ON-FPGA TRAINING WITH ULTRA MEMORY REDUCTION: A LOW-PRECISION TENSOR METHOD

Kaiqi Zhang[1], Cole Hawkins[2], Xiyuan Zhang[3], Cong Hao[4], Zheng Zhang[1]

[1] Department of ECE, UC Santa Barbara. E-mail: {kzhang70, zzhang01}@ucsb.edu
[2]Department of Mathematics, UC Santa Barbara. E-mail: colehawkins@ucsb.edu
[3]Department of ECE, Carnegie Mellon University. E-mail: xiyuanzh@andrew.cmu.edu
[4]School of ECE, Georgia Institute of Technology. E-mail: callie.hao@gatech.edu

ABSTRACT

Various hardware accelerators have been developed for energy-efficient and real-time inference of neural networks on edge devices. However, most training is done on high-performance GPUs or servers, and the huge memory and computing costs prevent training neural networks on edge devices. This paper proposes a novel tensor-based training framework, which offers orders-of-magnitude memory reduction in the training process. We propose a novel rank-adaptive tensorized neural network model, and design a hardware-friendly low-precision algorithm to train this model. We present an FPGA accelerator to demonstrate the benefits of this training method on edge devices. Our preliminary FPGA implementation achieves $59\times$ speedup and $123\times$ energy reduction compared to embedded CPU, and $292\times$ memory reduction over a standard full-size training.

1 INTRODUCTION

Modern neural networks consume huge memory and computing resources during both training and inference. This challenge is compounded by increasing demands for on-device training to preserve data privacy (Teerapittayanon et al., 2017) and to increase energy efficiency. Energy-efficient and low-latency inference with limited hardware resources have been well studied at both the algorithm (LeCun et al., 1990; Neklyudov et al., 2017; Lebedev et al., 2014; Hinton et al., 2015; Han et al., 2015; Zhou et al., 2017) and hardware levels (Chen et al., 2016; 2019; Zhang et al., 2015; Hao et al., 2019; Xu et al., 2020). However, training on resource-constrained hardware remains an open challenge. Most on-device training algorithms utilize low-precision optimization (Courbariaux et al., 2015; Hubara et al., 2017) to reduce the computing and memory cost per parameter in the training process. However, the cost reduction is limited to *a single order of magnitude* even if the most recent ultra low-precision 4-bit training (Sun et al., 2020) is employed.

Paper Contributions. This paper presents, for the first time, an *end-to-end* neural network training framework on FPGA with *orders-of-magnitude memory reduction*. This is achieved by developing a low-precision tensorized training framework. We propose a rank-adaptive tensorized training model, which employs a Bayesian method for automatic tensor rank determination and model compression in the training process. We further reduce the memory by employing low-precision computation to train the tensorized model. We use our algorithm to train a two-layer neural network on a Xilinx MPSoC, which stores all model parameters on chip, achieves $59\times$ speedup and $123\times$ energy reduction compared to embedded CPU, and $292\times$ memory reduction compared to original model.

2 PRELIMINARY

We use Tensor-Train Matrix (TTM) decomposition (Oseledets, 2011) to compactly represent the neural network parameters in the training process.

Definition 1. *Let $\mathbf{W} \in \mathbb{R}^{J\times I}$ be the weight matrix in a fully connected layer and let $I = \prod_{n=1}^{d} I_n, J = \prod_{n=1}^{d} J_n$ be a factorization of its dimensions. We reshape $\mathbf{W}$ into a tensor $\mathcal{W}$ with dimensions $J_1 \times \cdots \times J_d \times I_1 \times \cdots \times I_d$ (Novikov et al., 2015). The tensor-train matrix (TTM) factorization expresses each data element of $\mathcal{W}$ as a series of matrix products.*

$$\mathcal{W}(j_1, \ldots, j_d, i_1, \ldots, i_d) = \mathcal{G}_1(:\, j_1, i_1, :)\mathcal{G}_2(:, j_2, i_2, :) \cdots \mathcal{G}_d(:, j_d, i_d, :)$$

Each TT-core factor $\mathcal{G}_n \in \mathbb{R}^{R_{n-1}\times J_n\times I_n\times R_n}$ is an order 4-tensor, and $R_0{=}R_d{=}\,1$. The vector $\mathbf{R} = (R_0, R_1, \cdots, R_d)$ is called the TT-rank. This TTM format requires $\sum_{n=1}^{d} R_{n-1}J_nI_nR_n$ parameters to represent $\mathbf{W}$, which is much smaller than the original number of variables $\prod_n J_nI_n$. We may expect huge memory reduction if the TTM format can be employed in training, but it is challenging to decide the TT rank $\mathbf{R}$ in practice. Existing methods often use a fixed-rank training (Novikov et al., 2015; Calvi et al., 2019; Khrulkov et al., 2019) which require combinatoral rank searches and multiple training runs. Therefore, they are not suitable for one-shot on-device training.

