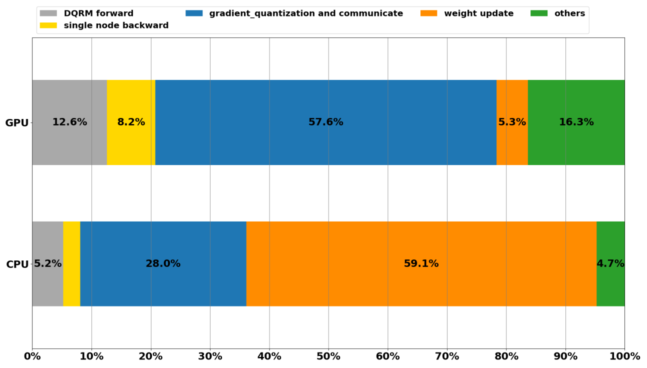


Figure 3: A breakdown of training time on distributed data parallelism environment on multiple GPUs and CPU nodes

In Figure 3, we profile the total time spent during the entire training process when using the PyTorch built-in specified sparsity on Embedding tables. During the profile, we train DQRM Kaggle across four Nvidia A5000 GPUs and DQRM Terabyte across two Intel Xeon Platinum 8280 CPUs with update period set to 1000 to completely eliminate the difficulties of finding the scale. Under the pure Data Parallelism settings, gradient communication (blue) is a severe bottleneck for both GPU and CPU platforms. Notice that weight update (orange) portion of CPU platforms is significant, because the vanilla PyTorch library we used is highly inefficient in sparse tensor operations, and the inefficiency in arithmetic will be largely reduced by the recently published IPEX[1] package from Intel. Besides the significant weight update (orange) portion which can be optimized relatively easily, we argue that the gradient communication (blue) is still a severe bottleneck on CPU platform.

Inspired by previous sparsification [9] and quantization [24] studies on DLRM models in distributed training message compression, we strive to add gradient quantization on top of heavy gradient sparsification for embedding tables to achieve additional compression ratio. We propose to quantize all gradients into fixed-point INT8 format. However, naively doing so is challenging. Also, naively performing error compensation to all parts of the weights can also be highly inefficient due to the enormous error buffer needed for the large embedding tables. Adding large error buffers will greatly increase model size stored in memory during training. On the contrary, we identify surprisingly that gradients in MLP layers are more sensitive to quantization than those in embedding tables, and we propose to use the Error Compensation technique similar to [24] to reduce accuracy degradation incurred by gradient quantization. We achieve reasonable accuracy degradation while compressing the communication message size by roughly 4X on top of the already heavy and well-supported gradient sparsification added to embedding tables. Detailed experiment settings, results, and evaluations are presented in Section 3.3.

[1]IPEX official github repo: https://github.com/intel/intel-extension-for-pytorch

3 EXPERIMENTAL RESULTS

In this section, we present results evaluating the DQRM framework on two commonly used click-through rate recommendation datasets: the Criteo Kaggle Display Advertising Challenge Dataset (shortened below as the Kaggle dataset) and the Criteo Terabyte Dataset (shortened as the Terabyte dataset). The original DLRM work [17] proposes two different configurations that are optimized specifically for each dataset, which we followed in our implementation of DQRM. Section 3.1 details experiments on embedding table quantization alone evaluated on the Kaggle dataset. Section 3.2 evaluates DQRM on the Kaggle dataset and on the Terabyte dataset under the distributed settings. Section 3.3 evaluates DQRM's effect on communication compression over multiple GPUs and multiple node CPU server environments.

3.1 Quantization of Embedding Tables

Previous works [6, 8] have found that the embedding tables occupy over 99% of the entire DLRM model size. Therefore, compressing embedding tables is important to the DLRM model size compression. The experimental results presented in the section are evaluated on the Kaggle Dataset. The specific DLRM model architecture we used follows official DLRM settings and is in Table 1.

Table 1: DLRM model architecture configurations for Kaggle and Terabyte Datasets

Model Specifications	Kaggle	Terabyte
# Embedding Tables	26	26
Maximum # Row among Tables	10131227	9994101
Embedding Feature Size	16	64
Bottom MLP Arch	13-512-256-64-16	13-512-256-64
Top MLP Arch	512-256-1	512-512-256-1

We used one Nvidia A5000 GPU to study the effect of quantization on the embedding tables. We investigate different bit widths for embedding table quantization: INT16, INT8, INT4. Unlike previous works [3, 7] that utilize a row-wise scheme for embedding table quantization, we quantize the embedding table with a per-table scheme. DLRMs with quantized embedding tables under different settings are all trained from scratch for 5 epochs using QAT. Figure 4(a) shows the curves of the testing accuracy during 5 epochs of QAT using different quantization bit widths. The vertical dashed lines signify the boundaries of each epoch. Through our experiments, we found that DLRM in single precision (blue) suffers from severe overfitting after the second epoch, and the testing accuracy drop precipitously afterward. Similarly, when the embedding tables are quantized into INT16 and INT8, the testing accuracies of INT16 (orange) and INT8 (grey) quantization follows the same overfitting trend. In contrast, INT4 quantization (green), although converges slower at the beginning, overcomes overfitting and continues to

