Towards Compact Neural Networks via End-to-End Training: A Bayesian Tensor Approach with Automatic Rank Determination

Cole Hawkins∗, Xing Liu†, and Zheng Zhang‡

Abstract. While post-training model compression can greatly reduce the inference cost of a deep neural network, uncompressed training still consumes a huge amount of hardware resources, run-time and energy. It is highly desirable to directly train a compact neural network from scratch with low memory and low computational cost. Low-rank tensor decomposition is one of the most effective approaches to reduce the memory and computing requirements of large-size neural networks. However, directly training a low-rank tensorized neural network is a very challenging task because it is hard to determine a proper tensor rank a priori, which controls the model complexity and compression ratio in the training process. This paper presents a novel end-to-end framework for low-rank tensorized training of neural networks. We first develop a flexible Bayesian model that can handle various low-rank tensor formats (e.g., CP, Tucker, tensor train and tensor-train matrix) that compress neural network parameters in training. This model can automatically determine the tensor ranks inside a nonlinear forward model, which is beyond the capability of existing Bayesian tensor methods. We further develop a scalable stochastic variational inference solver to estimate the posterior density of large-scale problems in training. Our work provides the first general-purpose rank-adaptive framework for end-to-end tensorized training. Our numerical results on various neural network architectures show orders-of-magnitude parameter reduction and little accuracy loss (or even better accuracy) in the training process. Specifically, on a very large deep learning recommendation system with over $4.2 \times 10^9$ model parameters, our method can reduce the variables to only $1.6 \times 10^6$ automatically in the training process (i.e., by $2.6 \times 10^4$ times) while achieving almost the same accuracy.

1. Introduction. Despite their success in many applications, deep neural networks are often over-parameterized, requiring extensive computing resources in their training and inference. For instance, the VGG-19 network requires 500M memory [1] for image recognition; the DLRM deep learning recommendation system [2] has over 540M parameters, and the practical model deployed by Facebook is even larger. It has become a common practice to reduce the size of neural networks before deploying them in various scenarios ranging from cloud services to embedded systems to mobile applications.

Motivated by real-time inference and hardware resource constraints [3], numerous techniques have been developed to build compact models [4, 5, 6] after training. Representative approaches include pruning [6, 7, 8, 9], quantization [10, 11, 12], knowledge distillation [13, 14, 15], and low-rank factorization [16, 17, 18, 19, 20]. Among these techniques, low-rank tensor compression [19, 20, 21, 22, 23, 24] has achieved possibly the most significant compression, leading to promising reduction of FLOPS and hardware cost [19, 25, 26, 27]. The recent progress of hardware design and algorithm/hardware co-design [26, 27, 28, 29, 30] of tensor operations can further reduce the run-time and boost the energy efficiency of tensorized models on clusters and on edge devices [e.g., implementations on FPGA (Field Programmable Gate Arrays) and ASIC (Application-Specific Integrated Circuits)]. While post-training compression methods

∗Department of Mathematics, University of California at Santa Barbara (colehawkins@math.ucsb.edu).
†Facebook AI Systems Hardware/Software Co-design (xingl@fb.com).
‡Department of Electrical and Computer Engineering, University of California at Santa Barbara (zhengzhang@ece.ucsb.edu).

This manuscript is for review purposes only.
can reduce the cost of deploying a deep neural network, they cannot reduce the training cost. Training is a much more challenging task than inference, consuming much more money, run-time, energy, and hardware resources. The study [31] has shown that training some common natural language processing models on the cloud could emit $5 \times$ carbon dioxide compared with the lifetime emissions of the average American car. Meanwhile, the increasing concerns about data privacy have become a driving force of training on resource-constrained edge devices [32]. These high costs and hardware constraints associated with neural network training motivate us to ask the following question: “Is it possible to train a compact neural network model from scratch?” Essentially, the computing and hardware cost could be significantly reduced on various platforms if we can avoid the full-size uncompressed training. While pruning techniques can be used in training to generate sparse models [7, 33], they do not necessarily reduce the unknown variables or computational cost of training. Recently low-precision training methods [34,35,36,37,38,39,40] have achieved great success on various platforms including cloud computers, distributed platforms and edge devices. These methods have effectively reduced the memory cost, data communications and training time by using less binary bits to represent weight parameters, activation functions and gradients. However, the memory cost reduction is quite limited in low-precision methods, which has limited their applications in resource-constraint scenarios. For instance, the maximum possible memory reduction (in an ideal case) is only $8 \times$ even if we replace a 32-bit training method with the most recent ultra low-precision 4-bit training framework [40].

1.1. Contributions. In this paper, we will present a rank-adaptive end-to-end tensorized training method to generate ultra-compact neural networks directly from scratch. As shown in Fig. 1 (a), our method avoids the expensive full-size training compared with existing post-training tensor compression methods [19, 20, 21, 23]. Different from low-precision training methods [34,35,36,37,38,39,40] which only offer at most a single order of magnitude in variable reductions, our method may reduce the training variables by several orders of magnitude. Our method can achieve further training variable reductions if it is combined with low-precision numerical operations. As a result, this work can make a great impact: it may enable energy-efficient training of medium- or large-size neural networks on edge devices (e.g, smart phones, embedded GPUs and FPGA), which is impossible to achieve at this moment with existing training methods. Some recent works have studied low-rank tensorized training [41,42,43], but they fix the tensor ranks before training. It is very hard to guess or select a proper tensor rank parameter a-priori in practice, therefore one often has to perform extensive combinatorial searches and many training runs until a good rank parameter is found.

In this paper, we make the following specific contributions to achieve efficient one-shot end-to-end tensorized training:

- A general-purpose rank-adaptive Bayesian tensorized model. The training cost, model complexity and accuracy of a tensorized neural network are controlled by tensor ranks, which are unknown a priori. Exactly determining a tensor rank is NP-hard even for the simplest tensor factorization problems [44]. In order to avoid the extensive combinatorial searches of tensor ranks in recent works [41,42,43], we develop a novel Bayesian model to determine both tensor ranks and factors automatically. Many advances in tensor-based modeling are problem-specific and focus on a

This manuscript is for review purposes only.
Figure 1: (a) Key idea of this work. Conventional train-then-compress approaches can generate small neural networks but have high training costs. In contrast, the proposed end-to-end tensorized training can reduce the training variables significantly and directly produce ultra-compact neural networks. (b) Effectiveness of this approach on a realistic DLRM benchmark. Standard training requires 4.25 billion training variables. In our proposed method, a TTM-format tensorization reduces the training variables to only 2.36 million, which is further reduced to only 164K in the training process due to the automatic tensor rank determination.

- A highly scalable Bayesian solver for the proposed tensorized neural networks. Training large-scale Bayesian tensorized neural networks is computationally expensive. Recently, mean-field variational inference has been employed to solve tensor completion problems \[45, 46, 47, 48\], but this approach does not work for tensorized neural networks because of the highly nonlinear forward propagation model. Meanwhile, alternative approaches such as advanced MCMC \[49, 50\] and Stein variational gradient descent \[51\] do not scale well for large-scale neural networks since they require storing multiple copies of neural network models. Therefore, this work will develop a more scalable solver by improving the approximate Bayesian inference method proposed in \[52\]. Specifically, we observe that directly employing the solver in \[52\] causes large gradient variance in our tensorized model. Therefore, we propose to simplify the posterior density of some rank-controlling hyper parameters, and further develop an analytical/numerical hybrid approach for the iterations of stochastic gradient descent. This customized Bayesian solver infers the unknown tensor factors and tensor ranks of realistic neural networks in a single training run, enabling training and quantifying the uncertainty of extremely large-scale deep learning models that are beyond the capability of existing Bayesian solvers.

- Extensive numerical validations. We test our algorithms on four benchmarks with model parameters ranging from \(4 \times 10^5\) to \(4.2 \times 10^9\). Our method can reduce
the training variables by several orders of magnitude with little or even no loss of
accuracy. For instance, our method achieves up to $26,000 \times$ parameter reduction in
the training process on the large-scale DLRM model as shown in Fig. 1 (b). We also
compare our methods with existing tensorized neural network methods [21, 41, 42, 43]
including post-training tensor compression and fixed-rank tensorized training, which
clearly demonstrates the advantage of our rank-adaptive training method in terms of
variable reduction and model accuracy.

To the best of our knowledge, this work is the first end-to-end Bayesian method that automat-
dically determines the tensor rank in large-scale neural network training (with billions of model
parameters) and supports multiple low-rank tensor formats simultaneously. This work will en-
able energy-efficient and low-cost training of realistic neural networks in resource-constrained
scenarios such as internet of things (IoT), robotic systems and mobile phones. The Bayesian
solution will enable uncertainty quantification of the prediction results, which is important in
safety-critical applications such as autonomous driving [53] and medical imaging [54].

1.2. Related Work. There is a massive body of work studying the pruning [6,7,8], quanti-
ization [10,11,12], knowledge distillation [13,14,15], and low-rank compression [16,17,18,19,20]
of deep neural networks. This work is most related to the following previous results.

Rank Determination for Linear Tensor Problems. While exact tensor rank determination is
NP-hard [44], some approximate methods have been developed to estimate the tensor ranks
in tensor factorization and completion. Optimization-based approaches employ a heuristic
tensor nuclear norm as the surrogate of tensor rank [55,56,57], but they require expensive
regularization on the unfolded tensor. A nice alternative solution is to use Bayesian inference
to automatically estimate tensor ranks from observed data [45,58,59,60,61]. Current Bayesian
tensor methods solve tensor factorization, completion and regression problems on small-scale
data where the observed data is a linear function of the hidden tensor. These problems allow
closed-form parameter updates in mean-field Bayesian inference [45,59,60,61]. Sampling-based
Bayesian methods (i.e. MCMC) require storing thousands of copies of the model, which is
not feasible for large neural networks. Because the mean-field variational approach for linear
tensor problems [45,46] does not work for tensorized neural networks, this paper develops a
scalable solver based on stochastic variational inference [52].

Tensorized Neural Networks. Most work uses tensor decomposition to compress pre-trained
neural networks. Examples include employing CP and Tucker factorizations to compress
convolutional layers [19,20]. In these examples the convolutional filters are already in a
tensor form. It has been a common practice to reshape the weights in a fully connected layer
to a high-order tensor, then a tensor factorization can achieve much higher a compression ratio
than a matrix factorization on convolution layers [20]. As shown in [19,62], a neural network
compressed by low-rank tensor decomposition can consumes less memory, latency and energy
on resource-constraint platforms such as mobile phones. Some recent approaches train low-
rank tensorized neural networks [41,42,43] by assuming a low-rank tensorization with a fixed
rank. While it is possible to tune the tensor ranks in post-training tensor compression [19,20]
based on approximation errors, one has to use manual tuning or combinatorial search to
determine tensor ranks in existing tensorized training methods [41,42,43]. This has been a
major challenge that prevents one-shot training of realistic neural networks on edge devices.
2. Preliminaries.