3 Low-Precision Rank-Adaptive Tensorized Neural Network

To enable memory-efficient, one-shot and on-device training, this section proposes a rank-adaptive tensorized training method with low-precision optimization.

3.1 Rank-Adaptive Tenosirzed Training

To simplify the descriptions, we assume that the parameters in each layer are represented with d TT core factors $\boldsymbol{\theta} = \{\mathcal{G}_n \in \mathbb{R}^{R_{n-1}\times J_n\times I_n\times R_n}\}_{n=1}^{d}$. Here R_n is an initial rank for dimension n, and it is larger than the actual rank parameter that will be determined later in training. Our method applies to general cases that have multiple layers with each layer being parameterized by some TT cores. We propose to train a tensorized neural network with the following rank-shrinkage model:

$$\min \mathcal{L}\left(\boldsymbol{\theta}, \{\boldsymbol{\lambda}_n\}_{n=1}^{d-1} \,|\mathcal{D}\right) = \frac{1}{|\mathcal{D}|}\sum_{\{\mathbf{x}_i, y_i\}\in\mathcal{D}} \mathrm{CE}\left(f\left(x_i, \boldsymbol{\theta}\right), y_i\right) + g\left(\boldsymbol{\theta}, \{\boldsymbol{\lambda}_n\}_{n=1}^{d-1}\right) \tag{1}$$

$$\text{with } g(\cdot) = \sum_{n=1}^{d-1}\sum_{r_n=1}^{R_n}\left[\frac{\|\mathcal{G}_n(:,:,:,r_n)\|_F^2}{\boldsymbol{\lambda}_n(r_n)} + \frac{1+R_{n-1}I_nJ_n}{2}\log\left(\boldsymbol{\lambda}_n(r_n)\right)\right] \tag{2}$$

where $\mathcal{D}$ is the training data set, $\mathrm{CE}(\cdot,\cdot)$ is a cross-entropy loss, and $\boldsymbol{\lambda}_n \in \mathbb{R}^{R_n}$ denotes a group of rank-controlling hyper-parameters for TT core $\mathcal{G}_n$.

Rank Reduction. Our objective function is the negative log-posterior of the Bayesian model in Hawkins et al. (2020). Empirical results shows this prior leads to good trade-off between model size and accuracy, and it is hyper-parameter-free which eliminates the need of hyper-parameter fine-tuning. Every element of subtensor $\mathcal{G}_n(:,:,:,r_n)$ (which is obtained by fixing the 4-th index of $\mathcal{G}_n$) is equipped with a zero-mean Gaussian prior with a variance $\boldsymbol{\lambda}_n(r_n)$. Each element of $\boldsymbol{\lambda}_n$ is further assumed to have a Log-Uniform prior distribution, such that some elements of $\boldsymbol{\lambda}_n$ will approach 0 with a high probability. In practice, once $\boldsymbol{\lambda}_n(r_n)$ becomes very small the variables in subtensor $\mathcal{G}_n(:,:,:,r_n)$ shrink to 0, and they can be removed from $\mathcal{G}_n$, leading to a rank reduction.