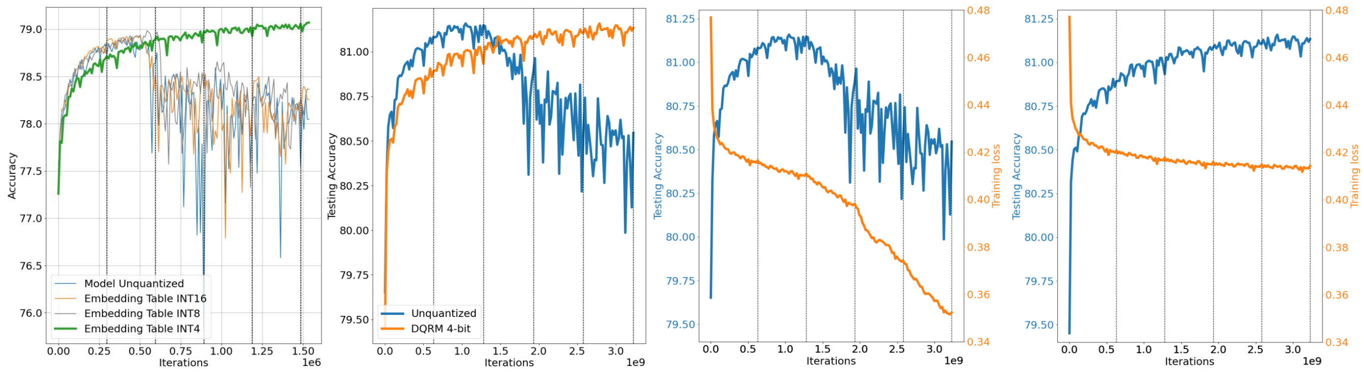


Figure 4: (a) shows the effect of using different QAT bit widths on quantizing embedding tables in DLRM for five epochs of training (epochs are separated by the black dashed lines in all figures). QAT in uniform 4-bit overcomes the severe overfitting suffered by the original DLRM training and leads to significantly higher testing accuracy over five epochs of training. (b) shows the comparison between DQRM 4-bit compared to normal training on the Terabyte dataset; DQRM, with significantly smaller model size, achieve on par test accuracy as DLRM FP32 model through better overcoming the overfitting problem. (c) shows that the training loss (orange curve) for normal training starts decreasing drastically in the third epoch, right where the overfitting occurs. In (d), the training loss curve for 4-bit DQRM decreases stably throughout five epochs of training.

increase over the entire five epochs. INT2 quantization is also tested. Although having a similar trend as INT4, it always incurs much significant accuracy degredation than INT4.

We compare the performance after embedding table quantization in Table 2. Bit widths of INT16 and INT8 lead to comparable performance to the original full-precision DLRM model. Uniform INT4 quantization outperforms the original model in testing accuracy by roughly 0.15% and ROC AUC score by 0.0047, significant in the Kaggle dataset. From experiment results, we show that uniform INT4 quantization is capable of preserving (even slightly outperforming) the original model accuracy while drastically compressing the majority portion of the original model size with 8X reduction.

Table 2: DLRM Embedding Tables Quantization, accuracies evaluated on the Kaggle Dataset

Quantization	Testing	
Bit Width	Accuracy	ROC AUC
Full-precision	78.923%	0.8047
INT16	78.928% (+0.005%)	0.8046 (-0.0001)
INT8	78.985% (+0.062%)	0.8054 (+0.0007)
INT4	**79.070% (+0.147%)**	**0.8075 (+0.0028)**

3.2 Quantization of the Whole Model

DQRM is based on the well-known DLRM recommender system which contains two major components in its architecture: Embedding Tables and Multi-Layer Perceptrons (MLP). Compared with quantizing only the embedding tables, we observe that MLP layers are much more sensitive to quantization, and implementing this naively can lead to more accuracy loss, which has been similarly observed in the prior work [3]. We found that using channel-wise quantization of MLP layers performs better, as it leads to a lower accuracy drop than the full matrix-wise quantization. We evaluate MLP quantization in depth in Section 4.

To better understand the effects of quantization, we evaluate quantizing the whole DLRM model (Embeddings+MLP) into uniform INT4 and tested the performance on both the Kaggle and Terabyte datasets. Following the original DLRM work [17] the specific reference model architectures for the two datasets are different and are summarized in Table 1. The Terabyte reference model is much larger compared to the Kaggle reference model. We used the same experiment platforms for the Kaggle model as in Section 3.1. However, running the Terabyte model is much more resource-demanding. We deploy the models on the Intel Academic Compute Environment (ACE) CPU clusters and specifically use Intel(R) Xeon(R) Platinum 8280 CPUs for model training and inference. The quantized models are all trained in DQRM for five epochs.

We observe similar overfitting issues when training for more than one epoch on the Terabyte dataset, and this concurs with the same overfitting seen on the smaller Kaggle dataset. We record the training loss and the testing accuracy curves in Figure 4 (b), (c), and (d). In (b), we contrast the testing accuracies of both original model training and DQRM 4-bit. Again, the black vertical dashed lines signify the boundaries of each epoch. At the end of the second epoch, the testing accuracy curve (blue) for the unquantized model drops significantly, while the 4-bit QAT testing accuracy curve (orange) rises steadily and reaches comparable testing accuracies at the end of the fifth epoch. The testing accuracies and the training loss curves during the five epochs are further contrasted for both training scenarios in (c) and (d), unquantized training in (c), and 4-bit QAT in (d). In (c), the training loss curve (orange) for the unquantized model drops drastically at the end of the second epoch, the same time when the model testing accuracy curve (blue) crashes. In contrast, the QAT training loss curve (orange) in (d) decreases steadily throughout the training, while the testing accuracy curve (blue) continues to rise and plateau. In contrast, the 4-bit QAT is

Table 3: DQRM 4-bit quantization results evaluated on Kaggle and Criteo datasets

(a) **4-bit quantization results for DLRM on Kaggle**

Quantization Settings	Model Bit Width	Model Size	Training loss	Testing Accuracy	ROC AUC
Baseline	FP32	2.161 GB	0.303685	78.923%	0.8047
Vanilla PTQ	INT4	0.270 GB	-	76.571%	0.7675
DQRM	INT4	0.270 GB	0.436685	79.071%	0.8073

(b) **4-bit quantization results for DLRM on Criteo**

Quantization Settings	Model Bit Width	Model Size	Training loss	Testing Accuracy	ROC AUC
Baseline	FP32	12.575 GB	0.347071	81.165%	0.8004
Vanilla PTQ	INT4	1.572 GB	-	78.681%	0.7283
DQRM	INT4	1.572 GB	0.409774	81.210%	0.8015

effective in regularizing and preventing the severe overfitting issue that normal DLRM model training suffers and is able to utilize the training dataset more effectively.

We report training and testing performance in Table 3. We are aware of the previous Post-Training Quantization (PTQ) efforts [3] for DLRM models. However, previous works neither evaluate the quantized model on the publicly available datasets (Kaggle or Terabyte datasets), nor do they publish their implementations. To compare, we showed the testing performance of the vanilla PTQ on quantizing DLRM models in uniform 4-bit. Following the DQRM 4-bit quantization, the vanilla PTQ 4-bit is achieved to have embedding tables quantized channel-wise and MLP quantized row-wise. Again, experiments are all evaluated on the single-machine settings. Table 3 (a) shows the experimental results on the Kaggle dataset. DQRM is able to get a substantial boost in testing performance while using a significantly smaller bit width INT4 than the unquantized model. We follow the official DLRM training recipe, specifically batch size 128 for Kaggle and 2048 for Terabyte. Using an update period of 200, DQRM achieves both a higher testing accuracy (79.071%) and a higher testing ROC AUC score (0.8073) than the unquantized baseline DLRM. The vanilla PTQ incurs a significant accuracy drop in both testing accuracy (> 2%) and testing ROC AUC score (> 0.03). Table 3 (b) presents the experimental results on the Terabyte dataset. By using an update period of 1000, DQRM INT4 achieves slightly higher than FP32 baseline test accuracy (0.45%) and testing ROC AUC (0.0011). Similarly, vanilla PTQ in INT4 leads to a significant testing accuracy drop (> 2%), which is suboptimal for quantizing the model.