2.1. Tensors and Tensor Decomposition. This paper uses lower-case letters (e.g., $a$) to denote scalars, bold lowercase letters (e.g., $\mathbf{a}$) to represent vectors, bold uppercase letters (e.g., $\mathbf{A}$) to represent matrices, and bold calligraphic letters (e.g., $\mathcal{A}$) to denote tensors. A tensor is a generalization of a matrix, or a multi-way data array. An order-$d$ tensor is a $d$-way data array $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$, where $I_n$ is the size of mode $n$. The $(i_1, i_2, \cdots, i_d)$-th element of $\mathbf{A}$ is denoted as $a_{i_1i_2\cdots i_d}$. An order-3 tensor is shown in Fig. 2 (a).

**Definition 2.1.** The mode-$n$ product of a tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is

$$\mathcal{B} = \mathbf{A} \times_n \mathbf{U} \iff b_{i_1\cdots i_{n-1}j_{i_{n+1}}\cdots i_d} = \sum_{i_n=1}^{I_n} a_{i_1\cdots i_d} u_{j_{i_n}}.$$  

(2.1)

The result is still a $d$-dimensional tensor $\mathcal{B}$, but the mode-$n$ size becomes $J$. In the special case $J = 1$, the $n$-th mode diminishes and $\mathcal{B}$ becomes an order-$d-1$ tensor.

A tensor has a massive number of entries if $d$ is large. This causes a high cost in both computing and storage. Fortunately, many practical tensors have a low-rank structure, and this property can be exploited to reduce the cost dramatically.

**Definition 2.2.** A $d$-way tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ is rank-1 if it can be written as a single outer product of $d$ vectors

$$\mathbf{A} = \mathbf{u}^{(1)} \odot \cdots \odot \mathbf{u}^{(d)}, \text{ with } \mathbf{u}^{(n)} \in \mathbb{R}^{I_n} \text{ for } n = 1, \cdots, d.$$  

Equivalently, each element of $\mathbf{A}$ can be written as

$$a_{i_1i_2\cdots i_d} = \prod_{n=1}^{d} u_{i_n}^{(n)},$$

where $u_{i_n}^{(n)}$ is the $i_n$-th element of the vector $\mathbf{u}^{(n)}$.

A rank-1 tensor can be stored with only $d$ vectors. Most tensors are not rank-1, but many can be well-approximated via tensor decomposition [63] if their ranks are low. In this paper we will use the following four tensor decomposition formats to reduce the parameters of neural networks.

**Definition 2.3.** The CP factorization [64, 65] expresses a $d$-way tensor $\mathbf{A}$ as the sum of multiple rank-1 tensors:

$$\mathbf{A} = \sum_{j=1}^{R} \mathbf{u}_j^{(1)} \odot \mathbf{u}_j^{(2)} \cdots \odot \mathbf{u}_j^{(d)}.$$  

(2.2)

Here $\odot$ denotes an outer product operator. The minimal integer $R$ that ensures the equality is called the CP rank of $\mathbf{A}$. To simplify notation we collect the rank-1 terms of the $n$-th mode into a factor matrix $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R}$ with

$$\mathbf{U}^{(n)}(:,j) = \mathbf{u}_j^{(n)}.$$  

(2.3)

This manuscript is for review purposes only.
A CP factorization with rank $R$ can be described with $d$ factor matrices $\{U^{(n)}\}_{n=1}^d$ using $R \sum_n I_n$ parameters.

**Definition 2.4.** The Tucker factorization [66] expresses a $d$-way tensor $\mathbf{A}$ as a series of mode-$n$ products:

$$\mathbf{A} = \mathbf{G} \times_1 U^{(1)} \times_2 \cdots \times_d U^{(d)}.$$  

Here $\mathbf{G} \in \mathbb{R}^{R_1 \times \cdots \times R_d}$ is a small core tensor, and $U^{(n)} \in \mathbb{R}^{I_n \times R_n}$ is a factor matrix for the $n$-th mode. The **Tucker rank** is the tuple $(R_1, \ldots, R_d)$. A Tucker factorization with ranks $R_n = R$ requires $R^d + R \sum_n I_n$ parameters.

**Definition 2.5.** The tensor-train (TT) factorization [67] expresses a $d$-way tensor $\mathbf{A}$ as a collection of matrix products:

$$a_{i_1 \cdots i_d} = g^{(1)}(:, i_1, :) g^{(2)}(:, i_2, :) \cdots g^{(d)}(:, i_d, :).$$

Each TT-core $g^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ is an order-3 tensor. The tuple $(R_0, R_1, \ldots, R_d)$ is the **TT-rank** and $R_0 = R_d = 1$.

The TT format uses $\sum_n R_{n-1} I_n R_n$ parameters in total and leads to more expressive interactions than the CP format.

Let $\mathbf{A} \in \mathbb{R}^{I \times J}$ be a matrix. We assume that the dimensions $I$ and $J$ can be factored as

---

**Figure 2:** (a): An order-3 tensor, (b) and (c): representations in CP and Tucker formats respectively, where low-rank factors are color-coded to indicate the corresponding modes. (d): TT representation of an order-$d$ tensor, where the purple lines and squares indicate $g^{(n)}(:, i_n, :)$, which is the $i_n$-th slice of the TT core $g^{(n)}$ obtained by fixing its second index.

This manuscript is for review purposes only.
follows:

\[
I = \prod_{n=1}^{d} I_n, \quad J = \prod_{n=1}^{d} J_n.
\]

We can reshape \( A \) into a tensor \( \mathbf{A} \) with dimensions \( I_1 \times \cdots \times I_d \times J_1 \times \cdots \times J_d \), such that the \((i, j)\)-th element of \( \mathbf{A} \) uniquely corresponds to the \((i_1, i_2, \ldots, i_d, j_1, j_2, \ldots, j_d)\)-th element of \( \mathbf{A} \).

The TT decomposition can extend to compress the resulting order-2d tensor as follows.

**Definition 2.6.** The tensor-train matrix (TTM) factorization expresses an order-2d tensor \( \mathbf{A} \) as \( d \) matrix products:

\[
a_{i_1 \ldots i_d j_1 \ldots j_d} = \mathbf{g}^{(1)}(:, i_1, j_1) \mathbf{g}^{(2)}(:, i_2, j_2) \ldots \mathbf{g}^{(d)}(:, i_d, j_d).
\]

Each TT-core \( \mathbf{g}^{(n)} \in \mathbb{R}^{I_n \times J_n \times R_n} \) is an order-4 tensor. The tuple \((R_0, R_1, R_2, \ldots, R_d)\) is the TT-rank and as before \( R_0 = R_d = 1 \). This TTM factorization requires \( \sum_n R_n - 1 \) \( I_n \) \( J_n \) \( R_n \) parameters to represent \( \mathbf{A} \).

We provide a visual representation of the CP, Tucker, and TT formats in Fig. 2 (b) – (d).

**2.2. Deep Neural Networks.** A deep neural network can be written as

\[
y = \mathbf{h}(x) = \mathbf{g}_L \left( \cdots \mathbf{g}_1(x) \right)
\]

where \( \mathbf{g}_k(\circ) \) represents layer \( k \), \( x \) is an input data sample (e.g., an image) and \( y \) is a predicted output label. In this work we focus on the following types of layers:

- **Fully connected layer:** A fully-connected layer \( \mathbf{g}_k \) is

\[
\mathbf{g}_k(x_k) = \sigma(\mathbf{W}_k x_k + \mathbf{b}_k)
\]

where \( \sigma \) is a nonlinear activation function (e.g., ReLU or sigmoid function), \( \mathbf{W}_k \) is a weight matrix, and \( \mathbf{b}_k \) is a bias vector. Most parameters in a fully connected layer are contained in the weight matrix \( \mathbf{W}_k \).

- **Embedding layer:** An embedding layer is defined as

\[
\mathbf{g}_k(X_k) = \mathbf{W}_k X_k
\]

where \( \mathbf{W}_k \) is the embedding matrix and the \( X_k \) is a sparse matrix with entries restricted to \( \{0, 1\} \). This type of layer is a common first layer of deep neural networks employed in natural language processing and recommendation tasks [43].

- **Convolutional layer:** A convolutional layer is

\[
\mathbf{g}_k(\mathcal{X}_k) = \sigma(\mathcal{W}_k \star \mathcal{X}_k + \mathbf{b}_k)
\]

where \( \sigma \) is a nonlinear activation function, \( \mathcal{W}_k \) is a weight tensor, \( \mathcal{X}_k \) is the input tensor of layer \( k \), “\( \star \)” is a convolution operator, and \( \mathbf{b}_k \) is a bias. For instance, let \( \mathcal{W} \in \mathbb{R}^{l \times l \times C_{in} \times C_{out}} \) be a 2D convolution for image processing with filter size \( l \times l \), \( C_{in} \) input output channels, and \( C_{out} \) output channels. The convolution operation is defined as

\[
\mathbf{y} = \mathcal{W} \star \mathbf{A} \iff y_{xyz} = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{c=1}^{C_{in}} w_{ijc} a_{x+i-1,y+j-1,c}.
\]
Considering parameter dependence, we can re-write (2.8) as
\[ (2.13) \quad y = h(x \mid \{W_k, b_k\}_{k=1}^L). \]
In a convolution layer \( W_k \) should be replaced with \( W_k \).

### 2.3. Tensorized Neural Networks.

In a modern neural network, \( \{W_k\}_{k=1}^L \) contain millions to billions of parameters. The resulting model consumes a huge amount of memory, energy and run-time in training and inference. This has become a major challenge across various hardware platforms. A promising solution is to generate a compact neural network via low-rank tensor compression [20, 21, 41] as follows:

- **Folding to high-order tensors.** A general weight matrix \( W \in \mathbb{R}^{I \times J} \) can be folded into an order-\( d \) tensor \( A \in \mathbb{R}^{I_1 \times \cdots \times I_d} \) where \( IJ = \prod_n I_n \). We can also fold \( W \) to an order-2\( d \) tensor \( A \in \mathbb{R}^{I_1 \times \cdots \times I_d \times J_1 \times \cdots \times J_d} \) such that \( w_{ij} = a_{i_1 \cdots i_d j_1 \cdots j_d} \). While a convolution filter is already a tensor, we can reshape it to a higher-order tensor with reduced mode sizes.

- **Low-rank tensor compression.** After folding a weight matrix or convolution filter into a higher-order tensor \( A \), one can employ low-rank tensor compression to reduce the number of parameters. Either the CP, Tucker, TT or TTM factorization can be applied [19, 20, 21, 21].

