Tensorizing Neural Networks

Alexander Novikov[1,4] **Dmitry Podoprikhin**[1] **Anton Osokin**[2] **Dmitry Vetrov**[1,3]
[1]Skolkovo Institute of Science and Technology, Moscow, Russia
[2]INRIA, SIERRA project-team, Paris, France
[3]National Research University Higher School of Economics, Moscow, Russia
[4]Institute of Numerical Mathematics of the Russian Academy of Sciences, Moscow, Russia
novikov@bayesgroup.ru podoprikhin.dmitry@gmail.com
anton.osokin@inria.fr vetrovd@yandex.ru

Abstract

Deep neural networks currently demonstrate state-of-the-art performance in several domains. At the same time, models of this class are very demanding in terms of computational resources. In particular, a large amount of memory is required by commonly used fully-connected layers, making it hard to use the models on low-end devices and stopping the further increase of the model size. In this paper we convert the dense weight matrices of the fully-connected layers to the Tensor Train [17] format such that the number of parameters is reduced by a huge factor and at the same time the expressive power of the layer is preserved. In particular, for the Very Deep VGG networks [21] we report the compression factor of the dense weight matrix of a fully-connected layer up to 200000 times leading to the compression factor of the whole network up to 7 times.

1 Introduction

Deep neural networks currently demonstrate state-of-the-art performance in many domains of large-scale machine learning, such as computer vision, speech recognition, text processing, etc. These advances have become possible because of algorithmic advances, large amounts of available data, and modern hardware. For example, convolutional neural networks (CNNs) [13, 21] show by a large margin superior performance on the task of image classification. These models have thousands of nodes and millions of learnable parameters and are trained using millions of images [19] on powerful Graphics Processing Units (GPUs).

The necessity of expensive hardware and long processing time are the factors that complicate the application of such models on conventional desktops and portable devices. Consequently, a large number of works tried to reduce both hardware requirements (e. g. memory demands) and running times (see Sec. 2).

In this paper we consider probably the most frequently used layer of the neural networks: the fully-connected layer. This layer consists in a linear transformation of a high-dimensional input signal to a high-dimensional output signal with a large dense matrix defining the transformation. For example, in modern CNNs the dimensions of the input and output signals of the fully-connected layers are of the order of thousands, bringing the number of parameters of the fully-connected layers up to millions.

We use a compact multiliniear format – Tensor-Train (TT-format) [17] – to represent the dense weight matrix of the fully-connected layers using few parameters while keeping enough flexibility to perform signal transformations. The resulting layer is compatible with the existing training algorithms for neural networks because all the derivatives required by the back-propagation algorithm [18] can be computed using the properties of the TT-format. We call the resulting layer a *TT-layer* and refer to a network with one or more TT-layers as *TensorNet*.

We apply our method to popular network architectures proposed for several datasets of different scales: MNIST [15], CIFAR-10 [12], ImageNet [13]. We experimentally show that the networks

with the TT-layers match the performance of their uncompressed counterparts but require up to $200\,000$ times less of parameters, decreasing the size of the whole network by a factor of 7.

The rest of the paper is organized as follows. We start with a review of the related work in Sec. 2. We introduce necessary notation and review the Tensor Train (TT) format in Sec. 3. In Sec. 4 we apply the TT-format to the weight matrix of a fully-connected layer and in Sec. 5 derive all the equations necessary for applying the back-propagation algorithm. In Sec. 6 we present the experimental evaluation of our ideas followed by a discussion in Sec. 7.

2 Related work

With sufficient amount of training data, big models usually outperform smaller ones. However state-of-the-art neural networks reached the hardware limits both in terms the computational power and the memory.

In particular, modern networks reached the memory limit with 89% [21] or even 100% [25] memory occupied by the weights of the fully-connected layers so it is not surprising that numerous attempts have been made to make the fully-connected layers more compact. One of the most straightforward approaches is to use a low-rank representation of the weight matrices. Recent studies show that the weight matrix of the fully-connected layer is highly redundant and by restricting its matrix rank it is possible to greatly reduce the number of parameters without significant drop in the predictive accuracy [6, 20, 25].

An alternative approach to the problem of model compression is to tie random subsets of weights using special hashing techniques [4]. The authors reported the compression factor of 8 for a two-layered network on the MNIST dataset without loss of accuracy. Memory consumption can also be reduced by using lower numerical precision [1] or allowing fewer possible carefully chosen parameter values [9].

In our paper we generalize the low-rank ideas. Instead of searching for low-rank approximation of the weight matrix we treat it as multi-dimensional tensor and apply the Tensor Train decomposition algorithm [17]. This framework has already been successfully applied to several data-processing tasks, e. g. [16, 27].

Another possible advantage of our approach is the ability to use more hidden units than was available before. A recent work [2] shows that it is possible to construct wide and shallow (i. e. not deep) neural networks with performance close to the state-of-the-art deep CNNs by training a shallow network on the outputs of a trained deep network. They report the improvement of performance with the increase of the layer size and used up to $30\,000$ hidden units while restricting the matrix rank of the weight matrix in order to be able to keep and to update it during the training. Restricting the TT-ranks of the weight matrix (in contrast to the matrix rank) allows to use much wider layers potentially leading to the greater expressive power of the model. We demonstrate this effect by training a very wide model ($262\,144$ hidden units) on the CIFAR-10 dataset that outperforms other non-convolutional networks.