3.3 Gradient Compression

In order to speed up the DQRM training further, communication message reduction is important, and we strive to add gradient quantization on top of the PyTorch-supported heavy sparsification for embedding tables to achieve an additional compression ratio.

In this work specifically, we evaluated the DQRM framework under the synchronous data parallelism settings, where the gradient on all machines allreduced every iteration. We found that the PyTorch `DistributedDataParallel` library doesn't support gradient quantization. In order to experiment on a real distributed environment, we customized our own Data Parallel library on PyTorch, which implements gradient quantization locally before the `allreduce` operation. More details of our implementation can be found in Appendix A.1, and we also open-sourced our implementation on the project's anonymous GitHub repository. Our implementation supports DP on both multi-GPU and CPU node platform. All gradient compression experiment results are presented in Table 4.

For experiments on the Kaggle dataset, we utilized four NVIDIA A5000 GPUs. Naively, because of the giant model size, when the gradient is completely uncompressed, the overhead of gradient communication is up to 2.1 GB per iteration. Among the overhead, over 99% is from the embedding tables. However, since most rows in the embedding tables are unused in every iteration, the gradients for the unused rows are guaranteed to be zero. PyTorch has built-in support to exploit specified sparsity in the EmbeddingBag modules which can compress 2.1 GB to 2 MB. Also, using specified sparsity is lossless and has no impact on model training performance.

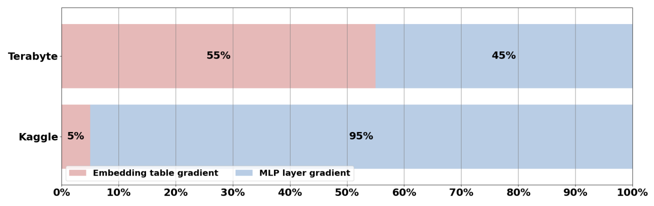


Figure 5: A breakdown of the composition of gradient communication

As shown in Section 2.4, heavy embedding gradient sparsification is not enough alone, as the communication during training continues to bottleneck the training time. On top of the heavily specified sparsity, we add INT8 fixed-point quantization on the gradient to further compress communication size. Interestingly, among the composition of the gradient after sparsification, the MLP gradient occupies a significant portion across both settings. We further add error compensation to the MLP layers only to reduce accuracy degradation. Unfortunately, PyTorch doesn't support low-precision sparse allreduce well. With limited backend support and significant communication overhead caused by sparse tensor coalescing, achieving speedup from communication reduction remains difficult. Currently, local gradient quantization operation per iteration in our system still adds additional overhead. We want to note that with in-depth system-level optimization for `allreduce` and gradient quantization workload, the training time cost of DQRM can be further reduced, but such an endeavor is outside the scope of our current work. Nevertheless, we showed that adding gradient quantization only introduces a trivial decrease 0.057% in the testing accuracy and 0.0012 in the ROC AUC score despite a significant reduction in communication size.

For experiments on the Terabyte dataset, we ran on two Intel(R) Xeon(R) Platinum 8280 CPU nodes. Similarly, gradient compression

Table 4: Communications compression for Distributed Data Parallelism training among four nodes or GPUs

Model Settings	Training Platforms	Communication Compression settings	Communication Overhead per iter	Latency per iter	Training Loss	Testing Accuracy	ROC AUC
Kaggle	4X Nvidia A5000 GPUs	gradient uncompressed (DQRM 4-bit)	2.161 GB	>1000 ms	0.436685	78.897%	0.8035
		+ EMB gradient sparsification (specified)[2]	2.010 MB	61 ms	0.436685	78.897%	0.8035
		+ INT8 gradient Quantization	0.509 MB	110 ms[3]	0.442300	78.840%	0.8023
Terabyte	2X (2 processes) Intel(R) Xeon(R) Platinum 8280 CPU	gradient uncompressed (DQRM 4-bit)	12.575 GB	>1000 ms	0.412688	81.156%	0.7997
		+ EMB gradient sparsification (specified)	6.756 MB	210 ms	0.412688	81.156%	0.7997
		+ INT8 gradient Quantization	1.732 MB	225 ms	0.414731	81.035%	0.7960

of the embedding tables introduces insignificant accuracy loss with roughly 0.1% for testing accuracy and less than 0.004 for testing ROC AUC. We supplement with more evaluations of MLP gradient sensitivity for quantization in the ablation studies.

4 ABLATION STUDIES

Table 5: Evaluation of Periodic Update on Kaggle and Terabyte Datasets

Model Settings	Period	Latency per iter	Testing Accuracy	ROC AUC
Kaggle	1	31 ms	79.040%	0.8064
	200	22 ms	79.071%	0.8073
	500	22 ms	79.034%	0.8067
Terabyte	1	>1200 ms	-	-
	200	58 ms	81.159%	0.7998
	500	51 ms	81.193%	0.8009
	1000	46 ms	81.210%	0.8015

4.1 Periodic update

As shown in the methodology section, the sheer size of embedding tables renders table traversal a bottleneck during the forward pass. We propose to utilize periodic updates of the quantization scale of each embedding table during quantization in order to amortize the huge latency of table lookup into multiple iterations. Among the options, we found that the optimal period is different for training settings with different model sizes and batch sizes. The experimental results are presented in Table 5.

For experiments running on the Kaggle dataset, we run on one Nvidia A5000 GPU. We again trained the model for 5 epochs. During DQRM, all weights are quantized into INT4. Training time per iteration decreases significantly, from over 31 ms to 22 ms when the quantization scale is updated once every 200 iterations compared with every iteration. Also, along with decreased single-node training time, model convergence even slightly improved. We also test the update period of 500, and we found diminishing gains in training time reduction and a slight decrease in testing performance.

Similar experiments are conducted on a single Intel Xeon platinum 8280 CPU on Terabyte. We found that with the current ML framework, CPUs have much greater difficulties in traversing large embedding tables. A single iteration of training of the model under the Terabyte setting takes more than 1.2 seconds to finish, thus making it impractical to complete. When the quantization scale is periodically updated once every 200 iterations, the training time per iteration dramatically improves to 57 ms per iteration. Due to the difference in model size and training mini-batch size, we found that using a longer period of 500 iterations brings further speedup in training time per iteration, dropping to 34 ms, while even slightly benefiting the training accuracy (0.03% compared with 200 iterations). We hypothesize that the reason for the slight accuracy boost is that, QAT inherently has an unstably growing quantization scale, and periodic update of the quantization scale helps stabilize training, which further helps the model converge.