Assume that \( \Phi_k \) includes all low-rank tensor factors required to represent \( W_k \). Considering the dependence of \( W_k \) on \( \Phi_k \), we can now write (2.13) as
\[ (2.14) \quad y = h(x \mid \{W_k(\Phi_k), b_k\}_{k=1}^L) = f(x \mid \Psi), \] with \( \Psi = \{\Phi_k, b_k\}_{k=1}^L \).

Please note the following:

- The tensor factors in \( \Phi_k \) depend on the tensor format we choose. For instance, in CP format, \( \Phi_k \) only includes \( d \) matrix factors; in a Tucker format, \( \Phi_k \) includes \( d \) factor matrices and a small-size order-\( d \) core tensor as shown in (2.4); when the TT or TTM format is used, \( \Phi_k \) includes \( d \) order-3 or order-4 TT cores shown in (2.5) and (2.7) respectively. Different tensor formats can lead to different compression ratios.

- The number of variables in each \( \Phi_k \) depends on the tensor ranks used in the compression. A higher tensor rank leads to higher expressive power but a lower compression ratio. In existing approaches, it is hard to select a proper tensor rank \textit{a-priori}. We hope to address this issue in this paper.

- \( \Psi \) include all tensor factors and bias vectors of all layers in a tensorized neural network. The number of variables in \( \Psi \) is often orders-of-magnitude fewer than that in the original model (2.13).

- The function \( f \) includes the process of mapping \( \Phi_k \) back to its associated weight matrix \( W_k \) in addition to the forward evaluation of function \( h \).

Two current approaches exist to produce low-rank, tensor-compressed neural networks.

- **High training costs.** The network \( h \) is trained in uncompressed format so the training device the must support the high memory, compute, and energy requirements of the large network.
• **Lower accuracy.** After the initial training, the weights \( \{ W_k \}_{k=1}^{L} \) must be decomposed to low-rank tensor format [19, 20]. The tensor factorization followed by fine-tuning may greatly reduce model accuracy, especially when the weights produced by uncompressed training possess no hidden low-rank structure.

The second approach is fixed-rank tensorized training. In this approach the user pre-specifies the tensor rank and trains low-rank tensor factors of weight parameters. This approach avoids the compute and memory requirements of uncompressed training but requires that the user manually select a good rank *a-priori*. This approach usually requires multiple training runs to select the rank. In addition a user-specified rank may achieve suboptimal compression.

**3. Bayesian Low-Rank Tensorized Model.** In this work, we plan to develop a tensorized training method that can automatically determine the tensor ranks in the training process. This method requires only one training run and avoids the high cost of uncompressed training.

It is hard to determine tensor ranks in the training process. Exactly determining the tensor rank is NP-hard [44, 63]. Unlike matrix factorization which often uses the nuclear norm as a convex surrogate of matrix rank [68], there are no rigorous surrogate models for tensor rank. Existing heuristic surrogates [55, 56, 57] only work for small-size tensors due to the memory-consuming tensor unfolding operations. In this paper we will develop a Bayesian approach for rank determination in tensorized neural networks. Bayesian methods have been employed for tensor completion and factorization [45, 46, 59], where the observed data is a linear function of tensor elements. However, existing Bayesian tensor solvers do not work for tensorized neural networks due to the nonlinear forward model and large number of unknown variables.

**3.1. High-Level Bayesian Formulation.** We first describe a general-purpose Bayesian model for training low-rank tensorized neural networks. For notational convenience we assume that our neural network model \( f \) has one nonlinear layer, and that its weight matrix \( \mathbf{W} \) is folded to a single tensor \( \mathbf{A} \). Extending our method to general multi-layer cases with multiple tensors is straightforward, and we will report numerical results on general multi-layer models in Section 5.

Given a training data set \( \mathcal{D} \), our goal is determine the unknown low-rank factors \( \Phi \) for \( \mathbf{A} \), the associated tensor ranks and bias vector \( \mathbf{b} \). We introduce some hyper parameters \( \Lambda \) to control the tensor ranks and model complexity in our Bayesian model. Our posterior distribution is

\[
(3.1) \quad p(\Psi, \Lambda|\mathcal{D}) = \frac{p(\mathcal{D} | \Psi)p(\Psi, \Lambda)}{p(\mathcal{D})}, \text{ with } \Psi = \{ \Phi, \mathbf{b} \}.
\]

Here \( p(\mathcal{D} | \Psi) \) is the model likelihood, \( p(\Psi, \Lambda) \) is the joint prior and \( p(\mathcal{D}) \) is the model evidence defined by

\[
(3.2) \quad p(\mathcal{D}) = \int_{\Psi, \Lambda} p(\mathcal{D} | \Psi)p(\Psi, \Lambda)d\Psi d\Lambda.
\]

The likelihood and joint prior are specified below:

• **Likelihood function:** \( p(\mathcal{D} | \Psi) \) and data \( \mathcal{D} \) are determined by a forward propagation model. Our proposed method is suitable for any differentiable likelihood function.

This manuscript is for review purposes only.
In this work we focus on neural network classification problems with a multinomial likelihood function. Let \((x, y) \in D\) be a training sample where \(x\) is the neural network input and \(y\) is the associated true label. The likelihood is

\[
p(D|\Psi) = \prod_{(x, y) \in D} \text{Mult}(y|f(x|\Psi)).
\]

Here \(\text{Mult}\) is the multinomial distribution probability density function, and \(f\) is the forward propagation model as specified in (2.14) which is conditioned on the given low-rank tensor factors and bias vectors.

**Joint Prior:** We place an independent prior over the low-rank tensor factors and the bias term. We choose a weak normal prior for the bias term:

\[
p(\Psi, \Lambda) = p(b)p(\Phi, \Lambda), \quad p(b) = \prod_i N(b_i|0, \sigma^2_0).
\]

Here \(p(\Phi, \Lambda)\) is the joint prior for the low-rank tensor factors \(\Phi\) and the rank-controlling hyper parameter \(\Lambda\). The design of \(p(\Phi, \Lambda)\) is non-trivial and depends on the tensor format we choose for the model, which will be explained in Section 3.2 & 3.3.

### 3.2. Tensor Factor Priors

Proper priors should be chosen for the tensor factors in order to automatically enforce low-rank compression in the training process. The low-rank tensor factors in \(\Phi\) and their influence on the forward model \(h\) depends on the tensor format we choose. Therefore, we will specify the joint prior \(p(\Phi, \Lambda)\) for the four tensor formats described in Section 2.1: CP, Tucker, TT and TTM.

Firstly we specify the general form of \(p(\Phi, \Lambda)\).

- For the CP format, we initialize each factor \(U^{(n)}\) as a matrix with \(R\) columns. Assume that \(R\) is larger than the actual rank \(r\), and all factors shrink to \(r\) columns in the training process. Because all factors have the same maximum rank (number of columns), we use a single vector \(\Lambda = \lambda \in \mathbb{R}^R\) to control the rank.
- In contrast, the tensor rank in Tucker, TT or TTM format is a vector, and the rank associated with each mode can be different. Therefore, for an order-\(d\) tensor, we require a collection of vectors \(\Lambda = \{\lambda^{(n)}\}_{n=1}^d\) to control the ranks of each mode individually. Here \(\lambda^{(n)} \in \mathbb{R}^{R_n}\), and the “maximum rank” \(R_n\) exceeds \(r_n\) (the actual rank of mode \(n\)).

As a result, we introduce the general form

\[
p(\Phi, \Lambda) = \begin{cases} p(\Phi|\lambda)p(\lambda) & \text{for CP format} \\ p(\Phi|\{\lambda^{(n)}\}) \prod_{n=1}^d p(\lambda^{(n)}) & \text{for Tucker, TT & TTM formats} \end{cases}
\]

where the prior distribution(s) on \(\lambda\) or \(\{\lambda^{(n)}\}_{n=1}^d\) enforce(s) rank reduction.

Next we specify the tensor factor priors \(p(\Phi|\lambda)\) or \(p(\Phi|\{\lambda^{(n)}\})\) for each tensor format, and we defer the prior on \(\lambda\) and \(\{\lambda^{(n)}\}_{n=1}^d\) to Section 3.3.

- **CP Format:** The tensor factors in a CP factorization are \(d\) matrices, therefore \(\Phi = \{U^{(n)}\}_{n=1}^d\). We assign a Gaussian prior with controllable variance to each element of
(a) For the CP prior, if one element of $\lambda$ is small, one column is removed from every factor matrix. (b) For the Tucker prior, if one element of $\lambda^{(n)}$ is small then one column of $U^{(n)}$ shrinks to zero. (c) For the TT prior, if one element of $\lambda^{(n)}$ is small then one slice of $G^{(n)}$ shrinks to zero. The columns/slices to be removed are marked in white.

Each factor matrix $U^{(n)}$:

$$p(\Phi, \Lambda) = p(\lambda) \prod_n p\left(U^{(n)} | \lambda^{(n)}\right), \quad p\left(U^{(n)} | \lambda\right) = \prod_{i,j} \mathcal{N}\left(u_{ij}^{(n)} | 0, \lambda_j\right).$$

Here $u_{ij}^{(n)}$ is the $(i,j)$-th element of $U^{(n)}$. Each entry of $\lambda$ controls one column of each factor matrix. If a single entry $\lambda_j$ approaches zero, then the prior mean and prior variance of $u_{ij}^{(n)}$ are both close to zero for all row indices $i \in [1, I_n]$ and mode indices $n \in [1, d]$. This encourages the whole $j$-th column of $U^{(n)}$ to shrink to zero, leading to a rank reduction. The vector $\lambda$ is shared across all modes, therefore it will shrink the same column of all CP factor matrices $\{U^{(n)}\}_{n=1}^d$ simultaneously, as shown in Fig. 3 (a).

**Tucker Format**: A Tucker factorization includes a core tensor and $d$ factor matrices, therefore $\Phi = \{G, \{U^{(n)}\}_{n=1}^d\}$. Similar to the CP model, each factor matrix $U^{(n)}$ is assigned a Gaussian distribution with controllable variances. Unlike the CP model, a Tucker model has $d$ separate rank parameters $(r_1, \ldots, r_d)$ to determine, one per factor matrix as shown in Fig. 3 (b). Furthermore, the factor matrices and core tensor are handled separately. Therefore, we propose the following prior distributions:

$$p(\Phi, \Lambda) = p(G) \prod_n p\left(U^{(n)} | \lambda^{(n)}\right) p\left(\lambda^{(n)}\right), \quad p\left(U^{(n)} | \lambda^{(n)}\right) = \prod_{i,j} \mathcal{N}\left(u_{ij}^{(n)} | 0, \lambda_{ij}^{(n)}\right).$$

We use $d$ independent rank controlling vectors $\{\lambda^{(n)}\}_{n=1}^d$ to control the prior variances of different factor matrices separately. The $j$-th element of $\lambda^{(n)}$ controls the $j$-th column of factor matrix $U^{(n)}$. Therefore $\lambda^{(n)}$ controls $r_n$, the $n$-th entry of the Tucker rank. We place a weak normal prior over the entries of the core tensor $G$:

$$p(G) = \prod_{i_1, \ldots, i_d} \mathcal{N}(g_{i_1 \ldots i_d} | 0, \sigma_0).$$
We make this choice to simplify parameter inference compared to the alternative of placing low-rank priors on both of the core tensor and the factor matrices.