Matrix and tensor decompositions were recently used to speed up the inference time of CNNs [7, 14]. While we focus on fully-connected layers, Lebedev et al. [14] used the CP-decomposition to compress a 4-dimensional convolution kernel and then used the properties of the decomposition to speed up the inference time. This work shares the same spirit with our method and the approaches can be readily combined.

Gilboa et al. exploit the properties of the Kronecker product of matrices to perform fast matrix-by-vector multiplication [8]. These matrices have the same structure as TT-matrices with unit TT-ranks.

Compared to the Tucker format [23] and the canonical format [3], the TT-format is immune to the curse of dimensionality and its algorithms are robust. Compared to the Hierarchical Tucker format [11], TT is quite similar but has simpler algorithms for basic operations.

3 TT-format

Throughout this paper we work with arrays of different dimensionality. We refer to the one-dimensional arrays as *vectors*, the two-dimensional arrays – *matrices*, the arrays of higher dimensions – *tensors*. Bold lower case letters (e. g. $\boldsymbol{a}$) denote vectors, ordinary lower case letters (e. g. $a(i) = a_i$) – vector elements, bold upper case letters (e. g. $\boldsymbol{A}$) – matrices, ordinary upper case letters (e. g. $A(i, j)$) – matrix elements, calligraphic bold upper case letters (e. g. $\boldsymbol{\mathcal{A}}$) – for tensors and

ordinary calligraphic upper case letters (e. g. $\mathcal{A}(\boldsymbol{i}) = \mathcal{A}(i_1, \ldots, i_d)$) – tensor elements, where d is the dimensionality of the tensor $\boldsymbol{\mathcal{A}}$.

We will call arrays *explicit* to highlight cases when they are stored explicitly, i. e. by enumeration of all the elements.

A d-dimensional array (tensor) $\boldsymbol{\mathcal{A}}$ is said to be represented in the *TT-format* [17] if for each dimension $k = 1, \ldots, d$ and for each possible value of the k-th dimension index $j_k = 1, \ldots, n_k$ there exists a matrix $\boldsymbol{G}_k[j_k]$ such that all the elements of $\boldsymbol{\mathcal{A}}$ can be computed as the following matrix product:

$$\mathcal{A}(j_1, \ldots, j_d) = \boldsymbol{G}_1[j_1]\boldsymbol{G}_2[j_2] \cdots \boldsymbol{G}_d[j_d]. \quad (1)$$

All the matrices $\boldsymbol{G}_k[j_k]$ related to the same dimension k are restricted to be of the same size $\mathrm{r}_{k-1} \times \mathrm{r}_k$. The values r_0 and r_d equal 1 in order to keep the matrix product (1) of size 1×1. In what follows we refer to the representation of a tensor in the TT-format as the *TT-representation* or the *TT-decomposition*. The sequence $\{\mathrm{r}_k\}_{k=0}^d$ is referred to as the *TT-ranks* of the TT-representation of $\boldsymbol{\mathcal{A}}$ (or the *ranks* for short), its maximum – as the *maximal TT-rank* of the TT-representation of $\boldsymbol{\mathcal{A}}$: $\mathrm{r} = \max_{k=0,\ldots,d} \mathrm{r}_k$. The collections of the matrices $(\boldsymbol{G}_k[j_k])_{j_k=1}^{n_k}$ corresponding to the same dimension (technically, 3-dimensional arrays $\boldsymbol{\mathcal{G}}_k$) are called the *cores*.

Oseledets [17, Th. 2.1] shows that for an arbitrary tensor $\boldsymbol{\mathcal{A}}$ a TT-representation exists but is not unique. The ranks among different TT-representations can vary and it's natural to seek a representation with the lowest ranks.

We use the symbols $G_k[j_k](\alpha_{k-1}, \alpha_k)$ to denote the element of the matrix $\boldsymbol{G}_k[j_k]$ in the position (α_{k-1}, α_k), where $\alpha_{k-1} = 1, \ldots, \mathrm{r}_{k-1}$, $\alpha_k = 1, \ldots, \mathrm{r}_k$. Equation (1) can be equivalently rewritten as the sum of the products of the elements of the cores:

$$\mathcal{A}(j_1, \ldots, j_d) = \sum_{\alpha_0, \ldots, \alpha_d} G_1[j_1](\alpha_0, \alpha_1) \ldots G_d[j_d](\alpha_{d-1}, \alpha_d). \quad (2)$$

The representation of a tensor $\boldsymbol{\mathcal{A}}$ via the explicit enumeration of all its elements requires to store $\prod_{k=1}^d n_k$ numbers compared with $\sum_{k=1}^d n_k \, \mathrm{r}_{k-1} \, \mathrm{r}_k$ numbers if the tensor is stored in the TT-format. Thus, the TT-format is very efficient in terms of memory if the ranks are small.