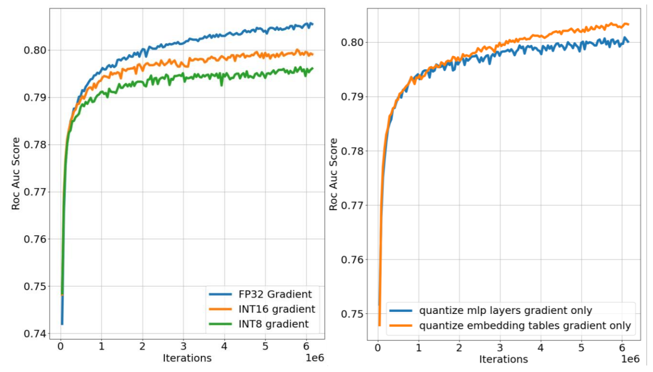


Figure 6: (a) Testing accuracy of naively quantizing gradients into different bit widths. Naive gradient quantization leads to a significant accuracy drop. (b) MLP gradients are more sensitive to quantization. If only quantizing the embedding table gradients into INT16, it will lead to less drop in accuracy compared to only quantizing MLP gradients into INT16.

4.2 Different Gradient Quantization Bit Width

In this Section, we present our findings related to the quantization of the gradients during distributed DP training for DQRM. All experiments presented in this Section assume that specified sparsity is enabled. Firstly, we study the effect of naive quantization of

[2]specified sparsification is a lossless compression for embedding tables so the testing accuracy is exactly the same as uncompressed case
[3]PyTorch sparse tensor allreduce library doesn't support low-precision arithmetic, without further system level effort in low-precision optimization, the latency per iteration increases purely from the quantization overhead per iteration

gradients on the model convergence. The experiments are run on a single NVIDIA M40 GPU under a simulated DP environment under Kaggle dataset. More details of the experiment platform setup can be found in Appendix A.2. Quantizing all the gradients naively introduces huge accuracy loss and largely hurts the model convergence. In Figure 6 (a), blue curve signifies the baseline: FP32 gradients. The orange curve is for INT16 gradients, while the green curve is for INT8 quantization gradients. Both the INT16 and the INT8 quantization force the training to plateau earlier and converge to a much lower testing accuracy.

Following our first experiment and setup, we investigate quantizing just embedding table gradients or just MLP layers gradient but not both Figure 6 (b). We show that only quantizing embedding table gradients allows the model to continuously converge throughout the five epochs while quantizing the MLP layers significantly slows down the model convergence. Therefore, we decided to add Error Compensation only to MLP layers' gradients, as MLP layers' gradients are more sensitive to quantization. More experiments are presented in Table 6 on the similar 4 Nvidia A5000 GPUs with our custom framework. We showed that quantizing Embedding Table gradient into INT8 doesn't hurt test accuracy much, and MLP error compensation largely prevents the large accuracy degradation from MLP gradient quantization.

Table 6: Evaluation of MLP Quantization and Error Compensation

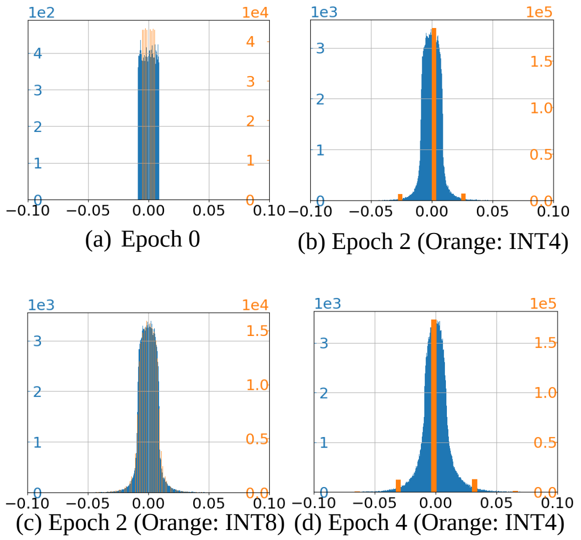
Settings	Testing Accuracy	Testing ROC AUC
baseline	78.897%	0.8035
Only Embedding gradient in INT8	78.858%	0.8023
Embedding and MLP gradient in INT8	78.608%	0.7974
Embedding and MLP gradient in INT8 + MLP EC	78.840%	0.8023

4.3 Quantization of Different Part of Models

Table 7: Quantization Evaluation of Each Part of the Model

Settings	Testing Accuracy	Testing ROC AUC
Baseline	78.718%	0.8001
+ Embedding Tables in 4-bit	78.936%	0.8040
+ MLP in 4-bit matrix-wise	78.830%	0.8022
+ MLP in 4-bit channelwise	78.897%	0.8035
+ MLP in 8-bit channelwise	78.950%	0.8045

In DQRM training, we observe that different layers exhibit different sensitivities to quantization. To study this, we utilize four NVIDIA M40 GPUs and conduct experiments on the Kaggle dataset under the distributed DP settings. The experiment results are presented in Table 4.3. Compared with quantizing embedding tables, MLP layers quantization cannot benefit from the diminishing overfitting effect. Instead, when quantizing MLP layers in the matrix-wise fashion into INT4 and quantizing activation during QAT, the DQRM framework fails to converge. When just quantizing the MLP layers in the matrix-wise fashion and without quantizing the activations, DQRM gives more than 0.1% testing accuracy drop, which is a lot for the Kaggle dataset. In contrast, we analyze MLP quantization in the channel-wise fashion, specifically finding a scale for every row inside the linear layer weights. MLP channel-wise quantization outperforms matrix-wise, bringing the overall accuracy loss for the entire INT4 model to be under 0.04% and 0.0005 in testing ROC AUC compared to single precision unquantized MLP model. We also provide the performance of quantizing the MLP layers in INT8 channel-wise, it is on par with the single precision MLP in test accuracy and test ROC AUC score.