- **Tensor-Train (TT) Format:** A TT factorization uses $d$ order-3 TT cores to represent an order-$d$ tensor, therefore $\Phi = \{\mathcal{G}^{(n)}\}_{n=1}^d$. The TT format requires a more complicated prior because each low-rank factor $\mathcal{G}^{(n)} \in \mathbb{R}^{r_{n-1} \times I_n \times r_n}$ depends on two rank parameters $r_{n-1}$ and $r_n$. In order to automatically determine the TT rank, we choose $R_n > r_n$, and initialize the $n$-th TT core with size $R_{n-1} \times I_n \times R_n$. The prior density of all TT cores are given as

$$p(\Phi, \Lambda) = p\left(\mathcal{G}^{(d)}|\lambda^{(d-1)}\right) \prod_{1 \leq n \leq d-1} p\left(\mathcal{G}^{(n)}|\lambda^{(n)}\right) p\left(\lambda^{(n)}\right),$$

(3.8)

$$p\left(\mathcal{G}^{(n)}|\lambda^{(n)}\right) = \prod_{i,j,k} \mathcal{N}\left(\theta_{ijk}^{(n)} | 0, \lambda_i^{(n)}\right) \text{ for } n \in [1, d-1],$$

$$p\left(\mathcal{G}^{(d)}|\lambda^{(d-1)}\right) = \prod_{i,j,k} \mathcal{N}\left(\theta_{ijk}^{(d)} | 0, \lambda_i^{(d-1)}\right).$$

We introduce a vector $\lambda^{(n)} \in \mathbb{R}^{R_n}$ to directly control the actual rank $r_n$ for mode 1 to $d-1$. As shown in Fig. 3 (c), the $k$-th element of $\lambda^{(n)}$ (i.e., $\lambda_k^{(n)}$) controls the prior variance of a slice $\mathcal{G}^{(n)}(:, :, k)$. If $\lambda_k^{(n)}$ is small, then a whole slice of $\mathcal{G}^{(n)}$ is close to zero, leading to a rank reduction in the $n$-th mode. The only rank parameter that controls two separate cores is $\lambda^{(d-1)}$. This prevents any rank parameters from overlapping and it simplifies posterior inference.

- **Tensor-Train Matrix (TTM) Format:** Similar to the TT format, a TTM decomposition also has $d$ core tensors, therefore $\Phi = \{\mathcal{G}^{(n)}\}_{n=1}^d$. The only difference is that each $\mathcal{G}^{(n)}$ is an order-4 tensor, which is initialized with a size $R_{n-1} \times I_n \times J_n \times R_n$ in our Bayesian model. The prior for the TTM low-rank factors is

$$p(\Phi, \Lambda) = p\left(\mathcal{G}^{(d)}|\lambda^{(d-1)}\right) \prod_{1 \leq n \leq d-1} p\left(\mathcal{G}^{(n)}|\lambda^{(n)}\right) p\left(\lambda^{(n)}\right),$$

(3.9)

$$p\left(\mathcal{G}^{(n)}|\lambda^{(n)}\right) = \prod_{i,j,k,l} \mathcal{N}\left(\theta_{ijkl}^{(n)} | 0, \lambda_i^{(n)}\right), \text{ for } n \in [1, d-1],$$

$$p\left(\mathcal{G}^{(d)}|\lambda^{(d-1)}\right) = \prod_{i,j,k,l} \mathcal{N}\left(\theta_{ijkl}^{(d)} | 0, \lambda_i^{(d-1)}\right).$$

This prior very similar to that of TT format. We use a vector parameter $\lambda^{(n)}$ to control the actual rank $r_n$ of the $n$-th mode for $n \in [1, d-1]$, and $\lambda^{(d-1)}$ is shared among $\mathcal{G}^{(d)}$ and $\mathcal{G}^{(d-1)}$.

### 3.3. Rank-Shrinking Hyper-Parameter Priors.

To complete the setup of the full Bayesian model (3.1), we still need to specify the prior of rank-control hyper parameters $\Lambda = \lambda$ (for CP) or $\Lambda = \{\lambda^{(n)}\}_{n=1}^d$ (for Tucker, TT and TTM). Note that small elements in $\lambda$ and $\lambda^{(n)}$ leads to rank reductions in the tensor models, therefore we choose two hyper-prior densities that place high probability near zero. We focus our notation in this subsection on the CP model for simplicity.

This manuscript is for review purposes only.
We consider two choices of prior on the hyper parameter $\lambda$: the Half-Cauchy with scale parameter $\eta$ and the improper Log-Uniform on $(0, \infty)$:

\begin{equation}
\begin{split}
p(\lambda) &= \prod_{i=1}^{R} p(\lambda_i), \quad \text{with } p(\lambda_i) = \begin{cases} 
HC(\sqrt{\lambda_i}|0, \eta) \text{ or } 
LU(\sqrt{\lambda_i}). 
\end{cases} 
\end{split}
\end{equation}

The improper Log-uniform distribution has a fatter tail than the Half-Cauchy distribution and is parameter-free. We illustrate the probability density functions of both densities in Fig. 5a. The Half-Cauchy scaling parameter $\eta$ offers the advantage that the user can adjust $\eta > 0$ to tune the tradeoff between accuracy and rank-sparsity but requires additional tuning. Decreasing the magnitude of $\eta$ increases rank-sparsity. Both the Half-Cauchy density function and the Log-Uniform density function place high probability in regions where the input is close to zero. The parameter $\lambda$ controls the prior variance of the tensor factors in $\Phi$, all of which have prior mean zero. Therefore the prior density encodes a prior belief that the tensor rank is low. Since each element of $\lambda$ controls an entire column of every tensor factor, the prior density encourages structured rank shrinkage. We provide the Bayesian graphical models for each low-rank tensor format in Fig. 4 and a visual description of our low-rank tensor prior in Fig. 5.

In Fig 5 we demonstrate how our prior induces rank-sparsity. We consider the prior on a single CP rank parameter $\lambda_j$ and the marginal prior on a single CP factor matrix entry $u_{ij}^{(n)}$. 

---

This manuscript is for review purposes only.
Figure 5: (a) Comparison of the probability density functions of the Log-Uniform and Half-Cauchy hyperprior on $\lambda_j$. Several values of the Half-Cauchy scale parameter $\eta$ are given. (b) Comparison of the probability density functions of the corresponding marginal prior on the low-rank tensor factor entry $u_{ij}^{(n)}$. but the observations made in this section apply to entries in all tensor formats. In Fig 5a we plot the prior density on the rank parameter $\lambda_j$ and in Fig. 5b we plot the corresponding marginal prior on $u_{ij}^{(n)}$. In all cases, the prior places high probability on values of $\lambda_j$ that are close to zero. Both the Log-Uniform and Half-Cauchy hyper-prior encourage rank-shrinkage and the marginal priors illustrated in Fig 5b illustrate how those shrinkage effects differ. The flat tail and sharp peak of the marginal prior induced by the Log-Uniform rank hyper-prior leads to strong shrinkage of small values of $u_{ij}^{(n)}$ towards 0 but permits medium values to escape the “gravitational pull” around 0 [69]. In comparison, the marginal Horseshoe prior induced by the Half-Cauchy hyper-prior exerts a weaker shrinkage effect at small values of $u_{ij}^{(n)}$ but a stronger shrinkage effect on larger values.

4. Scalable Parameter Inference. Now we discuss how to estimate the resulting posterior density (3.1). We develop an efficient tensorized Bayesian inference approach by improving Stochastic Variational Inference (SVI) [52]. Mainstream inference methods such as fast MCMC based on stochastic dynamics [49,50] and Stein variational gradient descent [51] require storing multiple copies of the entire neural network for both training and inference. This increases both memory and computing requirements by orders of magnitude [50] and is unsuitable for large-scale problems or for application scenarios with constrained computing resources. The mean-field variational inference in [45,46] does not require storing multiple copies of model parameters, but its closed-form updates in the expectation-maximization steps only work for simple linear problems such as tensor completion and factorization. We consider SVI [52] due to its superior computational and memory efficiency. However, directly applying SVI to our tensorized training can cause numerical failures. Therefore, we will develop a customized SVI solver that is suitable for our Bayesian tensorized neural networks.
In this following, we firstly review the SVI method [52] for general Bayesian inference. Next, we explain some obstacles to a straightforward implementation in our problem setting. Then we propose a suitable approximate posterior to model the true posterior of rank-controlling parameters. Based on this approximation, we develop an analytical/numerical hybrid iteration method to overcome the challenges of SVI in our problem setting. Finally, we present the complete algorithm flow and highlight some implementation issues.

4.1. Review of Stochastic Variational Inference (SVI). SVI [52] is a popular method for scalable Bayesian machine learning. Let $\theta$ be the parameters to infer and let $q(\theta)$ be the approximating distribution to the target posterior distribution $p(\theta|D) \propto p(D|\theta)p(\theta)$.

SVI is formulated as an optimization problem where the loss function is the KL divergence and the goal is to find the best approximating density $q^*$ among a parameterized class of densities $\mathcal{P}$:

$$q^*(\theta) = \arg\min_{q(\theta) \in \mathcal{P}} \text{KL}(q(\theta)||p(\theta|D)),$$

where $\text{KL}(q(\theta)||p(\theta|D)) = \mathbb{E}_{q(\theta)} \left[ \log q(\theta) - \log p(D|\theta) - \log p(\theta) \right]$. (4.1)

We rearrange the KL divergence to illustrate the requirements on the approximating distribution $q$:

$$\text{KL}(q(\theta)||p(\theta|D)) = \mathbb{E}_{q(\theta)} \left[ \log q(\theta) - \log p(D|\theta) - \log p(\theta) \right] + \text{const.} = -\mathbb{E}_{q(\theta)} \left[ \log p(D|\theta) \right] + \text{KL}(q(\theta)||p(\theta)) + \text{const.}.$$ (4.2)

The first line follows from the Bayes rule and the second from the definition of the KL divergence. Therefore the objective function is a combination of the log-likelihood (model fit) and the divergence from the approximate posterior to the prior (low-rank). To approximate the log-likelihood one samples from the variational distribution $q$. The KL-divergence is either approximated via sampling or evaluated in a closed form depending on the model specification. The form in Equation (4.2) requires the evaluation of the full-data model likelihood. If the data is large the full-data likelihood $p(D|\theta)$ is intractable, so we approximate the likelihood by subsampling a minibatch $\mathcal{M} \subset D$.