An attractive property of the TT-decomposition is the ability to efficiently perform several types of operations on tensors if they are in the TT-format: basic linear algebra operations, such as the addition of a constant and the multiplication by a constant, the summation and the entrywise product of tensors (the results of these operations are tensors in the TT-format generally with the increased ranks); computation of global characteristics of a tensor, such as the sum of all elements and the Frobenius norm. See [17] for a detailed description of all the supported operations.

3.1 TT-representations for vectors and matrices

The direct application of the TT-decomposition to a matrix (2-dimensional tensor) coincides with the low-rank matrix format and the direct TT-decomposition of a vector is equivalent to explicitly storing its elements. To be able to efficiently work with large vectors and matrices the TT-format for them is defined in a special manner. Consider a vector $\boldsymbol{b} \in \mathbb{R}^N$, where $N = \prod_{k=1}^d n_k$. We can establish a bijection $\boldsymbol{\mu}$ between the coordinate $\ell \in \{1, \ldots, N\}$ of $\boldsymbol{b}$ and a d-dimensional vector-index $\boldsymbol{\mu}(\ell) = (\mu_1(\ell), \ldots, \mu_d(\ell))$ of the corresponding tensor $\boldsymbol{\mathcal{B}}$, where $\mu_k(\ell) \in \{1, \ldots, n_k\}$. The tensor $\boldsymbol{\mathcal{B}}$ is then defined by the corresponding vector elements: $\mathcal{B}(\boldsymbol{\mu}(\ell)) = b_\ell$. Building a TT-representation of $\boldsymbol{\mathcal{B}}$ allows us to establish a compact format for the vector $\boldsymbol{b}$. We refer to it as a *TT-vector*.

Now we define a TT-representation of a matrix $\boldsymbol{W} \in \mathbb{R}^{M \times N}$, where $M = \prod_{k=1}^d m_k$ and $N = \prod_{k=1}^d n_k$. Let bijections $\boldsymbol{\nu}(t) = (\nu_1(t), \ldots, \nu_d(t))$ and $\boldsymbol{\mu}(\ell) = (\mu_1(\ell), \ldots, \mu_d(\ell))$ map row and column indices t and ℓ of the matrix $\boldsymbol{W}$ to the d-dimensional vector-indices whose k-th dimensions are of length m_k and n_k respectively, $k = 1, \ldots, d$. From the matrix $\boldsymbol{W}$ we can form a d-dimensional tensor $\boldsymbol{\mathcal{W}}$ whose k-th dimension is of length $m_k n_k$ and is indexed by the tuple $(\nu_k(t), \mu_k(\ell))$. The tensor $\boldsymbol{\mathcal{W}}$ can then be converted into the TT-format:

$$W(t, \ell) = \mathcal{W}((\nu_1(t), \mu_1(\ell)), \ldots, (\nu_d(t), \mu_d(\ell))) = \boldsymbol{G}_1[\nu_1(t), \mu_1(\ell)] \ldots \boldsymbol{G}_d[\nu_d(t), \mu_d(\ell)], \quad (3)$$

where the matrices $\boldsymbol{G}_k[\nu_k(t), \mu_k(\ell)]$, $k = 1, \ldots, d$, serve as the cores with tuple $(\nu_k(t), \mu_k(\ell))$ being an index. Note that a matrix in the TT-format is not restricted to be square. Although index-vectors $\boldsymbol{\nu}(t)$ and $\boldsymbol{\mu}(\ell)$ are of the same length d, the sizes of the domains of the dimensions can vary. We call a matrix in the TT-format a *TT-matrix*.

All operations available for the TT-tensors are applicable to the TT-vectors and the TT-matrices as well (for example one can efficiently sum two TT-matrices and get the result in the TT-format). Additionally, the TT-format allows to efficiently perform the matrix-by-vector (matrix-by-matrix) product. If only one of the operands is in the TT-format, the result would be an explicit vector (matrix); if both operands are in the TT-format, the operation would be even more efficient and the result would be given in the TT-format as well (generally with the increased ranks). For the case of the TT-matrix-by-explicit-vector product $\boldsymbol{c} = \boldsymbol{W}\boldsymbol{b}$, the computational complexity is $O(d\,r^2\,m\max\{M,N\})$, where d is the number of the cores of the TT-matrix $\boldsymbol{W}$, $m = \max_{k=1,\ldots,d} m_k$, r is the maximal rank and $N = \prod_{k=1}^{d} n_k$ is the length of the vector $\boldsymbol{b}$.

The ranks and, correspondingly, the efficiency of the TT-format for a vector (matrix) depend on the choice of the mapping $\boldsymbol{\mu}(\ell)$ (mappings $\boldsymbol{\nu}(t)$ and $\boldsymbol{\mu}(\ell)$) between vector (matrix) elements and the underlying tensor elements. In what follows we use a column-major MATLAB `reshape` command [1] to form a d-dimensional tensor from the data (e. g. from a multichannel image), but one can choose a different mapping.

4 TT-layer

In this section we introduce the *TT-layer* of a neural network. In short, the TT-layer is a fully-connected layer with the weight matrix stored in the TT-format. We will refer to a neural network with one or more TT-layers as *TensorNet*.

