# Canonical convolutional neural networks

Lokesh Veeramacheneni
*Department of Computer Science*
*Hochschule Bonn-Rhein-Sieg*
lokesh.veeramacheneni@smail.inf.h-brs.de

Moritz Wolter
*High Performance Computing and Analytics Lab*
*University of Bonn*
*Fraunhofer Center for Machine Learning and SCAI*
wolter@cs.uni-bonn.de

Reinhard Klein
*Department of Computer Science*
*University of Bonn*
rk@cs.uni-bonn.de

Jochen Garcke
*Fraunhofer Center for Machine Learning and SCAI*
*Institute for Numerical Simulation*
*University of Bonn*
garcke@ins.uni-bonn.de, jochen.garcke@scai.fhg.de

***Abstract*—We introduce canonical weight normalization for convolutional neural networks. Inspired by the canonical tensor decomposition, we express the weight tensors in so-called canonical networks as scaled sums of outer vector products. In particular, we train network weights in the decomposed form, where scale weights are optimized separately for each mode. Additionally, similarly to weight normalization, we include a global scaling parameter. We study the initialization of the canonical form by running the power method and by drawing randomly from Gaussian or uniform distributions. Our results indicate that we can replace the power method with cheaper initializations drawn from standard distributions. The canonical re-parametrization leads to competitive normalization performance on the MNIST, CIFAR10, and SVHN data sets. Moreover, the formulation simplifies network compression. Once training has converged, the canonical form allows convenient model-compression by truncating the parameter sums.**

## I. Introduction

While deep neural networks have increased in size and complexity, the tensor structure of convolutional kernels and weight matrices has not changed as rapidly. We believe that much of the potential that tensor representations, such as so-called canonical decompositions, can offer remains to be discovered.

This paper proposes to express, train, *normalize, and compress* network weight tensors in a canonical form as a sum of weighted normalized outer vector products. Similar to weight normalization [30], the resulting canonical normalization allows us to learn an overall length weight. In addition to the overall length, the canonical representation lets the optimizer scale all individual rank terms in the sum separately, where all canonical rank vectors are re-normalized after each optimization step. Consequently, the sum's rank weights remain meaningful throughout training. Orientation and magnitude are decoupled. In addition to normalization, we observe that the canonical representation is helpful for network compression. Having learned the scales for each rank separately, we can truncate the sum of outer products according to the weight of each rank component to allow easy compression.

The methods closest to ours are the low-rank form forms proposed in [22] and [33]. Like [33], we overcome the instability problem observed by [22]. Instead of working with batch-normalization [33], we propose CP-normalization. We jointly consider convolutional and fully connected layers, propose a new way to initialize the canonical form, and explore compression. CP-normalization is a form of weight normalization [30] for convolutional neural networks.

We implement the proposed methods and all experiments using Tensorly [15] and Pytorch [28]. Our contributions can be summarized as follows:

- Canonical convolutional neural networks re-express network weights as sums of outer vector products. The formulation improves convergence by scaling overall and per rank lengths separately.
- We compare our formulation to weight normalization [30] and the low rank form of [33], on the MNIST [23], CIFAR-10 [17], and SVHN [25] data sets. We find CP-normalization performs competitively.
- Having optimized weights for every rank summand, we can sort the summands according to the absolute value of their weight and truncate them according to importance. Consequently, the CP-formulation allows straightforward weight compression by truncation.
- We study the initialization of the canonical form and replace the standard power method approach with direct initialization for an AlexNet-like architecture on CIFAR-10.

For reproducibility and future work, the source code is available at https://github.com/Fraunhofer-SCAI/canonical-cnn.

## II. Related Work

### A. Normalization and Regularization

Normalization and regularization methods broadly fall into three categories. Noise-based methods encourage generalization by corrupting network features or input data. The noise

randomly hides certain features, thereby denying overfitting by forcing the network to rely on multiple features to evade the noise. Dropout [32], adaptive dropout [3], stochastic pooling [39], input noise [5], [10] as well as weight or synaptic noise [10] fall into this category. Dropout randomly removes neurons during training, using a fixed removal probability. Adaptive dropout optimizes the removal probability per neuron. Stochastic pooling sub-samples by choosing activations randomly. Input noise randomly corrupts training inputs. Weight noise is added to the parameters to move the model away from local minima and reduce the amount to which subsequent layers can rely on individual features.