4.2. Challenges in Training Bayesian Tensorized Neural Networks. Now we explain the challenges of directly applying SVI to train our Bayesian tensorized neural network model. As an example, we focus our notation on the CP model parameters

$$\theta = \{\Phi, \Lambda\} = \{\{U^{(n)}\}_{n=1}^d, \lambda\}$$ (4.3)

with one layer for clarity. The extension to the other three tensor models and to multi-layer networks is straightforward. We ignore the bias term $b$ which is assigned a factored normal variational posterior and follows well-studied update rules [70].

In variational Bayesian inference, it is a common practice to simplify a posterior density in order to meet computational constraints. In our problem setting, we firstly use the mean-field approximation [71] to achieve a tractable optimization:

$$q\left(\{U^{(n)}\}, \lambda\right) = q\left(\{U^{(n)}\}\right) q(\lambda).$$ (4.4)
We further model the posterior of the tensor factors with a normal distribution

\[
q(\{U^{(n)}\}) = \prod_{n=1}^{d} q(U^{(n)}) = \prod_{i,j} \mathcal{N}(u_{ij}^{(n)} | \overline{u}_{ij}^{(n)}, \Sigma_{ij}^{(n)^2}),
\]

where \(u_{ij}^{(n)}\) and \(\Sigma_{ij}^{(n)}\) are the \((i,j)\)-th elements of the unknown posterior mean \(\overline{U}^{(n)}\) and posterior standard deviation \(\Sigma^{(n)}\) to be inferred, respectively.

Now we discuss the challenges in learning the variational posterior distribution. We modify Eq. (4.2) to obtain our objective function:

\[
L(q) = -E_{q(\{U^{(n)}\}, \lambda)} \log p(D|\{U^{(n)}\}) + KL(q(\{U^{(n)}\}, \lambda) || p(\{U^{(n)}\}, \lambda)).
\]

In linear problems (i.e. tensor completion) the popular mean-field inference approach cycles through closed-form updates to the parameters of the variational distribution to select the optimal tensor factor parameters [45, 46]. Unfortunately, these closed-form updates do not work in our problem setting due to the nonlinear tensorized forward model. As a result, we need to employ gradient-based iterations in SVI to update the tensor factor parameters.

In the SVI framework, the expected log-likelihood in Equation (4.6) must be approximated by sampling the variational distribution \(q\). Therefore, various approaches only differ in how they interact with the KL divergence term in Equation (4.6). The first standard approach is to select a variational distribution \(q(\{U^{(n)}\}, \lambda)\) for which the KL divergence in Equation (4.6) can be obtained in a closed form. The second standard approach is to approximate the KL divergence term by sampling from the variational posterior. In practice, two challenges prevent us from applying these standard SVI approaches:

- **Challenge 1: Closed-form objectives require multiple training runs:** Variational distributions \(q\) that permit a closed-form approximation of the KL divergence require additional hyperparameters. Existing distributions that enable a closed-form KL divergence require a hierarchical Bayesian parameterization of the rank parameter \(\lambda\) [72, 73]. Such hierarchical Bayesian parameterizations require up to five additional hyperparameters for the new random variables in the parametrization [72]. Additional hyperparameters would require additional tuning runs and remove the benefits of one-shot tensorized training. Therefore, we avoid this option.

- **Challenge 2: Sampling-based approximation increases gradient variance:** Sampling-based approximation of the KL divergence leads to gradient instability during rank shrinkage. The main challenge is that the gradient variance with respect to the low-rank tensor factor parameters is proportional to the variance of \(1/\lambda\). In practical implementations, the gradient variance may explode during rank-shrinkage as \(\lambda\) approaches 0, so sampling \(\lambda\) is not feasible.

We provide more technical details about the second challenge. We consider the gradient of the objective function in Equation 4.6 with respect to the parameters \(u_{ij}^{(n)}\) and \(\Sigma_{ij}^{(n)}\). First we observe that the KL divergence satisfies the proportionality relation

\[
KL\left(\mathcal{N}\left(u_{ij}^{(n)}, \Sigma_{ij}^{(n)^2}\right) || \mathcal{N}(0, \lambda_{ij})\right) \propto \frac{u_{ij}^{(n)^2} + \Sigma_{ij}^{(n)^2}}{\lambda_{ij}}.
\]
Figure 6: The gradient variance of a single low-rank tensor factor parameter. Sampling the rank parameter $\lambda$ leads to high-variance gradients, while our proposed delta approximation of hyper parameters can reduce the gradient variance significantly (see Section 4.3 and Section 4.4).

Let $\phi \in \{u_{ij}^{(n)}, \Sigma_{ij}^{(n)}\}$ represent either parameter of the posterior distribution of $u_{ij}^{(n)}$. Then sampling $\lambda$ yields a gradient variance

$$\mathbb{V} \left[ \nabla_{\phi} KL \left( \mathcal{N} \left( u_{ij}^{(n)}, \Sigma_{ij}^{(n)} \right) \| \mathcal{N} (0, \lambda_j) \right) \right] \propto \frac{1}{\lambda_j}.$$  

The goal of our low-rank prior is to shrink many $\lambda_j$’s to 0 in the training process. If the distribution of $\lambda_j$ is non-degenerate, even small uncertainties in the value of $\lambda_j$ will lead to large variance in Equation (4.8) as the posterior probability of $\lambda_j$ concentrates around 0. This implies that a rank shrinkage can cause high-variance gradients which in turn may increase the magnitude of factor matrix parameters. In Figure 6 we illustrate how sampling $\lambda$ dramatically increases the gradient variance of the tensor factor parameters. Therefore, directly employing a sampling-based gradient estimation of the KL divergence for nonlinear rank determination is not a viable option either.

### 4.3. Simplified Posterior for Rank-Controlling Hyper Parameters.

In order to avoid the above gradient variance explosion, we propose a deterministic approximation to the rank-controlling hyper parameter $\lambda$:

$$q(\lambda) = \delta_{\overline{\lambda}}(\lambda)$$

where $\delta$ is a Delta function and $\overline{\lambda}$ is the posterior mean of $\lambda$. This delta approximation was used for empirical partially Bayes estimation in [74]. This approximation admits closed-form updates to the following sub-problem when the factor matrices are fixed:

$$\arg \min_{\overline{\lambda}_k} KL \left( q \left( \{U^{(n)}\}, \lambda \right) || p(\{U^{(n)}\}, \lambda) \right).$$

Now we explain how to obtain an analytical update for $\overline{\lambda}_k$. We derive the closed-form updates to a single rank parameter $\overline{\lambda}_k$ for the CP model. The results associated with other
Here we have used the notations
\begin{align}
\lambda_k & \leftarrow \frac{M}{D+1}.
\end{align}

Finally, enforcing the gradient (4.12) to be zero yields a closed-form update:
\begin{align}
\lambda_k^* = \frac{M}{D+1}.
\end{align}

Next, we consider a log-uniform rank prior \( p(\lambda) \) and take the derivative of the KL divergence with respect to \( \lambda_k \). This yields
\begin{align}
\frac{\partial}{\partial \lambda_k} \text{KL} \left( q \left( \{U^{(n)}\}, \lambda \right) \| p(\{U^{(n)}\}, \lambda) \right) \propto \sum_{1 \leq n \leq d, 1 \leq i \leq I_n} \left( \frac{u_{ik}^{(n)^2} + \Sigma_{ik}^{(n)^2}}{2\lambda_k} \right) + \frac{1}{2\lambda_k}.
\end{align}

Finally, enforcing the gradient (4.12) to be zero yields a closed-form update:
\begin{align}
\lambda_k^* = \frac{M}{D+1}.
\end{align}

Here we have used the notations
\begin{align}
D = \sum_n I_n, \quad M = \sum_{1 \leq n \leq d, 1 \leq i \leq I_n} u_{ik}^{(n)^2} + \Sigma_{ik}^{(n)^2}.
\end{align}

The number of entries controlled by \( \lambda_j \) is \( D \), and \( M \) is their combined magnitude and variance.

In the case of the Half-Cauchy prior with scale parameter \( \eta \), the update is
\begin{align}
\lambda_k^* = \frac{M - \eta^2 D + \sqrt{M^2 + (2D + 8)\eta^2 M + \eta^4 D^2}}{2D + 2}.
\end{align}

Decreasing the magnitude of the scale parameter \( \eta \) decreases the magnitude of the update of \( \lambda_k \), thereby increasing rank-sparsity.

### 4.4. Analytical/Numerical Hybrid Parameter Update in SVI

With the proposed Delta posterior approximation for \( \lambda \), now we can implement SVI for our tensorized neural network training with an analytical/numerical hybrid parameter update rule. Specifically, in every iteration of SVI, we use a half step of gradient-based update for the tensor factors in \( \Phi \) and a half step of closed-form update for the the rank-control hyper parameters \( \lambda \). We apply the reparametrization trick [75]
\begin{align}
\frac{u_{ij}^{(n)}}{\lambda_k^{*}} = \frac{u_{ij}^{(n)}}{\lambda_k^{*}} + z\Sigma_{ij}^{(n)}, \quad z \sim \mathcal{N}(0,1).
\end{align}

to sample from the tensor factor distributions.

\this manuscript is for review purposes only.
• **Half Step 1:** Gradient Update for tensor factors: We sample the low-rank tensor factors $\Phi$ and update all parameters of the tensor factor variational distributions using gradient descent on the loss $L(q)$ of Eq. (4.6) with a learning rate $\alpha$:

$$\Phi \leftarrow \Phi + \alpha \nabla \Phi L(q).$$

(4.17)

In the specific case of the CP model the gradients for the posterior variance and mean of the factor matrices are given by

$$\nabla_{\Sigma_{ij}} L(q) = -z \nabla_{u_{ij}} \log p(D|\{U_{ij}\}) - \frac{1}{\Sigma_{ij}} + \frac{\Sigma_{ij}}{\lambda_j} \nabla_{u_{ij}} L(q) = -\nabla_{u_{ij}} \log p(D|\{U_{ij}\}) + \frac{u_{ij}}{\lambda_j}.$$  

(4.18)

Note that $z$ is the random variable sampled during the forward pass due to the reparameterization in Eq. (4.16) and the gradients with respect the log-likelihood are computed using standard automatic differentiation. We describe the gradients for the other three tensor formats in the appendix.