3.2 Training Quantized Tensor Factors

To further reduce the memory and computing cost, we use BinaryConnect (Courbariaux et al., 2015) to perform low-precision training. BinaryConnect keeps the real values of all low-precision parameters in a buffer. In each iteration, the gradients are accumulated in the buffer, and the low-precision parameters are updated by quantizing the buffer. Let a real-valued TT factor be $\mathcal{G}_n$, denote its quantized variant as $\tilde{\mathcal{G}}_n$ and $Q(\cdot)$ as the quantization function, the update process for each TT core is

$$\mathcal{G}_n^{(t+1)} = \mathcal{G}_n^{(t)} - \eta_t \nabla_{\tilde{\mathcal{G}}_n^t}\mathcal{L}\left(\{\tilde{\mathcal{G}}_n^t\}_{n=1}^{d}, \{\boldsymbol{\lambda}_n\}_{n=1}^{d-1}\,|\mathcal{B}\right), \quad \tilde{\mathcal{G}}_n^{(t+1)} = Q(\mathcal{G}_n^{(t+1)}), \tag{3}$$

where $\mathcal{B}\in\mathcal{D}$ is a batch of randomly sampled training data, and t is an index of the low-precision stochastic gradient-descent iteration. In iteration t, the hyper-parameters are updated as:

$$\boldsymbol{\lambda}_n^t(r_n) = \frac{2}{1+R_{n-1}I_nJ_n}\|\mathcal{G}_n^t(:,:,:,r_n)\|_F^2. \tag{4}$$

The quantization function is not differentiable. Therefore, we use the straight-through estimator (STE) (Bengio et al., 2013) to approximate the gradient of a quantization function. Specifically we use the gradient of a smooth function instead of the original non-differentiable activation in the backpropagation. This work uses a clipped ReLU as the STE. For ReLU activation function, denote $\mathbf{1}(\cdot)$ as the indicator function, the backpropagation rule can be written as $\frac{\partial}{\partial x} = \mathbf{1}(x\geq 0)\frac{\partial}{\partial y}$. We use 4 bits to represent TT factors, 8 bits for activations and bias, and 16 bits for the gradients. Since the

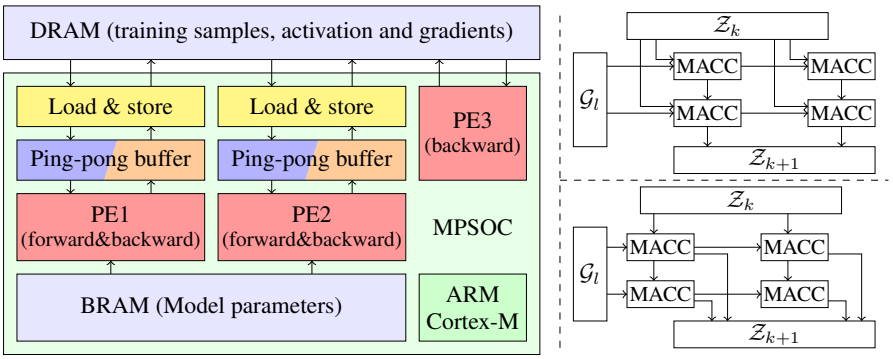

Figure 1: Left: overall view of the FPGA accelerator. Right: the structures of PE1 (top) and PE2 (bottom). MACC means "multiply and accumulator".

processing elements (PE) are shared between forward and backward propagations, they are designed to handle 16-bit activations or gradients and 4-bit TT factors. During forward propagation only the 8 least significant bits (LSB) of the 16 bits are used. Therefore our trained model can be deployed for inference on edge devices with only 8 bits for activation and bias as well as 4 bits for TT factors.

3.3 Automatic Scale Selection

In fixed-point representations, each variable needs to be scaled carefully to avoid overflow or large quantization errors. Furthermore, a bit shift is needed if the scaling factors differ by $2^i\times$. In the training process, the values of activation functions and gradients can vary by several orders of magnitude. To handle this issue, we determine the scaling factor on the fly. The scaling factors of all variables are enforced to be 2^k. We allow different scaling factors for each activation, gradient or intermediate result but share scaling factors across different data samples and different neurons of the same layer. The scaling factors of the TT factors are fixed. To determine the scaling factor of the activation and gradients, we track the mean of their absolute values during training, and we enforce it to be in the range $[0.1, 0.3]$ by dynamically adjusting the scaling factor. This allows a small margin to avoid overflows, while making the most use of the hardware bits to reduce quantization errors.