(a) Epoch 0 (b) Epoch 2 (Orange: INT4)

(c) Epoch 2 (Orange: INT8) (d) Epoch 4 (Orange: INT4)

Figure 7: Table 6 in DQRM Kaggle model weight distribution shift. Orange is for the quantized weight distribution under QAT, while blue is for the unquantized weight under normal training. (a) Uniform initialization of embedding tables before training starts. (b) The contrast between unquantized weight and INT4 quantized weight distribution after roughly 2 epochs. (c) Meanwhile, if model is running QAT in INT8 would have different weight distribution (orange) compared with weight during normal training (blue). (d) At the end of 5 epochs of training, DQRM INT4 weight distribution.

4.4 How Quantization Affects Weight Distribution

Here, we strive to provide more intuition of why INT4 quantization can provide stronger resilience towards overfitting. We look into the weight distribution shift of normal unquantized training, the QAT in INT8 training, and the DQRM in INT4. The weight distribution is plotted in Figure 7. The blue histogram displays the distribution of the unquantized weights, while the quantized weights are in orange. More histograms can be found in the Appendix.

The general trend of weight distribution shift during normal DLRM training is to gradually shift outward, which can be noted by the contrast of the blue histogram between (b) the weight distribution after the second epoch and (d) after 5 epochs. We observed that the quantized weights (orange) exhibit a similar outward-shifting trend. DQRM INT4 quantized weights (orange) in (b) and (d) seemingly trace the unquantized weight poorly compared with INT8 weights in (c), having the majority of the weight distributed near the origin while having some portion of the weights located symmetrically around 0. In comparison, (c) shows INT8 can roughly cover the entire weight distribution neatly. We also observe that the FP32 copy of weights under the quantization-aware training regardless of bitwidth are highly similar in distribution compared with normal training. As shown in section 3.1, INT8 although converges slightly higher than the unquantized model, still follows a similar overfitting trend. We hypothesized that the INT4 quantization resilience towards overfitting comes from its large alienation to normal embedding table weight distribution shift. Also, we observe that the INT4 model has the majority of weights close to zero while performing well, which provokes the thought of whether high sparsity in large embedding tables exists and can be better utilized to tackle overfitting. We leave the question to later works to explore.

5 PREVIOUS WORKS

Neural Network Quantization - Neural network quantization has been widely explored in almost all types of neural networks. We point to one survey paper [5] for a comprehensive summary of quantization development and techniques. Quantization-aware Training (QAT) has been widely deployed in CNNs [2, 10–12, 18, 22] and transformers [1, 4, 14, 20, 25]. Compared with Post-Training Quantization (PTQ), QAT benefits from an additional round of fine-tuning using low precision during the forward pass, achieving lower accuracy degradation for various model architectures. However, due to the extra training process, QAT usually costs much longer time compared to PTQ. Due to the presence of large embedding tables, this problem exacerbates in recommendation models where conventional QAT methods worsen the already severe memory overhead during training. On the other hand, for other types of commonly used recommendation models such as In this work, we propose techniques to alleviate the issue by optimizing the forward pass of QAT tailored for DLRM-like models.

Compressing Large-scale Recommendation Models - DLRM [17] is the state-of-the-art click-through-rate recommendation model designed and deployed by Meta. However, under the MLPerf settings, the model trained on the Terabyte dataset already contains Embedding tables that require storage of more than 89 GB, which exceeds the memory capacity of every current generation GPU. The giant model size has brought huge difficulties in both model training and inference, which motivates works that compress the model. Over 99% of the DLRM model size is occupied by the embedding tables, and various previous works attempt to shrink the embedding tables. [6] proposes to shrink the embedding dimension of each embedding table based on their hotness evaluated through a metric based on spectral decay and the popularity of each table. Similarly, [21] proposed to break large embedding tables into smaller sub-tables and combine sub-table vectors using compositional operations. Note that both works essentially prune the embedding table sizes and leave the precision of the model unchanged, which are actually orthogonal to our work. In addition, [13] uses differentiable product quantization for matrix factorization and neural collaborative filtering models, and [15] shows that half-precision quantized model weights benefit the training speed and do not hurt convergence for matrix factorization models. Besides, [7] uses codebook quantization to quantize the embedding tables of DLRM models into INT4 under the PTQ settings, while [3] further quantizes the whole model into 4-bit. Both works revealed that PTQ introduces accuracy degradation, which makes QAT crucial. There also has been recent work [16] utilizing Low-Precision Training (LPT) on other large recommendation models such as DCN ([23] to achieve a low-precision model under low training memory cost. By doing so, they sacrifice some accuracy. They explore a different trade-off than ours. They trade model accuracy for low training memory, We aim for ultra-low precision during inference that has the same accuracy as the unquantized model, while compensating training by making QAT more efficient for recommendation systems.

Efficient Training of Recommendation Models - On the Matrix Factorization models side, [15] targets model training and proposes to quantize the gradients with dynamic shifting of gradient precision to preserve accuracy. On the other hand, for large-scale recommendation models such as DLRM, the models are usually trained using multiple GPUs or CPU nodes, and the training time is heavily affected by the communication strategies between nodes. [9] improved upon conventional Top-k sparsification and applies heavy sparsification to the communication of `all-to-all` and `allreduce` under both Model Parallelism (MP) and Data Parallelism (DP) settings. [24] impose quantization on communication messages under both MP and DP settings and show that with heavy error compensation, the communication workload can be significantly reduced without significant error degradation. On CNN and ASR models, previous work [19] looked at combining sparsification and quantization on communication messages together. However, it shows that the merit of combining the two techniques varies greatly between different layer architectures, making their technique not generalizable to all different model architectures. In this work, we looked into these techniques' effects for different parts of DQRM under the DP environment and combine specified sparsification and quantization together to further benefit DQRM training.

6 CONCLUSION

In this work, we propose a systematic quantization framework DQRM for large-scale recommendation models. Specifically, we discover that the DLRM model suffers severely from the overfitting problem. We show that ultra-low precision quantization can help overcome the strong overfitting and better utilize the training dataset, which eventually leads to higher test accuracy. We observe that conventional QAT is troublesome in training large-scale recommendation models and we propose two techniques that significantly alleviate the issue. Besides, to further optimize DQRM under the distributed environment, we combine specified sparsification and quantization together to compress communications. Our framework is intensively evaluated on the published dataset Kaggle and Terabyte, where we outperform the full-precision DLRM baselines while achieving an 8 $\times$ reduction of model size.

REFERENCES

[1] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. 2019. Efficient 8-Bit Quantization of Transformer Neural Machine Language Translation Model. https://doi.org/10.48550/ARXIV.1906.00532

[2] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems* 28 (2015).

[3] Zhaoxia Deng, Jongsoo Park, Ping Tak Peter Tang, Haixin Liu, Jie Yang, Hector Yuen, Jianyu Huang, Daya Khudia, Xiaohan Wei, Ellie Wen, et al. 2021. Low-precision hardware architectures meet recommendation model inference at scale. *IEEE Micro* 41, 5 (2021), 93–100.