Fully-connected layers apply a linear transformation to an N-dimensional input vector $\boldsymbol{x}$:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \tag{4}$$

where the *weight matrix* $\boldsymbol{W} \in \mathbb{R}^{M\times N}$ and the *bias vector* $\boldsymbol{b} \in \mathbb{R}^{M}$ define the transformation.

A *TT-layer* consists in storing the weights $\boldsymbol{W}$ of the fully-connected layer in the TT-format, allowing to use hundreds of thousands (or even millions) of hidden units while having moderate number of parameters. To control the number of parameters one can vary the number of hidden units as well as the TT-ranks of the weight matrix.

A TT-layer transforms a d-dimensional tensor $\boldsymbol{\mathcal{X}}$ (formed from the corresponding vector $\boldsymbol{x}$) to the d-dimensional tensor $\boldsymbol{\mathcal{Y}}$ (which correspond to the output vector $\boldsymbol{y}$). We assume that the weight matrix $\boldsymbol{W}$ is represented in the TT-format with the cores $\boldsymbol{G}_k[i_k, j_k]$. The linear transformation (4) of a fully-connected layer can be expressed in the tensor form:

$$\mathcal{Y}(i_1,\ldots,i_d) = \sum_{j_1,\ldots,j_d} \boldsymbol{G}_1[i_1,j_1]\ldots\boldsymbol{G}_d[i_d,j_d]\,\mathcal{X}(j_1,\ldots,j_d) + \mathcal{B}(i_1,\ldots,i_d). \tag{5}$$

Direct application of the TT-matrix-by-vector operation for the Eq. (5) yields the computational complexity of the forward pass $O(dr^2m\max\{m,n\}^d) = O(dr^2m\max\{M,N\})$.

5 Learning

Neural networks are usually trained with the stochastic gradient descent algorithm where the gradient is computed using the back-propagation procedure [18]. Back-propagation allows to compute the gradient of a loss-function L with respect to all the parameters of the network. The method starts with the computation of the gradient of L w.r.t. the output of the last layer and proceeds sequentially through the layers in the reversed order while computing the gradient w.r.t. the parameters and the input of the layer making use of the gradients computed earlier. Applied to the fully-connected layers (4) the back-propagation method computes the gradients w.r.t. the input $\boldsymbol{x}$ and the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ given the gradients $\frac{\partial L}{\partial \boldsymbol{y}}$ w.r.t to the output $\boldsymbol{y}$:

$$\frac{\partial L}{\partial \boldsymbol{x}} = \boldsymbol{W}^{\mathsf{T}}\frac{\partial L}{\partial \boldsymbol{y}}, \qquad \frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{y}}\boldsymbol{x}^{\mathsf{T}}, \qquad \frac{\partial L}{\partial \boldsymbol{b}} = \frac{\partial L}{\partial \boldsymbol{y}}. \tag{6}$$

In what follows we derive the gradients required to use the back-propagation algorithm with the TT-layer. To compute the gradient of the loss function w.r.t. the bias vector $\boldsymbol{b}$ and w.r.t. the input vector $\boldsymbol{x}$ one can use equations (6). The latter can be applied using the matrix-by-vector product (where the matrix is in the TT-format) with the complexity of $O(dr^2n\max\{m,n\}^d) = O(dr^2n\max\{M,N\})$.

[1] `http://www.mathworks.com/help/matlab/ref/reshape.html`

Operation	Time	Memory
FC forward pass	$O(MN)$	$O(MN)$
TT forward pass	$O(dr^2 m \max\{M, N\})$	$O(r \max\{M, N\})$
FC backward pass	$O(MN)$	$O(MN)$
TT backward pass	$O(d^2\, \mathrm{r}^4\, m \max\{M, N\})$	$O(\mathrm{r}^3 \max\{M, N\})$

Table 1: Comparison of the asymptotic complexity and memory usage of an $M \times N$ TT-layer and an $M \times N$ fully-connected layer (FC). The input and output tensor shapes are $m_1 \times \ldots \times m_d$ and $n_1 \times \ldots \times n_d$ respectively ($m = \max_{k=1\ldots d} m_k$) and r is the maximal TT-rank.

To perform a step of stochastic gradient descent one can use equation (6) to compute the gradient of the loss function w.r.t. the weight matrix $\boldsymbol{W}$, convert the gradient matrix into the TT-format (with the TT-SVD algorithm [17]) and then add this gradient (multiplied by a step size) to the current estimate of the weight matrix: $\boldsymbol{W}_{k+1} = \boldsymbol{W}_k + \gamma_k \frac{\partial L}{\partial \boldsymbol{W}}$. However, the direct computation of $\frac{\partial L}{\partial \boldsymbol{W}}$ requires $O(MN)$ memory. A better way to learn the TensorNet parameters is to compute the gradient of the loss function directly w.r.t. the cores of the TT-representation of $\boldsymbol{W}$.