4 FPGA Implementation

This section describes the FPGA implementation for our on-device tensorized training method. The overall FPGA design is shown in Fig 1. The detailed computations of forward and backward propagation are given in the Appendix. During training, the data samples, activations, and gradients are stored in the off-chip DRAM. Due to our low-rank tensorization, all the model parameters may be stored in the on-chip BRAM. The overall training involves three steps: forward propagation, backward propagation, and model parameter update. The forward and backward propagations run on the FPGA programmable logic; the TT factors $\mathcal{G}_n$'s and rank parameters $\boldsymbol{\lambda}_n$'s are updated on the embedded ARM core, which usually take less than 1% of the total computing time. We design three processing elements (PEs) for the forward and backward propagation: PE1 and PE2 are shared by the forward and backward propagation and can take advantage of different data locality during tensor contraction. PE3 is dedicated to computing the outer products in a backward propagation.

PE1 and PE2 are shown in Fig. 1. The computation involves three steps: loading a tensor slice from DRAM to the on-chip buffer, performing the multiply-and-accumulate (MAC) operations, and sending the results back to DRAM. With ping-pong buffers, the (most time-consuming) second step can be executed in parallel with other steps. We require only two key computational kernels to perform forward propagation. We describe the kernels of the first two PEs, and provide the details of the forward and backward pass in the Appendix. PE1 is used for a two-index tensor contraction which contains the last dimension of both operands. By reshaping the d-dimensional tensors into 3-dimensional tensors without permutation, the computation can be written element-wise as

$$\mathcal{Z}_{k+1}(a,d) = \sum_{b,c} \mathcal{Z}_k(a,b,c)\mathcal{G}_l(b,d,c). \tag{5}$$

The second PE (PE2) performs a tensor contraction along a single dimension that is not the last:

$$\mathcal{Z}_{k+1}(a,d,c) = \sum_{b} \mathcal{Z}_k(a,b,c)\mathcal{G}_l(b,d). \tag{6}$$

Table 1: Fashion MNIST training result.

Method	Training accuracy	Testing accuracy	Model parameters	Memory in bits	Memory reduction
Vanilla	95.75%	89.27%	4.67×10^5	1.49×10^7	N/A
Floating, w/o prior	92.54%	88.03%	1.48×10^4	4.74×10^5	31.4×
Fixed, w/o prior	88.31%	86.67%	1.48×10^4	6.13×10^4	243×
Floating, w/ prior	90.17%	87.88%	1.08×10^4	3.46×10^5	43.1×
Fixed, w/ prior (proposed)	85.45%	84.86%	1.22×10^4	5.11×10^4	292×

Table 2: Resource utilization of two-layer tensorized neural network.

resource	LUT	FF	DSP	BRAM	power
used	56131	30155	278	77	1.2W
available	70560	141120	360	432	-
utilization	79.55%	21.37%	77.22%	17.82%	-

Here $\mathcal{Z}_{k+1}$ and $\mathcal{Z}_k$ denote the intermediate results, and $\mathcal{G}_l$ is a reshaped TTM factor. Both PEs have 128 MACs that operate in parallel. Only order-2 and order-3 tensor contractions are required due to a reshape operation (see Table 3 in the Appendix). In a forward propagation, the mode index $l = d-k$; in a backward propagation, $l = k+1$. In PE1, we split the first operand based on the first dimension of $\mathcal{Z}_k$. We parallelize the computing along the last dimension (associated with index c) by a factor of 16, and along the first dimension (w.r.t. index a) by a factor of 8. This allows each element in the first operand to be shared across 8 multipliers. To simplify the design, we enforce the dimension size of index c to be a multiplier of 16, which is equivalent to enforcing the last dimension of both input and output tensors to be a multiplier of 16. Similarly, in PE2, the first operand is split based on dimensions associated with indices a and c. The computations associated with indices c and d are parallelized by a factor of 16 and 8, respectively. We introduce PE3 to perform outer products. The throughput of this step is bounded by the memory bandwidth of storing, so we parallelize the computing along the last dimension I_d only by a factor of 16. The elements of the second operand is cached, while the first operand is read from DRAM directly. A total of 16 multipliers calculate the product of elements from the two operands and the results are written to DRAM directly without caching, because they are not further used in this step.

5 Experiments and results

We implement our low-precision rank-adaptive tensorized training on an Avnet Ultra96-V2 FPGA board and use it to train a two-layer neural network to classify the FashionMNIST dataset. We implemented the neural network in HLS codes, yet they can be easily modified to fit different network structures. The detailed experimental setup, including the structure of neural network and the hyperparameters, is provided in the Appendix B. As shown in Table 1, our method achieves $294\times$ memory reduction for the model parameters compared with the standard non-tensorized training.

