Supplementary Materials for Deep Quantized Recommendation Models

ANONYMOUS AUTHOR(S)

ACM Reference Format:
Anonymous Author(s). 2023. Supplementary Materials for Deep Quantized Recommendation Models. 1, 1 (May 2023), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ABLATION STUDIES

In the ablation studies, we present extended studies on the parameters of the DQRM framework, including different update period's impact on DQRM in Section 1.1, different gradient quantization bit width's effect on model convergence in Section 1.2, different sensitivity to quantization among layers in DQRM in Section **??**, and the effect of QAT without retraining on DQRM. Then, we evaluate DQRM INT4 with the large number of epochs in training in Section

Table 1. 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

1.1 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 1.

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.

2023. Manuscript submitted to ACM

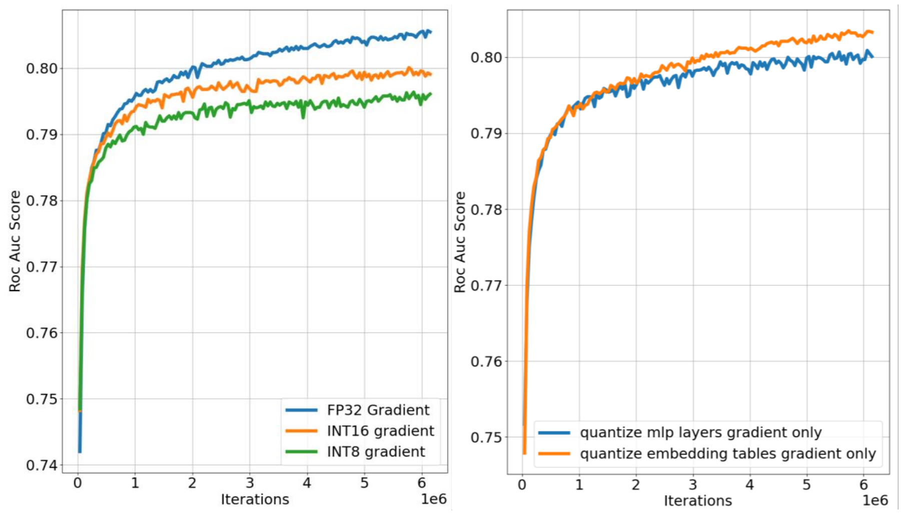

Fig. 1. (a) Testing accuracy of naively quantizing gradients into different bit widths. Naive gradient quantization leads to a significant accuracy drop. (b) MLP gradients are more sensitive to quantization. If only quantizing the embedding table gradients into INT16, it will lead to less drop in accuracy compared to only quantizing MLP gradients into INT16.

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.

1.2 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 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 section 2.3. Quantizing all the gradients naively introduces huge accuracy loss and largely hurts the model convergence. In Figure 1 (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 1 (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 2 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 2. 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

1.3 Quantization of Different Part of Models

Table 3. 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 1.3. 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.

1.4 QAT without Pre-training

Table 4. 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 1 (a). The experiment results are presented in Table 1.4. 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 1 (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.