In what follows we use shortened notation for prefix and postfix sequences of indices: $\boldsymbol{i}_k^- := (i_1, \ldots, i_{k-1})$, $\boldsymbol{i}_k^+ := (i_{k+1}, \ldots, i_d)$, $\boldsymbol{i} = (\boldsymbol{i}_k^-, i_k, \boldsymbol{i}_k^+)$. We also introduce notations for partial core products:

$$\begin{aligned} \boldsymbol{P}_k^-[\boldsymbol{i}_k^-, \boldsymbol{j}_k^-] &:= \boldsymbol{G}_1[i_1, j_1] \ldots \boldsymbol{G}_{k-1}[i_{k-1}, j_{k-1}], \\ \boldsymbol{P}_k^+[\boldsymbol{i}_k^+, \boldsymbol{j}_k^+] &:= \boldsymbol{G}_{k+1}[i_{k+1}, j_{k+1}] \ldots \boldsymbol{G}_d[i_d, j_d]. \end{aligned} \tag{7}$$

We now rewrite the definition of the TT-layer transformation (5) for any $k = 2, \ldots, d-1$:

$$\mathcal{Y}(\boldsymbol{i}) = \mathcal{Y}(\boldsymbol{i}_k^-, i_k, \boldsymbol{i}_k^+) = \sum_{\boldsymbol{j}_k^-, j_k, \boldsymbol{j}_k^+} \boldsymbol{P}_k^-[\boldsymbol{i}_k^-, \boldsymbol{j}_k^-] \boldsymbol{G}_k[i_k, j_k] \boldsymbol{P}_k^+[\boldsymbol{i}_k^+, \boldsymbol{j}_k^+] \mathcal{X}(\boldsymbol{j}_k^-, j_k, \boldsymbol{j}_k^+) + \mathcal{B}(\boldsymbol{i}). \tag{8}$$

The gradient of the loss function L w.r.t. to the k-th core in the position $[\tilde{i}_k, \tilde{j}_k]$ can be computed using the chain rule:

$$\underbrace{\frac{\partial L}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}}_{\mathrm{r}_{k-1} \times \mathrm{r}_k} = \sum_{\boldsymbol{i}} \frac{\partial L}{\partial \mathcal{Y}(\boldsymbol{i})} \frac{\partial \mathcal{Y}(\boldsymbol{i})}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}. \tag{9}$$

Given the gradient matrices $\frac{\partial \mathcal{Y}(\boldsymbol{i})}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}$ the summation (9) can be done explicitly in $O(M\, \mathrm{r}_{k-1}\, \mathrm{r}_k)$ time, where M is the length of the output vector $\boldsymbol{y}$.

We now show how to compute the matrix $\frac{\partial \mathcal{Y}(\boldsymbol{i})}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}$ for any values of the core index $k \in \{1, \ldots, d\}$ and $\tilde{i}_k \in \{1, \ldots, m_k\}$, $\tilde{j}_k \in \{1, \ldots, n_k\}$. For any $\boldsymbol{i} = (i_1, \ldots, i_d)$ such that $i_k \neq \tilde{i}_k$ the value of $\mathcal{Y}(\boldsymbol{i})$ doesn't depend on the elements of $\boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]$ making the corresponding gradient $\frac{\partial \mathcal{Y}(\boldsymbol{i})}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}$ equal zero. Similarly, any summand in the Eq. (8) such that $j_k \neq \tilde{j}_k$ doesn't affect the gradient $\frac{\partial \mathcal{Y}(\boldsymbol{i})}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]}$. These observations allow us to consider only $i_k = \tilde{i}_k$ and $j_k = \tilde{j}_k$.

$\mathcal{Y}(\boldsymbol{i}_k^-, \tilde{i}_k, \boldsymbol{i}_k^+)$ is a linear function of the core $\boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]$ and its gradient equals the following expression:

$$\frac{\partial \mathcal{Y}(\boldsymbol{i}_k^-, \tilde{i}_k, \boldsymbol{i}_k^+)}{\partial \boldsymbol{G}_k[\tilde{i}_k, \tilde{j}_k]} = \sum_{\boldsymbol{j}_k^-, \boldsymbol{j}_k^+} \underbrace{\left(\boldsymbol{P}_k^-[\boldsymbol{i}_k^-, \boldsymbol{j}_k^-]\right)^\intercal}_{\mathrm{r}_{k-1} \times 1} \underbrace{\left(\boldsymbol{P}_k^+[\boldsymbol{i}_k^+, \boldsymbol{j}_k^+]\right)^\intercal}_{1 \times \mathrm{r}_k} \mathcal{X}(\boldsymbol{j}_k^-, \tilde{j}_k, \boldsymbol{j}_k^+). \tag{10}$$

We denote the partial sum vector as $\boldsymbol{R}_k[\boldsymbol{j}_k^-, \tilde{j}_k, \boldsymbol{i}_k^+] \in \mathbb{R}^{\mathrm{r}_k}$:

$$\boldsymbol{R}_k[j_1, \ldots, j_{k-1}, \tilde{j}_k, i_{k+1}, \ldots, i_d] = \boldsymbol{R}_k[\boldsymbol{j}_k^-, \tilde{j}_k, \boldsymbol{i}_k^+] = \sum_{\boldsymbol{j}_k^+} \boldsymbol{P}_k^+[\boldsymbol{i}_k^+, \boldsymbol{j}_k^+]\ \mathcal{X}(\boldsymbol{j}_k^-, \tilde{j}_k, \boldsymbol{j}_k^+).$$