• **Half Step 2:** Incremental closed-form update for $\lambda$: We analytically update the rank-controlling parameters $\lambda$ based on the results in (4.13) and (4.15). We found empirically that incremental updates to the rank-control parameters, rather than direct assignment of the minimizing values $\lambda_k^*$ from (4.13) or (4.15), led to better performance. Therefore we adopt an incremental update strategy with learning rate $\gamma$ for the rank parameter updates:

$$\lambda_k \leftarrow \gamma \lambda_k^* + (1 - \gamma) \lambda_k.$$  

(4.19)

As shown in Figure 6, this proposed hybrid parameter update can greatly reduce the gradient variance of tensor factors.

4.5. Algorithm Flow and Implementation Issues. With the two half steps of parameter updates in Section 4.4, we are ready to employ SVI to train our rank-adaptive tensorized neural networks. The full description of our end-to-end tensorized training with rank determination is shown in Alg. 4.1. We iteratively repeat the hybrid parameter updates for a predetermined number of epochs $m$.

In the following, we discuss some important implementation issues.

**Warmup Schedule.** A general challenge in Bayesian tensor computation is that poor initializations can lead to excessive rank shrinkage and trivial rank-zero solutions. In tensor completion the SVD is used to generate high-quality initializations [45, 46]. For nonlinear tensorized neural networks we randomly initialize the factor matrices so the predictive accuracy is low and the KL divergence to the prior may dominate the local loss landscape around the initialization point. To avoid trivial rank-zero local optima early in the training process we re-weight the KL divergence from the variational approximation to the prior by a factor that linearly increases from 0 to 1 during the training process. Let $e_m$ be the number of
Algorithm 4.1 SVI-Based Tensorized Training with Rank Determination

**Input:** Factor learning rate $\alpha$, EM stepsize $\gamma$, rank cutoff $\epsilon$, warmup epochs $e_w$, total epochs $m$, tuning epochs $t$

for Epoch $e$ in $[1, \ldots, m]$ do

Assign $\beta$ according to Equation (4.20).

for each batch $B \subset D$ do

Update the low-rank factor distribution variational parameters as in Half Step 1, Equation (4.17).

Update the rank-control hyper-parameters as in Half Step 2, Equation (4.19).

end for

end for

Prune tensor ranks as described in Equation (4.22).

Warmup training epochs and $e$ be the current epoch. We re-weight the KL divergence from the variational approximation to the prior by a factor $\beta$ defined by

$$\beta = \min \left( 1, \frac{e}{e_w} \right),$$

and update the loss from Eq. (4.6) accordingly:

$$\mathcal{L}(q) = - \log \mathbb{E}_q(\{U^{(n)}\}, \lambda)p(D|\{U^{(n)}\}) + \beta \text{KL} \left( q \left( \{U^{(n)}\}, \lambda \right) || p(\{U^{(n)}\}, \lambda) \right).$$

Gradually increasing the weight of the KL divergence to the prior avoids early local optima in which all ranks shrink to zero. We have found empirically that $e_w = m/2$ is a good choice for the number of warmup steps.

**Rank Pruning.** After we run our Bayesian solver we truncate the ranks with variance $\lambda_k$ below a pre-specified threshold $\epsilon$. For example, for the CP format if $\lambda_k < \epsilon$ we assign

$$u_{ik}^{(n)} \leftarrow 0 \text{ for } 1 \leq n \leq d, 1 \leq i \leq I_n,$$

$$\Sigma_{ik}^{(n)} \leftarrow 0 \text{ for } 1 \leq n \leq d, 1 \leq i \leq I_n.$$

The associated $k$-th column of $U^{(n)}$ is removed, leading to a rank shrinkage and automatic model parameter reduction.

**5. Experiments.** We demonstrate the applications of our rank-adaptive tensorized end-to-end training method on several models containing fully connected layers, embedding layers and convolution layers. We also compare our method with several recent methods of tensorized neural networks by reporting the predictive accuracy on the test data set and the number of model parameters in the training and in the resulting model. Our method trains a Bayesian neural network, therefore we report the predictive accuracy of the posterior mean.

In order to evaluate and compare the performance, we implement the following methods in our experiments:

\[ \text{This manuscript is for review purposes only.} \]
Table 1: Summary of different training methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Tensorized model</th>
<th>End-to-end training</th>
<th>Automatic rank determination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR [41]</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TC-MR [20]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC-OR [20]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARD-LU (Proposed)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ARD-HC (Proposed)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Baseline**: this is a standard training method, where the model parameters are neither tensorized nor compressed. We use this method as a baseline for comparison of predictive accuracy and model size.

- **TC-MR [19,20]**: train and then compress with maximum ranks. In order to compare our method with the most popular tensorized neural network methods, we implement the method in [19, 20]. In this approach we train a non-tensorized and uncompressed neural network (equivalent to “baseline”), followed by a tensor decomposition and fine-tuning. For the DLRM model we fine-tune for one epoch. In all other experiments we fine-tune for 20 epochs. This approach requires that the user select the compression rank. Here we use the maximum rank ($R$ for CP or $R = [R_1, R_2, \cdots R_d]$ for other formats) used in our proposed Bayesian model.

- **TC-OR**: train and then compress with oracle rank ($r$ in CP or $r = [r_1, r_2, \cdots, r_d]$ for other formats). This post-training compression method follows the same procedures of TC-MR [19, 20], except that it uses the “oracle rank” discovered by our proposed rank determination method. In practice the “TC-OR” method would require a combinatorial rank search over a high-dimensional discrete space to discover the same rank as our method. Here we just assume that we know the oracle rank *a-priori* by luck in order to compare the performance of the post-training compression method with our proposed method when they have the same compressed model size.

- **FR**: Fixed-rank tensorized training. We implement the end-to-end tensorized training [41] with a tensor rank fixed *a priori*. It is hard to determine a rank properly in advance, and the model can be very inaccurate if the chosen rank parameter is too low. In the experiment, we train the low-rank tensor factors with the maximum rank that is chosen for our Bayesian model in order to achieve a good accuracy.

- **ARD-LU**: this is the first version of our proposed tensorized training method with automatic rank determination. In this version, we use the log-uniform prior in (3.12) for the rank-control hyper-parameters. All tensor factors are initialized with a maximum rank ($R$ for CP and $R = [R, \cdots, R_d]$ for other formats), and the actual ranks ($r$ for CP and $r = [r_1, \cdots, r_d]$ for other formats) are automatically determined by our training process.

- **ARD-HC**: this is the second version of our proposed training method. It differs from ARD-LU only in the hyper prior: in ARD-HC we use the half-Cauchy prior (3.11) for the rank-control hyper parameters.

Table 1 has summarized the features of various methods. Clearly, our proposed methods enjoy...
all of the listed advantages compared with other methods. We consider four (CP, Tucker, TT and TTM) low-rank tensor formats for each tensorized method. Therefore, our experiments involve the implementation of 21 specific methods in total (20 tensorized implementations plus one baseline method).

We do not compare our method against low-rank matrix compression methods [16,17,18], because the previous study [41] has already shown the significantly higher compression ratios of fixed-rank tensorized training (i.e., the FR method in our experiments) as compared to matrix compression approaches with a similar level of accuracy.

Remark 5.1. In our Bayesian training, every tensorized model parameter is equipped with two training variables (i.e., posterior mean and variance). Therefore the number of training variables is $2 \times$ that of the tensorized model parameter numbers. This parameter overhead in Bayesian training brings in the capability of uncertainty quantification in output prediction, which is important for safety-critical applications. Our Bayesian model also allows a point-wise maximum-a-posterior (MAP) training. In MAP training, the only additional parameters required are the rank-control parameters so the number of training variables is only slightly larger than the number of training variables in fixed-rank tensorized training.

5.1. Synthetic Example for Rank Determination. First we test the ability of our proposed method to infer the tensor rank of model parameters in a neural network. For each tensor format we construct a synthetic version of the MNIST dataset using a one-layer tensorized neural network (equivalent to tensorized logistic regression). The tensorized layer is fully connected and the fixed tensor rank is five for each tensor format: 5 for CP, [5, 5, 5] for Tucker and [1, 5, 5, 1] for TT/TTM. We use the rank-5 model to generate synthetic labels for the MNIST images. Then we train a set of low-rank tensorized models with a maximum rank of 10 on the synthetic dataset. For the CP, tensor-train, and Tucker formats we reshape the weight matrix $W \in \mathbb{R}^{784 \times 10}$ into a tensor of shape size $[28, 28, 10]$ (i.e., an order-3 tensor of size $28 \times 28 \times 10$). For the tensor-train matrix format we use the dimensions $[4, 7, 4], [7, 2, 5]$.

We plot the mean inferred ranks for our log-uniform (LU) and half-cauchy (HC) priors in Fig. 7. We observe that our model performs well across tensor formats. The actual CP rank is exactly recovered. The inferred ranks of Tucker, TT and TTM are close to but not equal...
Table 2: Tensorization settings for the MNIST example.

<table>
<thead>
<tr>
<th>Model</th>
<th>Layer 1 Dimensions</th>
<th>Layer 2 Dimensions</th>
<th>Max Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>$784 \times 512$</td>
<td>$512 \times 10$</td>
<td>NA</td>
</tr>
<tr>
<td>CP</td>
<td>$[28,28,16,32]$</td>
<td>$[32,16,10]$</td>
<td>50</td>
</tr>
<tr>
<td>Tucker</td>
<td>$[28,28,16,32]$</td>
<td>$[32,16,10]$</td>
<td>20</td>
</tr>
<tr>
<td>TT</td>
<td>$[28,28,16,32]$</td>
<td>$[32,16,10]$</td>
<td>20</td>
</tr>
<tr>
<td>TTM</td>
<td>$[4,7,4,7],[4,4,8,4]$</td>
<td>$[32,16],[2,5]$</td>
<td>20</td>
</tr>
</tbody>
</table>

(a) CP format  (b) TT format  (c) TTM format  (d) Tucker format

Figure 8: Inferred ranks for one run of the MNIST experiment using a log-uniform prior. The maximum rank is given by a dashed black line. The inferred ranks are given by colored bars.

to the exact values, because tensor ranks are not unique, which is a fundamental difference between matrices and tensors.

5.2. MNIST. Next we apply all methods to a neural network with two fully connected layers on the MNIST dataset with images of size $28 \times 28$. The first fully connected layer is size $784 \times 512$ and has a ReLU activation function. The second fully connected layer is size $512 \times 10$ with a softmax activation function. We list the specifications of tensorization in Table 2. We use a maximum rank of 20 for all models except the CP, for which we increase the maximum rank to 50.

We observe that in all cases our automatic rank determination can achieve the highest compression ratio in training. Our proposed automatic rank determination both improves accuracy and reduces parameter number in all tensor formats except the TT format which has slight accuracy loss but the highest compression ratio. We hypothesize that the automatic rank reduction can reduce over-fitting on the simple MNIST task. The TTM format is best-suited to fully connected layers, achieving the second-highest compression ratios and the second-best accuracy. In Fig. 8 we plot the rank determination output of a single training run using our log-uniform prior. We note that our algorithm discovers the actual ranks that are nearly impossible to determine via hand-tuning or combinatorial search (for example $[1,20,3,2,1]$ in the TTM model from a maximum rank of $[1,20,20,20,1]$, which may require up to 16,000 searches).