The hardware resource utilization is listed in Table 2. We compare the time and memory usage of tensorized neural network training on FPGA and on an embedded processor, a Raspberry Pi 3B with Quad Core 1.2GHz ARM processor. We use the Pytorch and Tensorly modules to implement our training algorithm on the embedded processor. For the FPGA we set the clock rate to 100MHz. The forward and backward propagation takes 0.09s per batch of 64 samples, where the embedded processor takes 5.34s. The estimated power consumption of our design is 1.2W compared to 2.5W of Raspberry Pi, indicating that our FPGA accelerator is $123\times$ more energy-efficient.

6 Conclusion and Future Work

We have proposed a low-precision tensor method to train neural networks on edge devices. By end-to-end compressed training, our approach produce an ultra-compact model from scratch while saving significant hardware resources during the training. Our algorithm uses a rank-adaptive approach to determine model complexity, and it has achieved $292\times$ memory reduction compared to the baseline model with only a 4.5% testing accuracy loss on Fashion MNIST. The FPGA implementation has achieved $59\times$ speedup and $123\times$ energy reduction than the training on an embedded CPU.

In the future we plan to: (1) further optimize the low-precision tensorized training algorithm and its FPGA implementation, and (2) demonstrate on-device training for large-size neural networks.

REFERENCES

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Giuseppe G Calvi, Ahmad Moniri, Mahmoud Mahfouz, Qibin Zhao, and Danilo P Mandic. Compression and interpretability of deep neural networks via tucker tensor layer. *arXiv:1903.06133*, 2019.

Yao Chen, Kai Zhang, Cheng Gong, Cong Hao, Xiaofan Zhang, Tao Li, and Deming Chen. T-dla: An open-source deep learning accelerator for ternarized dnn models on embedded fpga. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 13–18. IEEE, 2019.

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.

Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-mei Hwu, and Deming Chen. Fpga/dnn co-design: An efficient design methodology for 1ot intelligence on the edge. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2019.

Cole Hawkins, Xing Liu, and Zheng Zhang. Towards compact neural networks via end-to-end training: A bayesian tensor approach with automatic rank determination. *arXiv:2010.08689*, 2020.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531*, 2015.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Valentin Khrulkov, Oleksii Hrinchuk, Leyla Mirvakhabova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv:1901.10787*, 2019.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *NIPS*, pp. 598–605, 1990.

Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. Structured Bayesian pruning via log-normal multiplicative noise. In *NIPS*, pp. 6775–6784, 2017.

Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pp. 442–450, 2015.

Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *NIPS*, 33, 2020.

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *Intl. Conf. Distributed Computing Systems*, pp. 328–339, 2017.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 40–50, 2020.

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 161–170, 2015.

Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv:1702.03044*, 2017.

A DETAIL ABOUT TENSOR CONTRACTION

A.1 FORWARD PROPAGATION

To perform the forward propagation we take a vector $\mathbf{x}$ and the TTM cores $\{\mathcal{G}_n\}_{n=1}^{d}$ of the tensorized weight matrix $\mathbf{W}$. The forward propagation operation is

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{7}$$

We detail the tensorized forward and backward passes where $\mathcal{X}$ is achieved by reshaping $\mathbf{x}$ and $\mathcal{Y}$ is the tensorized output.

The forward and backward propagation of neural network involves tensor contraction. Denote the input dimensions as $I_1, I_2, \ldots, I_d$, output dimensions as $J_1, J_2, \ldots, J_d$, and the rank as $R_1, R_2, \ldots R_{d-1}$. the forward propagation involves the following computation:

$$\mathcal{Z}_1(i_1, i_2, \ldots, r_{d-1}, j_d) = \sum_{i_d} \mathcal{X}(i_1, i_2, \ldots i_d) \times \mathcal{G}_d(r_{d-1}, j_d, i_d) \tag{8}$$

$$\mathcal{Z}_2(i_1, \ldots, r_{d-2}, j_{d-1}, j_d) = \sum_{i_{d-1} r_{d-1}} \mathcal{Z}_1(i_1, i_2, \ldots, r_{d-1}, j_d) \times \mathcal{G}_{d-1}(j_{d-1}, r_{d-1}, r_{d-2}, i_{d-1}) \tag{9}$$

$$\ldots$$

$$\mathcal{Y}(j_1, j_2, \ldots j_d) = \sum_{i_1} \mathcal{Z}_{d-1}(i_1, r_1, j_2, \ldots, j_d) \times \mathcal{G}_1(j_1, r_1, i_1) \tag{10}$$

In these expressions, we denote $\mathcal{X}$ as the input to this layer, $\mathcal{G}_n$ as TTM core factors, $\mathcal{Y}$ as the output and $\mathcal{Z}_i$ as an intermediate result.