Vectors $\boldsymbol{R}_k[\boldsymbol{j}_k^-, \tilde{j}_k, \boldsymbol{i}_k^+]$ for all the possible values of k, $\boldsymbol{j}_k^-$, $\tilde{j}_k$ and $\boldsymbol{i}_k^+$ can be computed via dynamic programming (by pushing sums w.r.t. each $j_{k+1}, \ldots, j_d$ inside the equation and summing out one index at a time) in $O(dr^2 m \max\{M, N\})$. Substituting these vectors into (10) and using

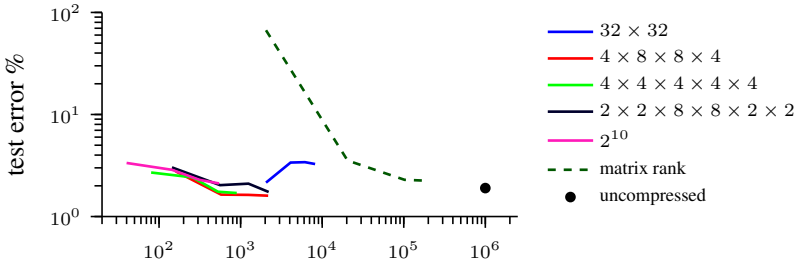

Figure 1: The experiment on the MNIST dataset. We use a two-layered neural network and substitute the first 1024×1024 fully-connected layer with the TT-layer (solid lines) and with the matrix rank decomposition based layer (dashed line). The solid lines of different colors correspond to different ways of reshaping the input and output vectors to tensors (the shapes are reported in the legend). To obtain the points of the plots we vary the maximal TT-rank or the matrix rank.

(again) dynamic programming yields us all the necesary matrices for summation (9). The overall computational complexity of the backward pass is $O(d^2\, \mathrm{r}^4\, m \max\{M, N\})$.

The presented algorithm reduces to a sequence of matrix-by-matrix products and permutations of dimensions and thus can be accelerated on a GPU device.

6 Experiments

6.1 Parameters of the TT-layer

In this experiment we investigate the properties of the TT-layer and compare different strategies for setting its parameters: dimensions of the tensors representing the input/output of the layer and the TT-ranks of the compressed weight matrix. We run the experiment on the MNIST dataset [15] for the task of handwritten-digit recognition. As a baseline we use a neural network with two fully-connected layers (1024 hidden units) and rectified linear unit (ReLU) achieving 1.9% error on the test set. For more reshaping options we resize the original 28×28 images to 32×32.

We train several networks differing in the parameters of the single TT-layer. The networks contain the following layers: the TT-layer with weight matrix of size 1024×1024, ReLU, the fully-connected layer with the weight matrix of size 1024×10. We test different ways of reshaping the input/output tensors and try different ranks of the TT-layer. As a simple compression baseline in the place of the TT-layer we use the fully-connected layer such that the rank of the weight matrix is bounded (implemented as follows: the two consecutive fully-connected layers with weight matrices of sizes $1024 \times \mathrm{r}$ and $\mathrm{r} \times 1024$, where r controls the matrix rank and the compression factor). The results of the experiment are shown in Figure 1. We conclude that the TT-ranks provide much better flexibility than the matrix rank when applied at the same compression level. In addition, we observe that the TT-layers with too small number of values for each tensor dimension and with too few dimensions perform worse than their more balanced counterparts.

Comparison with HashedNet [4]. We consider a two-layered neural network with 1024 hidden units and replace both fully-connected layers by the TT-layers. By setting all the TT-ranks in the network to 8 we achieved the test error of 1.6% with $12\,602$ parameters in total and by setting all the TT-ranks to 6 the test error of 1.9% with $7\,698$ parameters. Chen et al. [4] report results on the same architecture. By tying random subsets of weights they compressed the network by the factor of 64 to the $12\,720$ parameters in total with the test error equal 2.79%.

6.2 CIFAR-10

CIFAR-10 dataset [12] consists of 32×32 3-channel images assigned to 10 different classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. The dataset contains 50000 train and 10000 test images. Following [10] we preprocess the images by subtracting the mean and performing global contrast normalization and ZCA whitening.

As a baseline we use the CIFAR-10 Quick [22] CNN, which consists of convolutional, pooling and non-linearity layers followed by two fully-connected layers of sizes 1024×64 and 64×10. We fix the convolutional part of the network and substitute the fully-connected part by a $1024 \times N$ TT-layer

Architecture	TT-layers compr.	vgg-16 compr.	vgg-19 compr.	vgg-16 top 1	vgg-16 top 5	vgg-19 top 1	vgg-19 top 5
FC FC FC	1	1	1	30.9	11.2	29.0	10.1
TT4 FC FC	50 972	3.9	3.5	31.2	11.2	29.8	10.4
TT2 FC FC	194 622	3.9	3.5	31.5	11.5	30.4	10.9
TT1 FC FC	713 614	3.9	3.5	33.3	12.8	31.9	11.8
TT4 TT4 FC	37 732	7.4	6	32.2	12.3	31.6	11.7
MR1 FC FC	3 521	3.9	3.5	99.5	97.6	99.8	99
MR5 FC FC	704	3.9	3.5	81.7	53.9	79.1	52.4
MR50 FC FC	70	3.7	3.4	36.7	14.9	34.5	15.8