[4] Angela Fan, Edouard Grave, and Armand Joulin. 2019. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556* (2019).

[5] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. https://doi.org/10.48550/ARXIV.2103.13630

[6] Antonio A Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. 2021. Mixed dimension embeddings with application to memory-efficient recommendation systems. In *2021 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2786–2791.

[7] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. 2019. Post-training 4-bit quantization on embedding tables. *arXiv preprint arXiv:1911.02079* (2019).

[8] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2019. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. *CoRR* abs/1906.03109 (2019). https://arxiv.org/abs/1906.03109

[9] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W Mahoney. 2021. Training recommender systems at scale: Communication-efficient model and data parallelism. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2928–2936.

[10] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168* (2016).

[11] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. 2018. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems* 29, 11 (2018), 5784–5789.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. *Advances in neural information processing systems* 29 (2016).

[13] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H Chi. 2020. Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems. In *Companion Proceedings of the Web Conference 2020*. 562–566.

[14] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. I-bert: Integer-only bert quantization. In *International conference on machine learning*. PMLR, 5506–5518.

[15] Yunyong Ko, Jae-Seo Yu, Hong-Kyun Bae, Yongjun Park, Dongwon Lee, and Sang-Wook Kim. 2021. MASCOT: A Quantization Framework for Efficient Matrix Factorization in Recommender Systems. In *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 290–299.

[16] Shiwei Li, Huifeng Guo, Lu Hou, Wei Zhang, Xing Tang, Ruiming Tang, Rui Zhang, and Ruixuan Li. 2022. Adaptive Low-Precision Training for Embeddings in Click-Through Rate Prediction. *arXiv preprint arXiv:2212.05735* (2022).

[17] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[18] Renkun Ni, Hong-min Chu, Oscar Castañeda, Ping-yeh Chiang, Christoph Studer, and Tom Goldstein. 2020. WrapNet: Neural Net Inference with Ultra-Low-Resolution Arithmetic. https://doi.org/10.48550/ARXIV.2007.13242

[19] Cedric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2018. SparCML: High-Performance Sparse Communication for Machine Learning. https://doi.org/10.48550/ARXIV.1802.08021

[20] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. (2019). https://doi.org/10.48550/ARXIV.1909.05840

[21] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 165–175.

[22] Shyam A. Tailor, Javier Fernandez-Marques, and Nicholas D. Lane. 2020. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. https://doi.org/10.48550/ARXIV.2008.05000

[23] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.

[24] Jie Amy Yang, Jongsoo Park, Srinivas Sridharan, and Ping Tak Peter Tang. 2020. Training deep learning recommendation model with quantized collective communications. In *Conference on Knowledge Discovery and Data Mining (KDD)*.

[25] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/micro50266.2020.00071

[26] Jieming Zhu, Jinyang Liu, Shuai Yang, Qi Zhang, and Xiuqiang He. 2020. Fuxictr: An open benchmark for click-through rate prediction. *arXiv preprint arXiv:2009.05794* (2020).

A EXPERIMENT PLATFORMS

A.1 Distributed Environment with Gradient Quantization

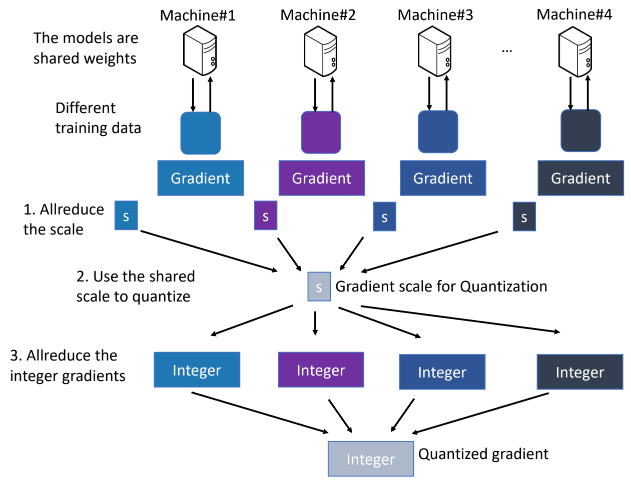


Figure 8: Illustration of the Customized Data Parallelism Framework that Supports Gradient Quantization

In PyTorch DistributedDataParallelism library, during backward propagation, the error tensor's backward() function handles both backpropagating gradients to local parameters together with the all reduce of gradient tensor across nodes. Therefore, after the powerful single line of code, gradients behind each parameter tensor in the distributed data parallelism model have its corresponding gradient ready. However, the quantization of the gradient has to happen between the local backpropagation and the global allreduce. Therefore, distributed data parallelism package is not very handy.

In order to implement gradient quantization, we implement a stand-alone framework that performs the majority of distributed data parallelism workflow. To implement gradient quantization, we summarized our implementation in Figure 8. We start by partitioning the training batch into pieces to be used by different machines. After forward propagation, the local devices are asked to backpropagate the corresponding gradient locally. Then, for all parameters in the model, the following sequence of steps is performed. The quantization scale of parameter tensors is computed locally at first. Computing the quantization scale is completely determined by the quantization bitwidth and local tensor distribution so can be performed locally. Then, the quantization scale is first allreduced to be unified. Then, using the same scale, gradient tensors on each device are quantized locally. The integer gradients are then allreduced. After two all reduce steps, each device then has the same quantization scale and the same quantized gradients. The local step update of parameters then can be dequantized. The quantization bit width used is INT8 and unified across all parameters. When it comes to sparse gradients from embedding tables, we only quantize the `values` part of the `sparse_coo` tensor format and leave the `indices` and other fields unchanged.

One caveat of the entire flow of our customized gradient communication is that it can only be used for allreduce paradigms that perform the reduce operation at the destined machine, such as Ring allreduce. However, for recursive doubling allreduce, where the reduce operation is collectively performed by many machines, our framework is not applicable. The main problem is at each reduce operation, where a sum is taken by two low-precision operations. To continue propagating the gradients under the low-precision bitwidth, an additional rounding should occur, which may potentially lose the gradient precision further. Unfortunately, PyTorch allreduce operations use recursive-doubling allreduce, thoughout our presented experiment, we use FP32 tensors to contain quantized INT8 integer gradients to avoid accumulation and rounding issues from the impact of allreduce.