With the obtained Bayesian solution, we can quantify the uncertainty of our model and its output prediction as a by-product. Popular metrics for uncertainty measures include negative log-likelihood [76], expected calibration error [77], which measures model over-/under-confidence, and out-of-distribution input detection [78]. In Fig. 9, we show the classification
Figure 9: (a) A challenging MNIST image with true label “2”. (b) Mean and standard deviation of the CP ARD-LU model softmax outputs. (c) Marginal predictive density of the two most likely labels “2” (x-axis) and “7” (y-axis).

Table 3: Training results of the MNIST example.

<table>
<thead>
<tr>
<th>Tensor Format</th>
<th>Model</th>
<th>Training Parameter #</th>
<th>Final Parameter #</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>FR [41]</td>
<td>407,050 (47.2×)</td>
<td>407,050</td>
<td>97.52</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>8,622 (47.2×)</td>
<td>8,622 (47.2×)</td>
<td>97.32</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>47 (47.2×)</td>
<td>97.36</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>7,175 (56.7×)</td>
<td>97.98</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>17,344 (23.5×)</td>
<td>7,134 (57.1×)</td>
<td>97.98</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>171,762 (2.4×)</td>
<td>171,762 (2.4×)</td>
<td>97.93</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>100,758 (4.0×)</td>
<td>97.91</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>343,644 (1.18×)</td>
<td>98.30</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>343,644 (1.18×)</td>
<td>98.30</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>26,562 (13.9×)</td>
<td>26,562 (15.3×)</td>
<td>97.78</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>26,562 (15.3×)</td>
<td>97.43</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>4,224 (96.4×)</td>
<td>96.91</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>53,224 (7.65×)</td>
<td>96.28</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>53,224 (7.65×)</td>
<td>97.04</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>29,242 (13.9×)</td>
<td>29,242 (13.9×)</td>
<td>98.06</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>29,242 (13.9×)</td>
<td>97.47</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>58,564 (6.95×)</td>
<td>98.24</td>
</tr>
<tr>
<td></td>
<td>FR [41]</td>
<td>407,050 (1×)</td>
<td>58,564 (6.95×)</td>
<td>98.23</td>
</tr>
</tbody>
</table>

Note: the training parameters in ARD-LU and ARD-HC include posterior mean and variance, so the training parameter number is 2× of that in FR.

uncertainty of an image that is hard to recognize in practice. With the CP tensorized model trained from ARD-LU, we plot the mean and variance of the predicted softmax outputs in Fig. 9 (b). This plot clearly shows that this image looks like “2”, “3” and “7”, with the highest probability of being classified as “7”. Fig. 9 (c) further plots the marginal predictive density of the two most likely labels “2” and “7”.

This manuscript is for review purposes only.
Table 4: Tensorization settings for the NLP embedding table.

<table>
<thead>
<tr>
<th>Model</th>
<th>Embedding Dimensions</th>
<th>Max Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>25,000 × 256</td>
<td>NA</td>
</tr>
<tr>
<td>CP</td>
<td>[5,8,25,25,4,8,8]</td>
<td>50</td>
</tr>
<tr>
<td>Tucker</td>
<td>[25,25,40,16,16]</td>
<td>5</td>
</tr>
<tr>
<td>TT</td>
<td>[5,5,5,5,6,8]</td>
<td>20</td>
</tr>
<tr>
<td>TTM</td>
<td>[5,5,5,56,8,22,2,2,4,4]</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5: Training results on the NLP embedding table.

<table>
<thead>
<tr>
<th>Tensor Type</th>
<th>Model</th>
<th>Training Parameter #</th>
<th>Final model Parameter #</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>6,400,000</td>
<td>6,400,000</td>
<td>88.34</td>
</tr>
<tr>
<td>CP</td>
<td>FR [41]</td>
<td>8,276 (774×)</td>
<td>8,276 (774×)</td>
<td>87.44</td>
</tr>
<tr>
<td></td>
<td>TC-MR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,138 (1024×)</td>
<td>74.46</td>
</tr>
<tr>
<td></td>
<td>TC-OR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,138 (1024×)</td>
<td>87.61</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>16,602 (385×)</td>
<td>6,476 (998×)</td>
<td>85.74</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>16,602 (385×)</td>
<td>6,476 (998×)</td>
<td>85.74</td>
</tr>
<tr>
<td>Tucker</td>
<td>FR [41]</td>
<td>78,540 (81×)</td>
<td>78,540 (81×)</td>
<td>87.80</td>
</tr>
<tr>
<td></td>
<td>TC-MR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,192 (103×)</td>
<td>75.12</td>
</tr>
<tr>
<td></td>
<td>TC-OR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,192 (103×)</td>
<td>75.12</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>157,105 (40×)</td>
<td>61,920 (103×)</td>
<td>87.79</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>157,105 (40×)</td>
<td>61,920 (103×)</td>
<td>87.79</td>
</tr>
<tr>
<td>TT</td>
<td>FR [41]</td>
<td>28,260 (226×)</td>
<td>28,260 (226×)</td>
<td>85.6</td>
</tr>
<tr>
<td></td>
<td>TC-MR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,132 (1024×)</td>
<td>82.34</td>
</tr>
<tr>
<td></td>
<td>TC-OR [20]</td>
<td>6,400,000 (1×)</td>
<td>6,132 (1024×)</td>
<td>82.34</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>56,450 (113×)</td>
<td>19,363 (331×)</td>
<td>85.82</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>56,450 (113×)</td>
<td>19,363 (331×)</td>
<td>85.82</td>
</tr>
<tr>
<td>TTM</td>
<td>FR [41]</td>
<td>22,312 (287×)</td>
<td>22,312 (287×)</td>
<td>88.59</td>
</tr>
<tr>
<td></td>
<td>TC-MR [20]</td>
<td>6,400,000 (1×)</td>
<td>15,932 (402×)</td>
<td>83.79</td>
</tr>
<tr>
<td></td>
<td>TC-OR [20]</td>
<td>6,400,000 (1×)</td>
<td>15,932 (402×)</td>
<td>83.79</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>44,724 (143×)</td>
<td>14,275 (331×)</td>
<td>88.93</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>44,724 (143×)</td>
<td>14,275 (331×)</td>
<td>88.93</td>
</tr>
</tbody>
</table>

Note: the training parameters in ARD-LU and ARD-HC include posterior mean and variance of each tensorized model parameter.

5.3. Embedding Table for Natural Language Processing (NLP). We continue to validate our rank-adaptive tensorized training with a sentiment classification task from [43]. Like many NLP models, the first layer is a large embedding table. Embedding tables are a promising target for tensor compression because their required input dimension equals the number of unique tokens in the input dataset (i.e. number of vocabulary words, number of users). Tensor decomposition can enforce weight sharing and dramatically reduce the parameter count of these models. Recent work in tensorized neural networks has applied the TTM format to compress large embedding tables with a high ratio [43]. We replicate a sentiment classification model on the IMDB dataset from their work. The neural network model consists of an embedding table with dimension 25,000 × 256, two bidirectional LSTM layers with hidden...
Table 6: Tensorization settings for the DLRM embedding tables.

<table>
<thead>
<tr>
<th>Embedding Layer</th>
<th>Model</th>
<th>Embedding Dimensions</th>
<th>Max Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baseline</td>
<td>10, 131, 227 × 128</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>[200, 220, 250, 128]</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>Tucker</td>
<td>[200, 220, 250, 128]</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>TT</td>
<td>[200, 220, 250, 128]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TTM</td>
<td>[200, 220, 250], [4,4,8]</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Baseline</td>
<td>2, 202, 608 × 128</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>[125, 130, 136, 128]</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td>Tucker</td>
<td>[125, 130, 136, 128]</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>TT</td>
<td>[125, 130, 136, 128]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TTM</td>
<td>[125, 130, 136], [4,4,8]</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Baseline</td>
<td>8, 351, 593 × 128</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>[200, 200, 209, 128]</td>
<td>333</td>
</tr>
<tr>
<td></td>
<td>Tucker</td>
<td>[200, 200, 209, 128]</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>TT</td>
<td>[200, 200, 209, 128]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TTM</td>
<td>[200, 220, 250], [4,4,8]</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Baseline</td>
<td>5, 461, 306 × 128</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>[166, 175, 188, 128]</td>
<td>326</td>
</tr>
<tr>
<td></td>
<td>Tucker</td>
<td>[166, 175, 188, 128]</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>TT</td>
<td>[166, 175, 188, 128]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TTM</td>
<td>[166, 175, 188], [4,4,8]</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Baseline</td>
<td>7, 046, 547 × 128</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>[200, 200, 200]</td>
<td>335</td>
</tr>
<tr>
<td></td>
<td>Tucker</td>
<td>[200, 200, 200, 128]</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>TT</td>
<td>[200, 200, 200, 128]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TTM</td>
<td>[200, 200, 200], [4,4,8]</td>
<td>16</td>
</tr>
</tbody>
</table>

unit size 128, and a fully-connected layer with 256 hidden units. Following the benchmark model in [43] we do not tensorize these layers. Dropout masks are applied to the output of each layer except the last.

We test all four tensor formats and both tensor priors on the sentiment classification problem. The tensor dimensions and maximum ranks used to compress the embedding table are given in Table 4. We use a lower number of dimensions and a lower maximum rank for the Tucker model to avoid a large-size core tensor $G$.

The outcomes of our experiments are reported in Table 5. Compared with all other tensor approaches, our methods (ARD-LU and ARD-HC) have achieved the best compression ratio for all tensor formats at little to no accuracy cost. The TTM format outperforms all other models (including the baseline uncompressed model) in terms of accuracy, though we note that the CP model performs well despite its extremely low parameter number.

5.4. Deep Learning Recommendation System (DLRM). We continue to use our proposed Bayesian tensorized method to train the benchmark Deep Learning Recommendation Model (DLRM) [2] released by Facebook. In the DLRM model, embedding tables are used to process categorical features, while continuous features are processed with a bottom multi-layer perceptron (MLP). Then, second-order interactions of different features are computed explicitly. The results are processed with a top MLP and fed into a sigmoid function in order
Table 7: Training results on the DLRM embedding tables.