Table 2: Substituting the fully-connected layers with the TT-layers in vgg-16 and vgg-19 networks on the ImageNet dataset. FC stands for a fully-connected layer; TT$\square$ stands for a TT-layer with all the TT-ranks equal "$\square$"; MR$\square$ stands for a fully-connected layer with the matrix rank restricted to "$\square$". We report the compression rate of the TT-layers matrices and of the whole network in the second, third and fourth columns.

followed by ReLU and by a $N \times 10$ fully-connected layer. With $N = 3125$ hidden units (contrary to 64 in the original network) we achieve the test error of 23.13% without fine-tuning which is slightly better than the test error of the baseline (23.25%). The TT-layer treated input and output vectors as $4 \times 4 \times 4 \times 4 \times 4 \times 4$ and $5 \times 5 \times 5 \times 5 \times 5 \times 5$ tensors respectively. All the TT-ranks equal 8, making the number of the parameters in the TT-layer equal $4\,160$. The compression rate of the TensorNet compared with the baseline w.r.t. all the parameters is 1.24. In addition, substituting the both fully-connected layers by the TT-layers yields the test error of 24.39% and reduces the number of parameters of the fully-connected layer matrices by the factor of 11.9 and the total parameter number by the factor of 1.7.

For comparison, in [6] the fully-connected layers in a CIFAR-10 CNN were compressed by the factor of at most 4.7 times with the loss of about 2% in accuracy.

6.2.1 Wide and shallow network

With sufficient amount of hidden units, even a neural network with two fully-connected layers and sigmoid non-linearity can approximate any decision boundary [5]. Traditionally, very wide shallow networks are not considered because of high computational and memory demands and the over-fitting risk. TensorNet can potentially address both issues. We use a three-layered TensorNet of the following architecture: the TT-layer with the weight matrix of size $3\,072 \times 262\,144$, ReLU, the TT-layer with the weight matrix of size $262\,144 \times 4\,096$, ReLU, the fully-connected layer with the weight matrix of size $4\,096 \times 10$. We report the test error of 31.47% which is (to the best of our knowledge) the best result achieved by a non-convolutional neural network.

6.3 ImageNet

In this experiment we evaluate the TT-layers on a large scale task. We consider the 1000-class ImageNet ILSVRC-2012 dataset [19], which consist of 1.2 million training images and 50 000 validation images. We use deep the CNNs vgg-16 and vgg-19 [21] as the reference models[2]. Both networks consist of the two parts: the convolutional and the fully-connected parts. In the both networks the second part consist of 3 fully-connected layers with weight matrices of sizes 25088×4096, 4096×4096 and 4096×1000.

In each network we substitute the first fully-connected layer with the TT-layer. To do this we reshape the 25088-dimensional input vectors to the tensors of the size $2 \times 7 \times 8 \times 8 \times 7 \times 4$ and the 4096-dimensional output vectors to the tensors of the size $4 \times 4 \times 4 \times 4 \times 4 \times 4$. The remaining fully-connected layers are initialized randomly. The parameters of the convolutional parts are kept fixed as trained by Simonyan and Zisserman [21]. We train the TT-layer and the fully-connected layers on the training set. In Table 2 we vary the ranks of the TT-layer and report the compression factor of the TT-layers (vs. the original fully-connected layer), the resulting compression factor of the whole network, and the top 1 and top 5 errors on the validation set. In addition, we substitute the second fully-connected layer with the TT-layer. As a baseline compression method we constrain the matrix rank of the weight matrix of the first fully-connected layer using the approach of [2].

[2]After we had started to experiment on the vgg-16 network the vgg-* networks have been improved by the authors. Thus, we report the results on a slightly outdated version of vgg-16 and the up-to-date version of vgg-19.

Type	1 im. time (ms)	100 im. time (ms)
CPU fully-connected layer	16.1	97.2
CPU TT-layer	1.2	94.7
GPU fully-connected layer	2.7	33
GPU TT-layer	1.9	12.9

Table 3: Inference time for a 25088×4096 fully-connected layer and its corresponding TT-layer with all the TT-ranks equal 4. The memory usage for feeding forward one image is 392MB for the fully-connected layer and 0.766MB for the TT-layer.

In Table 2 we observe that the TT-layer in the best case manages to reduce the number of the parameters in the matrix $\boldsymbol{W}$ of the largest fully-connected layer by a factor of $194\,622$ (from 25088×4096 parameters to 528) while increasing the top 5 error from 11.2 to 11.5. The compression factor of the whole network remains at the level of 3.9 because the TT-layer stops being the storage bottleneck. By compressing the largest of the remaining layers the compression factor goes up to 7.4. The baseline method when providing similar compression rates significantly increases the error.

For comparison, consider the results of [26] obtained for the compression of the fully-connected layers of the Krizhevsky-type network [13] with the Fastfood method. The model achieves compression factors of 2-3 without decreasing the network error.