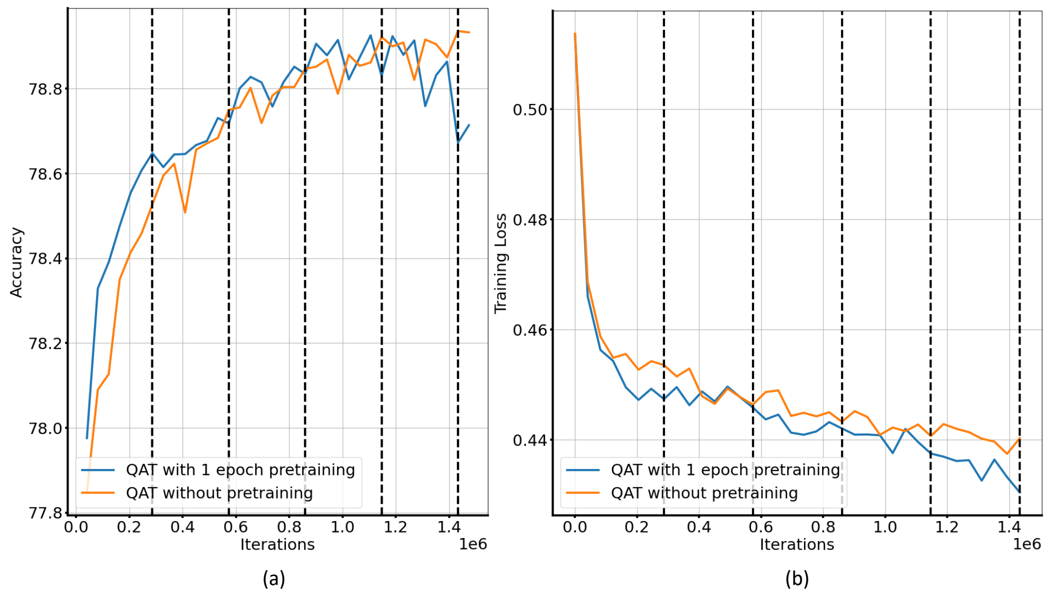

Fig. 2. (a) Testing accuracy over the 5 epochs for 1 epoch of pretraining before 4 epochs of QAT and 5 epochs of QAT without pretraining. Pretraining leads to faster overfitting, and QAT without pretraining avoids overfitting and achieves better testing accuracy from 5 epochs of training. (b) Training loss over the 5 epochs. Pretraining before QAT leads to a faster decrease in training loss in DLRM compared with QAT without pretraining.

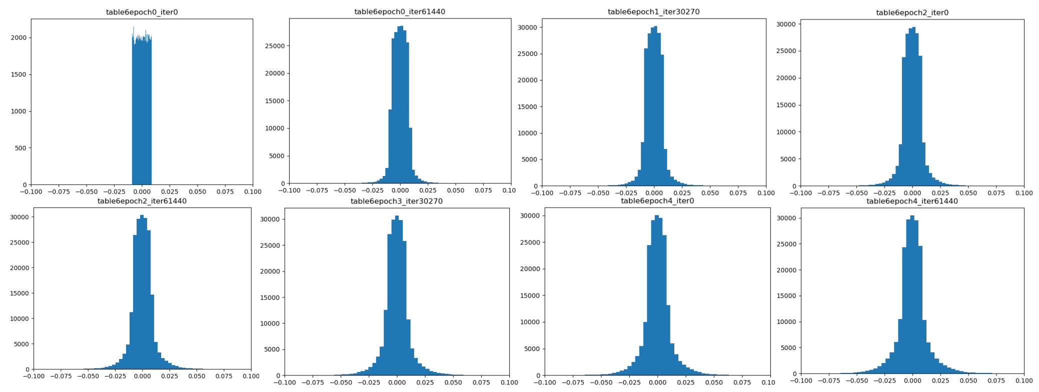

Fig. 3. The unquantized weight distribution shift across five epochs. The unquantized model starts overfit after the first epoch