<table>
<thead>
<tr>
<th>Tensor Type</th>
<th>Model</th>
<th>Training Parameter #</th>
<th>Final model Parameter #</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>4,248,739,968</td>
<td>4,248,739,968</td>
<td>78.75</td>
</tr>
<tr>
<td></td>
<td>FR [43]</td>
<td>1,141,597 (3,721×)</td>
<td>1,141,597 (3,721×)</td>
<td>78.60</td>
</tr>
<tr>
<td>CP</td>
<td>TC-MR [20]</td>
<td>4,248,739,968 (1×)</td>
<td>563,839 (7,555×)</td>
<td>74.92</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>2,284,844 (1860×)</td>
<td>563,839 (7,555×)</td>
<td>78.61</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>2,284,844 (1860×)</td>
<td>570,685 (7,444×)</td>
<td>78.57</td>
</tr>
<tr>
<td></td>
<td>FR [43]</td>
<td>1,131,212 (3,755×)</td>
<td>1,131,212 (3,755×)</td>
<td>78.60</td>
</tr>
<tr>
<td>Tucker</td>
<td>TC-MR [20]</td>
<td>4,248,739,968 (1×)</td>
<td>436,579 (9,731×)</td>
<td>78.67</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>2,262,852 (1,877×)</td>
<td>436,579 (9,731×)</td>
<td>78.64</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>2,262,852 (1,877×)</td>
<td>402,023 (10,568×)</td>
<td>78.62</td>
</tr>
<tr>
<td></td>
<td>FR [43]</td>
<td>1,135,752 (3,740×)</td>
<td>1,135,752 (3,740×)</td>
<td>78.68</td>
</tr>
<tr>
<td>TT</td>
<td>TC-MR [20]</td>
<td>4,248,739,968 (1×)</td>
<td>153,582 (27,664×)</td>
<td>78.45</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>2,271,864 (1870×)</td>
<td>153,582 (27,664×)</td>
<td>78.67</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>2,271,864 (1870×)</td>
<td>159,529 (26,633×)</td>
<td>78.63</td>
</tr>
<tr>
<td></td>
<td>FR [43]</td>
<td>1,130,048 (3759×)</td>
<td>1,130,048 (3759×)</td>
<td>78.73</td>
</tr>
<tr>
<td>TTM</td>
<td>TC-MR [20]</td>
<td>4,248,739,968 (1×)</td>
<td>199,504 (21,296×)</td>
<td>78.62</td>
</tr>
<tr>
<td></td>
<td>ARD-LU (Proposed)</td>
<td>2,260,256 (1879×)</td>
<td>199,504 (21,296×)</td>
<td>78.72</td>
</tr>
<tr>
<td></td>
<td>ARD-HC (Proposed)</td>
<td>2,260,256 (1879×)</td>
<td>163,976 (25,910×)</td>
<td>78.73</td>
</tr>
</tbody>
</table>

Note: the training parameters in ARD-LU and ARD-HC include posterior mean and variance of every tensorized model parameters, so the number of training variables is 2× of that in fixed-rank tensorized training (FR).

The whole model has over 4 billion training variables. In order to reduce the training variables, we tensorize the five largest embedding tables of the DLRM model as shown in Table 6. The outcomes of our experiments are reported in Table 7. We observe again that our proposed automatic rank reduction enables parameter reduction at little to no accuracy cost over fixed-rank tensorized training. Our approach outperforms the train-then-compress approach which requires expensive full-model training. Compared with the baseline full-size training, our method can significantly reduce the training variables and memory cost, leading to up to 27,664× (in TT format) parameter reduction in the training with little accuracy loss. Our approach also greatly increases the compression ratio over fixed-rank training at little to no accuracy cost, enabling up to 7× higher compression ratios even in the case of the highest-accuracy TTM model.

The train-then-compress approach can be expensive for this large-scale problem. Because the trained embedding tables are extremely large, compressing them in Tucker or CP format is computationally expensive and time-consuming. This challenge can be avoided in our end-to-end-training approaches because we do not need to explicitly form the embedding tables.

### 5.5. CIFAR Convolutional Model

Our final test model is a convolutional neural network taken from [21]. This model consists of six convolutional layers followed by three fully connected layers.
Table 8: Training results on the CNN model.

<table>
<thead>
<tr>
<th>Tensor Type</th>
<th>Model</th>
<th>Training Parameter #</th>
<th>Final Parameter #</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>13,942,602</td>
<td>13,942,602</td>
<td>90.36</td>
<td></td>
</tr>
<tr>
<td>FR [41]</td>
<td>652,748 (21.4×)</td>
<td>652,748 (21.4×)</td>
<td>90.13</td>
<td></td>
</tr>
<tr>
<td>TC-MR [20]</td>
<td>13,942,602 (1×)</td>
<td>568,412 (24.5×)</td>
<td>71.29</td>
<td></td>
</tr>
<tr>
<td>ARD-LU (Proposed)</td>
<td>1,308,418 (10.6×)</td>
<td>593,419 (23.5×)</td>
<td>90.08</td>
<td></td>
</tr>
<tr>
<td>ARD-HC (Proposed)</td>
<td>1,308,418 (10.6×)</td>
<td>593,419 (23.5×)</td>
<td>90.08</td>
<td></td>
</tr>
<tr>
<td>FR [41]</td>
<td>653,438 (21.3×)</td>
<td>653,438 (21.3×)</td>
<td>85.15</td>
<td></td>
</tr>
<tr>
<td>TC-MR [20]</td>
<td>13,942,602 (1×)</td>
<td>606,201 (23.0×)</td>
<td>84.86</td>
<td></td>
</tr>
<tr>
<td>ARD-LU (Proposed)</td>
<td>1,307,591 (10.7×)</td>
<td>606,201 (23.0×)</td>
<td>85.41</td>
<td></td>
</tr>
<tr>
<td>ARD-HC (Proposed)</td>
<td>1,307,591 (10.7×)</td>
<td>589,092 (23.7×)</td>
<td>85.86</td>
<td></td>
</tr>
<tr>
<td>FR [41]</td>
<td>653,438 (21.3×)</td>
<td>653,438 (21.3×)</td>
<td>85.15</td>
<td></td>
</tr>
<tr>
<td>TC-MR [20]</td>
<td>13,942,602 (1×)</td>
<td>649,328 (21.5×)</td>
<td>86.02</td>
<td></td>
</tr>
<tr>
<td>ARD-LU (Proposed)</td>
<td>1,299,106 (10.9×)</td>
<td>521,096 (26.8×)</td>
<td>85.92</td>
<td></td>
</tr>
<tr>
<td>ARD-HC (Proposed)</td>
<td>1,299,106 (10.9×)</td>
<td>521,096 (26.8×)</td>
<td>85.92</td>
<td></td>
</tr>
<tr>
<td>FR [41]</td>
<td>649,328 (21.5×)</td>
<td>649,328 (21.5×)</td>
<td>87.31</td>
<td></td>
</tr>
<tr>
<td>TC-MR [20]</td>
<td>13,942,602 (1×)</td>
<td>376,123 (37.1×)</td>
<td>85.42</td>
<td></td>
</tr>
<tr>
<td>ARD-LU (Proposed)</td>
<td>1,284,586 (10.9×)</td>
<td>598,693 (22.3×)</td>
<td>90.09</td>
<td></td>
</tr>
<tr>
<td>ARD-HC (Proposed)</td>
<td>1,284,586 (10.9×)</td>
<td>598,693 (22.3×)</td>
<td>90.09</td>
<td></td>
</tr>
</tbody>
</table>

Note: the training parameters in ARD-LU and ARD-HC include posterior mean and variance of every tensorized model parameters, so the number of training variables is 2× of that in fixed-rank tensorized training (FR).

Connected layers. We follow [21] and tensorize all layers except the first convolution and the last fully connected layer which together contain a small fraction of the total parameters. As before, we test all four tensor formats with our rank determination approach. The results of our method, the baseline model, and the train-and-then-compress approach are reported in Table 8.

We observe that our proposed method (ARD) leads to higher accuracy than the train-and-then-compress approach. Our automatic rank determination achieves parameter reduction with only slight accuracy reduction. The CP and TTM methods outperform Tucker and TT methods for this task in terms of accuracy. Previous studies [19, 21] have shown that the compression ratio on convolution layers are often much lower than on fully connected layers due to the small size of convolution filters. Nevertheless, our tensorized training with automatic rank determination always achieves the best compression performance.

6. Conclusion. This work has proposed a variational Bayesian method for one-shot end-to-end training of tensorized neural networks. Our work has addressed the fundamental challenge of automatic rank determination, which is important for training compact neural network models on many hardware platforms (especially on resource-constrained devices). The customized stochastic variational inference method developed in this paper enables us to train tensorized neural networks with billions of uncompressed model parameters. Our experiments
have demonstrated that the proposed end-to-end tensorized training can reduce the training variables by several orders of magnitude. Our proposed method has outperformed all existing tensor compression methods on the tested benchmarks in terms of both compression ratios and predictive accuracy.

**Acknowledgments.** We thank Kaiqi Zhang for pointing out an error in the original gradient update derivations.

**Appendix A. Gradient Updates.**

**Tucker:** In Tucker format the gradients take a similar form as in the CP format except the rank parameter is dimension-specific.

\[
\nabla \Sigma^{(n)} L(q) = -z \nabla u_{ij} \log p(D|G, \{U^{(n)}\}) - \frac{1}{\sum_{ij} \lambda_j^{(n)}} \Sigma^{(n)} + \sum_{ij} \lambda_j^{(n)}
\]

\[
\nabla u_{ij} L(q) = -\nabla u_{ij} \log p(D|G, \{U^{(n)}\}) + \frac{u_{ij}^{(n)}}{\lambda_j^{(n)}}
\]

**Tensor Train:** In TT-format the parameters are re-indexed to accommodate a third dimension in addition to the dimension-specific rank parameters referenced above.

\[
\nabla \Sigma^{(n)} L(q) = -z \nabla g_{ijk} \log p(D|G^{(n)}\{U^{(n)}\}) - \frac{1}{\sum_{ijk} \lambda_k} \Sigma^{(n)} + \sum_{ijk} \lambda_k
\]

\[
\nabla g_{ijk} L(q) = -\nabla g_{ijk} \log p(D|G^{(n)}\{G^{(n)}\}) + \frac{g_{ijk}^{(n)}}{\lambda_k^{(n)}}
\]

**Tensor Train Matrix:** Finally, for the TTM format we perform one additional re-indexing.

\[
\nabla \Sigma^{(n)} L(q) = -z \nabla g_{ijkl} \log p(D|G^{(n)}\{G^{(n)}\}) - \frac{1}{\sum_{ijkl} \lambda_l} \Sigma^{(n)} + \sum_{ijkl} \lambda_l
\]

\[
\nabla g_{ijkl} L(q) = -\nabla g_{ijkl} \log p(D|G^{(n)}\{G^{(n)}\}) + \frac{g_{ijkl}^{(n)}}{\lambda_l^{(n)}}
\]
REFERENCES


This manuscript is for review purposes only.
31


no. 4, pp. 736–748, 2016.