6.4 Implementation details

In all experiments we use our MATLAB extension[3] of the MatConvNet framework[4] [24]. For the operations related to the TT-format we use the TT-Toolbox[5] implemented in MATLAB as well. The experiments were performed on a computer with a quad-core Intel Core i5-4460 CPU, 16 GB RAM and a single NVidia Geforce GTX 980 GPU. We report the running times and the memory usage at the forward pass of the TT-layer and the baseline fully-connected layer in Table 3.

We train all the networks with stochastic gradient descent with momentum (coefficient 0.9). We initialize all the parameters of the TT- and fully-connected layers with a Gaussian noise and put L2-regularization (weight 0.0005) on them.

7 Discussion and future work

Recent studies indicate high redundancy in the current neural network parametrization. To exploit this redundancy we propose to use the TT-decomposition framework on the weight matrix of a fully-connected layer and to use the cores of the decomposition as the parameters of the layer. This allows us to train the fully-connected layers compressed by up to $200\,000\times$ compared with the explicit parametrization without significant error increase. Our experiments show that it is possible to capture complex dependencies within the data by using much more compact representations. On the other hand it becomes possible to use much wider layers than was available before and the preliminary experiments on the CIFAR-10 dataset show that wide and shallow TensorNets achieve promising results (setting new state-of-the-art for non-convolutional neural networks).

Another appealing property of the TT-layer is faster inference time (compared with the corresponding fully-connected layer). All in all a wide and shallow TensorNet can become a time and memory efficient model to use in real time applications and on mobile devices.

The main limiting factor for an $M \times N$ fully-connected layer size is its parameters number MN. The limiting factor for an $M \times N$ TT-layer is the maximal linear size $\max\{M, N\}$. As a future work we plan to consider the inputs and outputs of layers in the TT-format thus completely eliminating the dependency on M and N and allowing billions of hidden units in a TT-layer.

Acknowledgements. We would like to thank Ivan Oseledets for valuable discussions. A. Novikov, D. Podoprikhin, D. Vetrov were supported by RFBR project No. 15-31-20596 (mol-a-ved) and by Microsoft: Moscow State University Joint Research Center (RPD 1053945). A. Osokin was supported by the MSR-INRIA Joint Center. The results of the tensor toolbox application (in Sec. 6) are supported by Russian Science Foundation No. 14-11-00659.

[3] https://github.com/Bihaqo/TensorNet

[4] http://www.vlfeat.org/matconvnet/

[5] https://github.com/oseledets/TT-Toolbox

References

[1] K. Asanovi and N. Morgan, "Experimental determination of precision requirements for back-propagation training of artificial neural networks," International Computer Science Institute, Tech. Rep., 1991.

[2] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in Neural Information Processing Systems 27 (NIPS)*, 2014, pp. 2654–2662.

[3] J. D. Caroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via n-way generalization of Eckart-Young decomposition," *Psychometrika*, vol. 35, pp. 283–319, 1970.

[4] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International Conference on Machine Learning (ICML)*, 2015, pp. 2285–2294.

[5] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, pp. 303–314, 1989.

[6] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems 26 (NIPS)*, 2013, pp. 2148–2156.

[7] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems 27 (NIPS)*, 2014, pp. 1269–1277.

[8] E. Gilboa, Y. Saati, and J. P. Cunningham, "Scaling multidimensional inference for structured gaussian processes," *arXiv preprint*, no. 1209.4120, 2012.

[9] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint*, no. 1412.6115, 2014.

[10] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *International Conference on Machine Learning (ICML)*, 2013, pp. 1319–1327.

[11] W. Hackbusch and S. Kühn, "A new scheme for the tensor representation," *J. Fourier Anal. Appl.*, vol. 15, pp. 706–722, 2009.

[12] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, Computer Science Department, University of Toronto, 2009.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25 (NIPS)*, 2012, pp. 1097–1105.

[14] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," in *International Conference on Learning Representations (ICLR)*, 2014.

[15] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits," 1998.

[16] A. Novikov, A. Rodomanov, A. Osokin, and D. Vetrov, "Putting MRFs on a Tensor Train," in *International Conference on Machine Learning (ICML)*, 2014, pp. 811–819.

[17] I. V. Oseledets, "Tensor-Train decomposition," *SIAM J. Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, 2015.

[20] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *International Conference of Acoustics, Speech, and Signal Processing (ICASSP)*, 2013, pp. 6655–6659.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[22] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems 25 (NIPS)*, 2012, pp. 2951–2959.

[23] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[24] A. Vedaldi and K. Lenc, "Matconvnet – convolutional neural networks for MATLAB," in *Proceeding of the ACM Int. Conf. on Multimedia*.

[25] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition," in *Interspeech*, 2013, pp. 2365–2369.

[26] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang, "Deep fried convnets," *arXiv preprint*, no. 1412.7149, 2014.

[27] Z. Zhang, X. Yang, I. V. Oseledets, G. E. Karniadakis, and L. Daniel, "Enabling high-dimensional hierarchical uncertainty quantification by ANOVA and tensor-train decomposition," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pp. 63–76, 2014.