A.2 Simulated Data Parallelism Environment

Algorithm 1: Simulated Data Parallelism on single machine

```
buffer_clean ← False;
for j, batch in enumerate(train_loader) do
    Z ← model_forward(training_input);
    E ← loss_criterion(Z, label);
    clear_gradient(model);
    if buffer_clean then
        grad_buffer_zero();
        buffer_clean ← False;
    end
    E.backward() ;  /* local backward propagation */
    grad_buffer_update();
    if j % simulated_nodes == 0 then
        weight_buffer_update();
        buffer_clean ← True;
    end
end
```

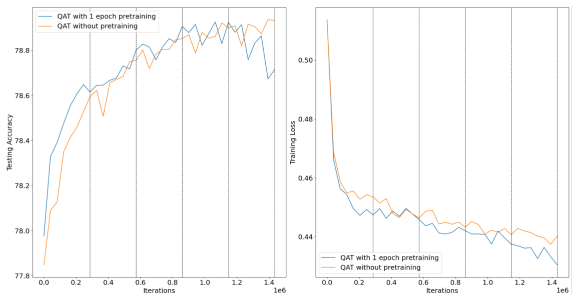
For some experiments presented in the main content, we use a simulated environment to run Data Parallelism on single device to imitate the multiple device environment. Here we provide more details of our implementation. In our implementation, we add an addition gradient buffer for every parameter tensor. As summarized in 1, the batch size is splited similar to one distributed machine. In the second if statement, the weight is actually updated by the gradient every simulated_node iterations. Between updates, training still backpropagate gradients from relevant iteration and are hold inside the parameter buffers. After weight being updated by the buffer, the buffer is cleared prior to new gradient being backpropagated. One key difference between simulated and the real distributed environment lays in its allreduce mechanism. Shown in section A.1, if gradients are quantized, two allreduce processes occur in one iteration: the first communicates quantization scale, while the second one communicates quantized gradient tensors. However, two allreduce processes are difficult to implement in the simulated environment. Therefore, we reuse the quantization scale of the first iteration throughout simulated_machine number of iterations in our implementation, which can potentially hurts the gradient quantization precision. However, through our experiments, we didn't observe this case.

A.3 Simulated framework evaluation

Table 8: Multi-node Experiment Results with 8-bit gradients, loss evaluated on the Terabyte Dataset

#Node	Training Loss Drop	Testing Acc Drop	Testing AUC Drop
2	**-0.002447**	0.092	**0.0025**
4	-0.00273	0.114	0.0036
8	-0.00395	**0.059**	0.0053

We also evaluate the effect of different node counts on gradient quantization. The result is listed in Table 8. Currently, different node counts are simulated on the single CPU node. Across three different node counts, 2, 4, and 8, the drop in training loss, testing accuracy, and ROC AUC score are consistent and small.



(a) (b)

Figure 9: (a) Testing accuracy over the 5 epochs for 1 epoch of pretraining before 4 epochs of QAT and 5 epochs of QAT without pretraining. Pretraining leads to faster overfitting, and QAT without pretraining avoids overfitting and achieves better testing accuracy from 5 epochs of training. (b) Training loss over the 5 epochs. Pretraining before QAT leads to a faster decrease in training loss in DLRM compared with QAT without pretraining.

B QAT WITHOUT PRE-TRAINING

Table 9: Evaluation of QAT from scratch on Recommendation Models

Settings	Testing Accuracy	Testing ROC AUC
One epoch pretraining + Four epochs of QAT	78.926%	0.8039
Five epochs of QAT without pretraining	78.936%	0.8040

Previously, QAT has always been used as a fine-tuning technique for training CNN and transformer models. Usually, with pretraining on the single-precision bit-width, the trained weights can reduce accuracy loss when fine tuning on low-precision settings. We examine such paradigm on the DLRM models. We train DLRM in QAT but only quantize the embedding tables into INT4 under the Kaggle dataset using four NVIDIA M40 GPUs under the distributed DP settings. We compare DLRM with one epoch of pretraining in the single precision followed by four epochs of INT4 QAT with DLRM with INT4 QAT from scratch.

We plot the testing accuracy versus iteration curves in Figure 6 (a). The experiment results are presented in Table B. In the diagram, the vertical dashed lines signify the boundary of each epoch. The blue curve is for QAT with pretraining, while the orange curve is without. After the first epoch, we can see that as the model transitions from single-precision to INT4 quantized data type, the blue curve drops slightly, which is expected. Further, in the mid third epoch, QAT with pretraining (blue) reaches its peak and then drops quickly afterward. QAT from scratch eventually has a slight testing accuracy edge over QAT with pretraining. Also, in Figure 6 (b), we plot the training loss across five epochs for two different settings. QAT with pretraining (blue) has training loss constantly below QAT from scratch. From here, we argue that QAT with pretraining on DLRM models does speed up the model convergence, taking less number of epochs to reach its peak. However, it suffers from earlier overfitting compared with QAT from scratch. Under our settings, we observe that QAT from scratch slightly gains in testing accuracy of 0.01% with 0.0001 of testing ROC AUC score.

C WEIGHT DISTRIBUTION SHIFT

We provide weight distribution shift with more granularity. Since the experiments performed are separated but of identical experimental settings from the one in section 4.4. The distribution presented in this section will be highly similar, but there might be some minor difference in details.

We select table 6 because table 6 is a medium size embedding table. We found that large tables usually have majority of weights distributed around 0. However, we found that the medium size embedding tables have similar trend with large embedding tables. The model gradually shifted outwards from uniform distribution probability mass function into a bell-shaped curve. Large embedding table has trend similar to that, but still after a few epochs, the majority of weight still gather around 0. In comparison, mid-sized embedding tables display a more obvious weight distribution shift, so we select table 6, a mid-sized embedding table among 26 tables.

Figure 10 displays the weight distribution shift of unquantized weights across 5 epochs. Weight gradually move from a uniform initialization into a bell-shaped curve. Figure 11 contrasts quantized weights (orange, foreground) and unquantized weights (blue background) distribution. INT8 traces the unquantized weight distribution much closer than INT4 quantization.