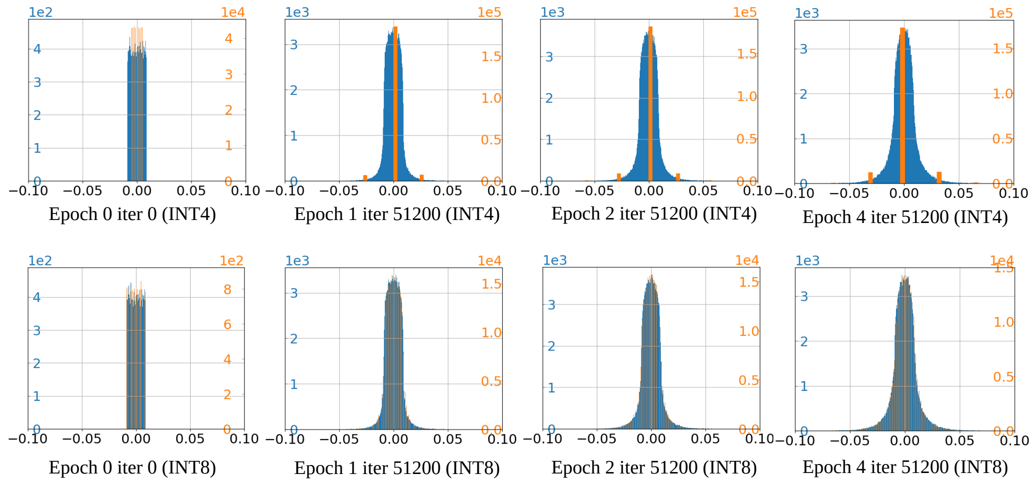

Fig. 4. Contrast between INT4 and INT8 quantization-aware training weight distribution shift. Orange pillars are for the quantized weight distribution under QAT with low-precision, while blue is for the unquantized weight under normal training. Figure **??** follows unquantized weight distribution plots from Figure **??** and plots these blue pillars on the background of each subplot. The orange pillars in the foreground signify the quantized weight distribution shift. The first row is for the distribution shift of the INT4 quantization (with four sampled sections: before training, after the second, the third, and the fifth epoch), while the second row is for the distribution shift of the INT8 quantization.

1.5 How Quantization Affects Weight Distribution

We provide weight distribution shifts with more granularity. Since the experiments performed are separated but of identical experimental settings from the one in section 1.5 in the main body. The distribution presented in this section will be highly similar, but there might be some minor differences in details.

We select table 6 among the 26 tables from the DLRM Kaggle setting because table 6 is a medium size embedding table. We found that large tables usually have the majority of weights distributed around 0. However, we found that the medium size embedding tables have a similar trend with large embedding tables. The model gradually shifted outwards from a uniform distribution probability mass function into a bell-shaped curve. The large embedding table has a trend similar to that, but still, after a few epochs, the majority of weight still gathers 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.

Across the 5 epochs, we sampled four sections in time: before training, at the end of the second, the third, and the fifth epoch. These four sections are aligned on the row. The contrast between rows is for the contrast between different quantization bit-width of the Quantization-aware training. The weight gradually moves from a uniform initialization into a bell-shaped curve. Figure **??** contrasts quantized weights (orange, foreground) and unquantized weights (blue background) distribution. INT8 traces the unquantized weight distribution much closer than INT4 quantization.

Here, we strive to provide more intuition of why INT4 quantization can provide stronger resilience toward 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 **??**. The blue histogram displays the distribution of the unquantized weights, while the quantized weights are in orange. More histograms can be found in the section 4.

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 **??**, 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.

1.6 Training DQRM for Large Number of Epochs

In our main body, we observed that DQRM INT4 prevents overfitting through five epochs of training. For reference, the baseline DLRM in FP32 overfits after one or two epochs. To what extent will DQRM INT4 prevent overfits? Are five epochs enough? To what extent the testing accuracy will plateau? Motivated by these questions, we run DQRM INT4 for an extended 20 epochs till it eventually overfits. We present the testing accuracy curve in Figure 5 (a) contrasting DLRM in FP32 (orange curve) and DQRM INT4 (blue curve). Below we also summarize the specific testing performance in Table 5. Training convergence reaches its peak at the 11th epoch. After then, it plateaus for 2 to 3 epochs and then goes down. Our experiment shows that 5 epochs aren't enough for making DQRM INT4 fully converged. However, still, considering spending twice the amount of time for slight accuracy advantage, we leave our results here for the user to decide which scenario they are willing to go for.