A.2 BACKWARD PROPAGATION

In back propagation, there are two tasks:

- To compute the gradients with respect to the inputs of the layer.
- To compute the gradients with respect to the compressed model parameters (TTM core factors) of the layer.

To compute the gradients with respect to the inputs, we denote $\hat{\mathcal{X}}$ as the gradient of input to this layer and $\hat{\mathcal{Y}}$ as the gradient of the output of this layer. The computation is shown below:

$$\mathcal{Z}_1(i_1, r_1, j_2, \ldots, j_d) = \sum_{j_1} \hat{\mathcal{Y}}(j_1, j_2, \ldots, j_d) \times \mathcal{G}_1(j_1, i_1, r_1) \tag{11}$$

$$\mathcal{Z}_2(i_1, i_2, r_2, \ldots, j_d) = \sum_{r_1 j_2} \mathcal{Z}_1(i_1, r_1, j_2, \ldots, j_d) \times \mathcal{G}_2(r_1, j_2, i_2, r_2) \tag{12}$$

$$\ldots$$

$$\hat{\mathcal{X}}(i_1, i_2, \ldots, i_d) = \sum_{j_d} \mathcal{Z}_{d-1}(i_1, \ldots, i_{d-1}, r_{d-1}, j_d) \times \mathcal{G}_d(r_{d-1}, j_d, i_d) \tag{13}$$

The first equation is to compute the gradients directly. The second is to compute the gradients with respect to full weights and then accumulate them and compute the gradients with respect to the TTM factors. The former method is more efficient if the batch size is small and the compressed model is small, while the latter is more efficient otherwise. In our work, we start training with a large rank initial guess (i.e., a larger model), so the latter method is more efficient. The first step, computing the gradient w.r.t. the full weight matrix (denoted as $\hat{\mathcal{W}}$), requires a simple outer product which is computed by PE3:

$$\hat{\mathcal{W}}(j_1, i_1, j_2, i_2, \ldots, j_d, i_d) = \mathcal{X}(i_1, i_2, \ldots, i_d) \times \hat{\mathcal{Y}}(j_1, j_2, \ldots, j_d)$$

The first operand of this PE is the input to this layer during forward propagation, and the second operand is the gradient of the output. After the gradient has been accumulated in a batch, the

Eq.	PE	a	b	c	d
(8)	PE1	$I_1 I_2 \dots I_{d-1}$	1	I_d	$R_{d-1} J_d$
(9)	PE2	$I_1 I_2 \dots I_{d-2}$	$I_{d-1} R_{d-1}$	J_d	$R_{d-2} J_{d-1}$
(10)	PE2	1	$I_1 R_1$	$J_2 \dots J_d$	J_1
(11)	PE2	1	J_1	$J_2 \dots J_d$	$I_1 R_1$
(12)	PE2	I_1	$R_1 J_2$	$J_3 \dots J_d$	$I_2 R_2$
(13)	PE1	$I_1 I_2 \dots I_{d-1}$	R_{d-1}	J_d	I_d
(14)	PE1	$J_1 I_1 \dots J_{d-1} I_{d-1}$	1	$J_d I_d$	R_{d-1}
(15)	PE1	$J_1 I_1 \dots J_{d-2} I_{d-2}$	1	$J_{d-1} I_{d-1}$	R_{d-2}
(16)	PE1	$J_1 I_1$	1	$J_2 I_2 R_2$	R_1
(17)	PE2	1	$J_1 I_1$	$J_2 I_2 R_2$	R_1
(18)	PE2	1	$J_1 I_1$	$J_2 I_2 J_3 I_3 R_3$	R_1
(19)	PE2	1	$R_1 J_2 I_2$	$J_3 I_3 R_3$	R_2

Table 3: PE and operand of each expression.