D DEMO RECOMMENDER ON THE PHONE

We include a demo of DQRM exported to an Android phone. From Figure 12, the tested DQRM model size is 405.65 MB. As a reference, the DLRM Kaggle model size is 2.16 GB. The model size is not strictly 8× compression because of the following two reasons: 1) Embedding tables can be quantized into INT4, but the embedding vectors have to be bit-packed together into INT8 format to

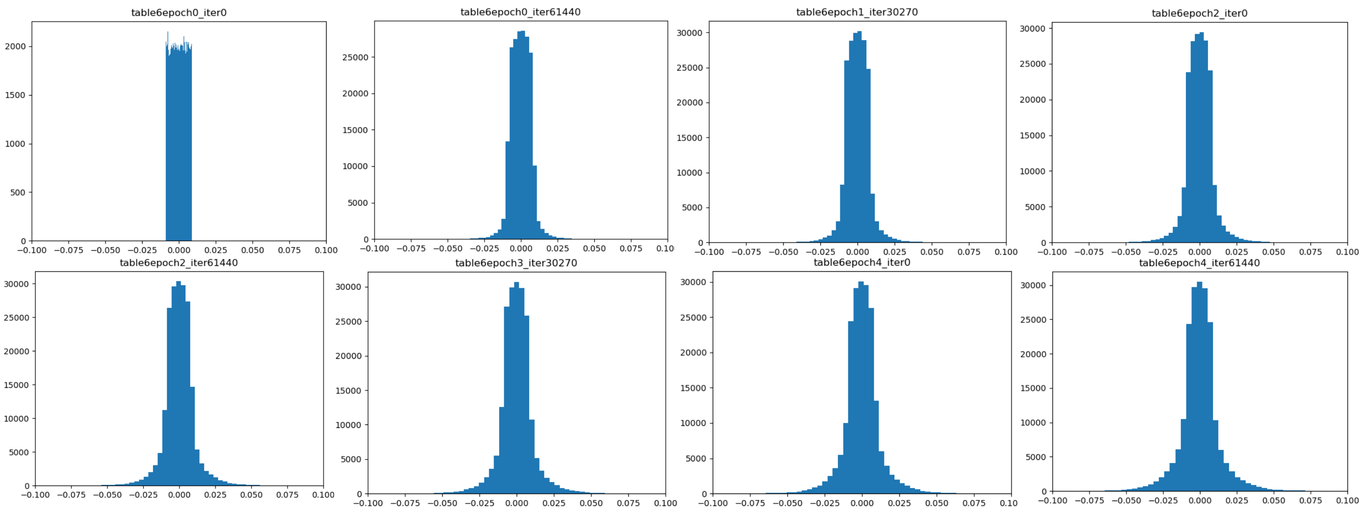


Figure 10: The unquantized weight distribution shift across five epochs. The unquantized model starts overfit after the first epoch

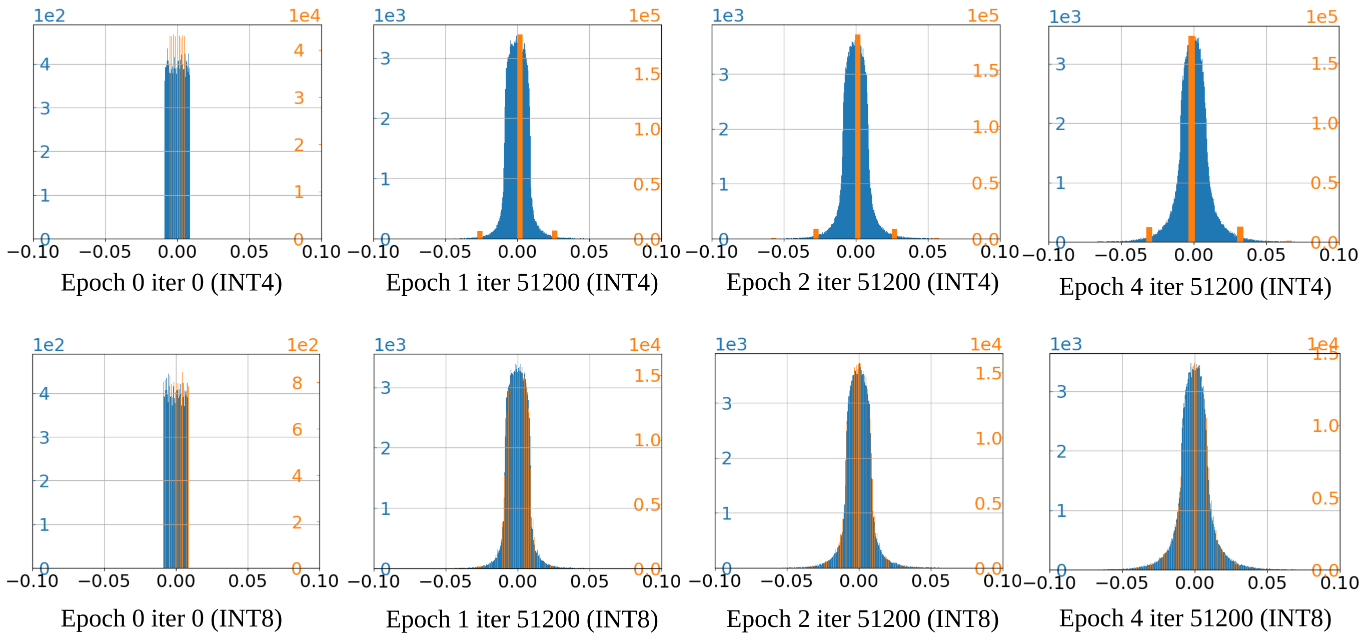


Figure 11: Contrast between INT4 and INT8 quantization-aware training weight distribution shift

fully benefit from the INT4 low precision. However, bitpacking on PyTorch is tricky and PyTorch 4-bit packing is not fully optimized in terms of performance. 2) PyTorch doesn't support MLP layers to be quantized into bitwidth below INT8. Therefore, MLP layers, although quantized into INT4, still need to be stored as INT8 numbers. Still, we believe this is a first step towards deploying large recommendation models to edge devices so as to alleviate the heavy cloud AI inference pressure.

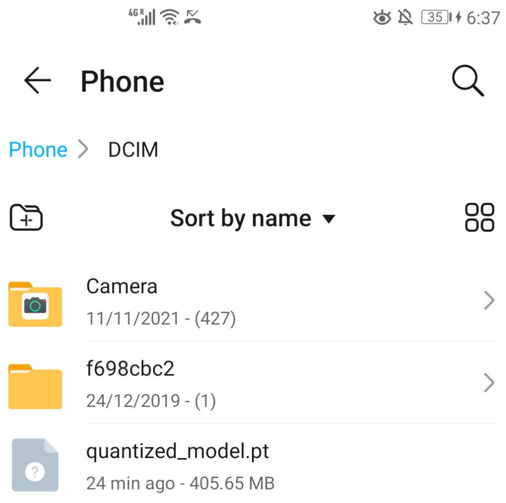


Figure 12: A screenshot of the quantized model on a high-end Android Phone