1.7 Prior QAT Techniques on DLRM

In this section, we try to address why different prior QAT methods underachieved in the DLRM model settings. In the main body, we also compare DQRM against HAWQ. Since DQRM without all the tricks added additionally was built on top of the mere quantization scheme of HAWQ. Therefore, HAWQ directly faces the overwhelmingly significant memory

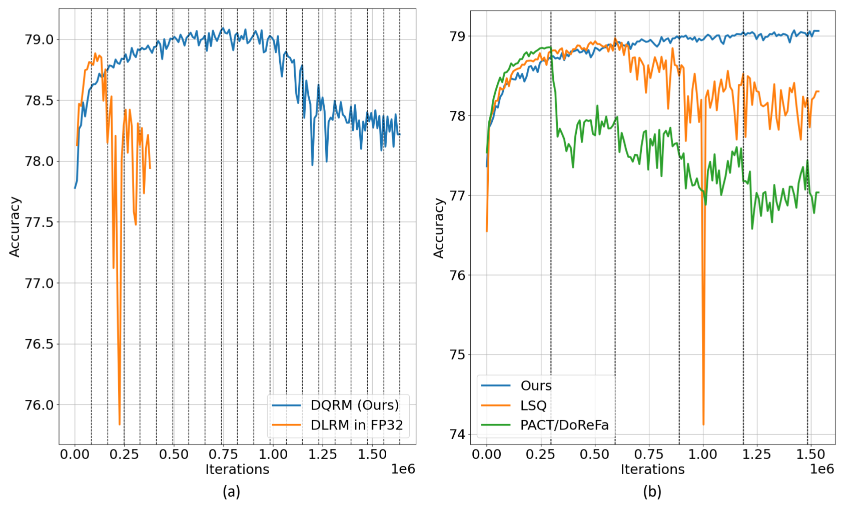

Fig. 5. Figure (a) shows the observations when running DQRM INT (blue) for 20 epochs versus running DLRM in FP32 for 5 epochs; Figure (b) compares DQRM, LSQ, and PACT;

Table 5. Test results by number of epochs

# Epochs	Test Accuracy	Test ROC AUC
1	78.581%	0.7967
5	78.949%	0.8050
10	79.092%	0.8073
Peak (epoch 11)	79.092%	0.8076
15	78.743%	0.8009
20	78.219%	0.7499

usage and long training time for the Terabyte dataset to converge. What we want to discuss here is more focused on other QAT techniques such as PACT and LSQ. Quantization has its key in building a mapping between quantized integer values and their original real values. PACT introduces the learned clipping range during activation quantization, while the weight quantization is completely reliant on DoReFa quantization. DoReFa uses tanh function extensively, but empirically the mapping is not comparable to uniform mapping implemented by the vast majority of quantization works that follow, such as LSQ or HAWQ. Shown in Figure 5 (b), it converges the fastest among all techniques and suffers from overfitting severely. LSQ on the other hand learns its clipping range during weight quantization. However, it is not as competitive as DQRM which uses static min max clipping range on both datasets. A potential explanation is that in section 1.4 or section 4 of the supplemental materials, we present the weight distribution shift of the DLRM model during training. The range of values are spreading out constantly throughout, unfortunately for LSQ, the optimal clipping range might happen to be a moving objective, which hurts its performance. DQRM's static min-max clipping range might actually benefit from its stronger ability to adapt.