gradient with respect to the factors denoted by $\hat{\mathcal{G}}_i$ can be computed by contracting the gradient of full weight with the tensor factors:

$$\mathcal{Z}_{1,1}(j_1, i_1, j_2, i_2, \dots, j_{d-1}, i_{d-1}, r_{d-1}) = \sum_{j_d, i_d} \hat{\mathcal{W}}(j_1, i_1, j_2, i_2, \dots, j_d, i_d) \times \mathcal{G}_d(r_{d-1}, j_d, i_d) \tag{14}$$

$$\mathcal{Z}_{2,1}(j_1, i_1, j_2, i_2, \dots, j_{d-2}, i_{d-2}, r_{d-2}) = \sum_{j_{d-1}, i_{d-1}, r_{d-1}} \mathcal{Z}_{1,1}(j_1, i_1, j_2, i_2, \dots, j_{d-1}, i_{d-1}, r_{d-1}) \times \mathcal{G}_{d-1,1}(r_{d-2}, j_{d-1}, i_{d-1}, r_{d-1}) \tag{15}$$

$\dots$

$$\hat{\mathcal{G}}_1(j_1, i_1, r_1) = \sum_{j_2, i_2, r_2} \mathcal{Z}_{d-2,1}(j_1, i_1, j_2, i_2, r_2) \times \mathcal{G}_2(r_1, j_2, i_2, r_2) \tag{16}$$

$$\hat{\mathcal{G}}_2(r_1, j_2, i_2, r_2) = \sum_{j_1, i_1} \mathcal{Z}_{d-2,1}(j_1, i_1, j_2, i_2, r_2) \times \mathcal{G}_1(j_1, i_1, r_1) \tag{17}$$

$$\mathcal{Z}_{d-3,2}(r_1, j_2, i_2, j_3, i_3, r_3) = \sum_{j_1, i_1} \mathcal{Z}_{d-3,1}(j_1, i_1, j_2, i_2, j_3, i_3, r_3) \times \mathcal{G}_1(j_1, i_1, r_1) \tag{18}$$

$$\hat{\mathcal{G}}_3(r_2, j_3, i_3, r_3) = \sum_{r_1, j_2, i_2} \mathcal{Z}_{d-3,2}(r_1, j_2, i_2, j_3, i_3, r_3) \times \mathcal{G}_2(r_1, j_2, i_2, r_2) \tag{19}$$

$\dots$

Note that the $\mathcal{Z}_{1,1}$ is shared to get $\hat{\mathcal{G}}_1$ and $\hat{\mathcal{G}}_2$, and $\mathcal{Z}_{2,1}$ is shared to get $\hat{\mathcal{G}}_1$ and $\hat{\mathcal{G}}_3$.

A.3 Use of PE

To compute the gradient with respect to the tensor factors, as in equation (14)-(19), we can reuse PE1 and PE2. In either case we can reshape the tensor in order to apply Equation (5) or Equation (6). If tensor contraction is executed along the the last dimension as in (8)(13)(14)(15)(16), PE1 (Equation (5)) is used; otherwise, as in (9)(10)(11)(12)(17)(18)(19), PE2 (Equation (6)) is used. The way to shape the tensors is given in Table 3. When necessary we add a dimension with size one and/or perform a reshape operation so that our computation fits the appropriate PE tensor contraction.

B Detail about numerical experiment

We used C++ to implement fixed point tensor contraction and used Pytorch to implement high level methods (ADAM, rank parameters update). In order to fit the requirement on the shape of tensors, we zero pad the input to 28×32 and decompose it to $7 \times 4 \times 2 \times 16$. There are 512 neurons in the

hidden layer decomposed into $4 \times 4 \times 2 \times 16$ for the first layer, and 32×16 for the second layer. The output is decomposed into 1×16. We trained this model for FashionMNIST dataset (Xiao et al., 2017), which has the same shape and size as MNIST dataset but is more complicated and can better represent modern machine learning tasks. To accelerate training, we pretrain the models on MNIST dataset. We train the model for 30 epochs and compare both standard floating-point computation in Pytorch (Floating) and our simulator (Fixed), and also compare the training methods with or without the low rank TT priors. We report the epoch with highest testing accuracy. For run-time comparison, only the time on forward and backward propagation is included, as the rest part (optimizer) is the same on both devices.