2 EXPERIMENT PLATFORMS

2.1 Distributed Environment with Gradient Quantization

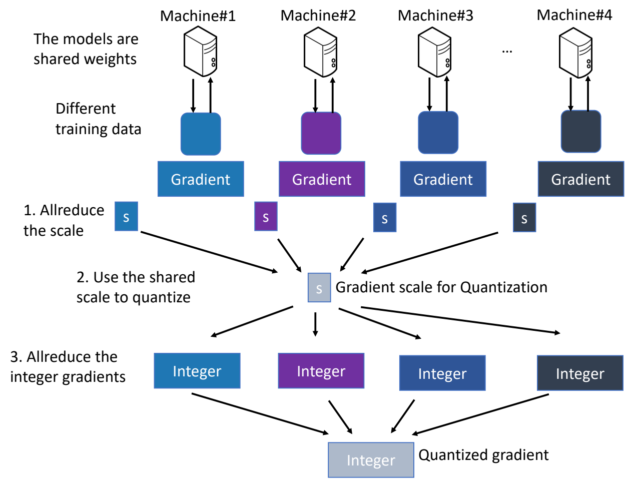
Machine#1
Machine#2
Machine#3
...
Machine#4
The models are shared weights
Different training data
Gradient
1. Allreduce the scale
s
2. Use the shared scale to quantize
Gradient scale for Quantization
3. Allreduce the integer gradients
Integer
Quantized gradient

Fig. 6. 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 6. 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.

2.2 Additional Breakdown on DLRM Distributed Training

Firstly, we provide more details on the experiment setup for Figure 1(c). The experiments are run on both single GPU and single CPU node. We train the model under the Kaggle dataset settings on a single Nvidia A5000 GPU, as shown in the left pillar. We also run experiments of the 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.

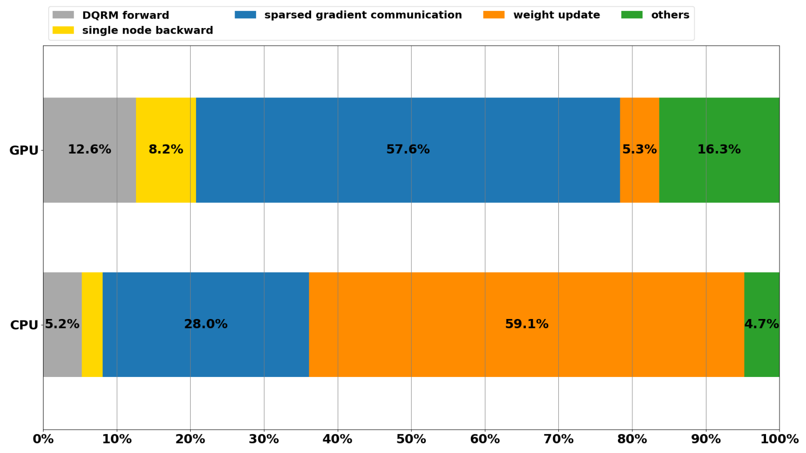

Fig. 7. A breakdown of training time on distributed data parallelism environment on multiple GPUs and CPU nodes

In Figure 7, 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.

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
```

2.3 Simulated Data Parallelism Environment

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 2.1, 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.

2.4 Simulated framework evaluation

Table 6. Multi-node Experiment Results with 8-bit gradients, loss evaluated on the Terabyte Dataset

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

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

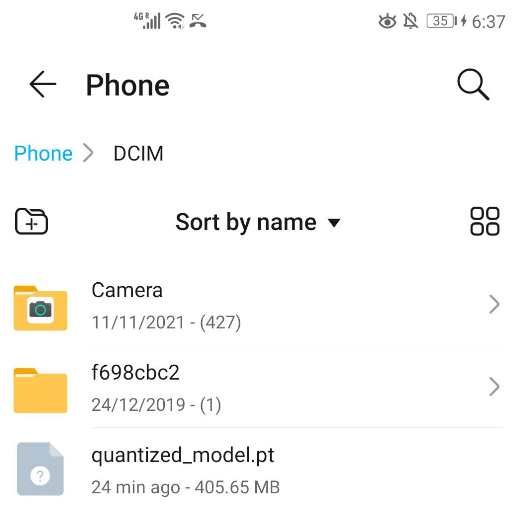

Fig. 8. A screenshot of the quantized model on a high-end Android Phone

We also evaluate the effect of different node counts on gradient quantization. The result is listed in Table 6. 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 is consistent and small.

3 DEMO RECOMMENDER ON THE PHONE

We include a demo of DQRM exported to an Android phone. From Figure 8, 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 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.