Methods in the second group change the cost function to encourage generalization. Weight decay adds an $L_2$ loss term to the cost function [5], [9], to prevent excessive parameter growth. Placing a cost on the $L_2$ parameter norm prevents weight growth and limits network complexity by pushing parameters to zero.

Finally, structural methods modify the network structure to achieve a regularizing effect. Batch normalization [12] for example, adjusts the mean and variance of intermediate representations to be approximately standard normal. Weight normalization [30] resets the length of weight tensors and introduces an additional length parameter per tensor. It measures tensor length by computing the length of a corresponding flat vector. The normalized weight vector is divided by the vector length after every weight update [30].

Weight decay penalizes the $L_2$ term. Normalization does not. It merely seeks to improve the conditioning of the underlying optimization problem.

Our approach also falls into this category. Even though the Euclidean length is a valid tensor norm [14], we argue that preserving and measuring individual rank length is beneficial because it shares the normalizing properties of weight normalization while additionally simplifying network compression.

### *B. Network Compression*

Two effective ways to compress artificial neural networks are quantization [16] and pruning. Quantization techniques save memory space and computation time by storing the network tensors at less than floating-point precision. Pruning removes parameters that contribute little to the overall performance. Pruned weights are, for example, removed based on the individual magnitude of weights [11] or by removing entire rows based on the row-norm [21] for improved efficiency.

An alternative to pruning is to enforce sparse or structured matrices a-priori. By shifting and reusing the same row, implementing circulant matrix structures saves weights [1]. Alternatively, the frequency domain can help us impose sparse diagonal patterns onto the network weight matrices. The fastfood approach proposed in [38] works with a Welsh-Hadamard transform. In addition to the fixed Welsh-Hadamard transform, adaptive wavelet transforms [36] are known to work.

### *C. Tensor Decompositions*

Tensor decompositions are well-established in science, and engineering [6], [14]. The machine learning community has previously studied CANDECOMP/PARAFAC (CP) decompositions, see, e.g., [19], [24]. A common approach is to compress pre-trained convolutional neural networks [2], [13], [19], [22], [29]. In particular, after converting the converged weights to the canonical format, [22] uses fine-tuning after application of the CP-decomposition, with tiny learning rates. [4] investigated sums of separable functions as a functional analog to the canonical decomposition for regression and classification [8]. [29] adjusts the computation of the CP-decomposition to yield a representation that is stable during the ensuing fine-tuning.

An alternative to the canonical or parallel factors format is the tensor train representation [14]. The tensor train format has been used to train compressed versions of fully connected layers in CNN [26], [37], RNN [34], and GANs [27].

Training low-rank CNN from scratch was previously explored in [33]. The proposed approach introduces additional layers. The extra layers lead to deeper networks, which are harder to train. Batch normalization is applied to deal with arising instabilities.

We argue that introducing additional layers is not required. Our scaled CP-coefficients stabilize training and allow joint normalization and compression similar to weight normalization.

To bolster our argument, we study the link between normalization and compression. We, therefore, apply a canonical formulation already during training. To stabilize the formulation, we regularly re-normalize our vectors. We explore initialization by computing the CP-decomposition and random initializations using various distributions. Additionally, we will explore truncating the canonical sum for compression after convergence.

## III. Methods

In this section, we will discuss the mathematical background and notations. Afterwards, we briefly revisit the canonical tensor decomposition and introduce our weight reparametrization. We adopt the notation from [14], to which we also refer for a further introduction into tensor decompositions. Throughout the text, vectors are denoted as boldface lowercase letters, for example, $\mathbf{x}, \mathbf{y}$. Capital letters $\mathbf{A}, \mathbf{B}$ stand for matrices. Finally, Euler script $\mathcal{X}$ denotes tensors.

### *A. The Outer Product* $\circ$

In the two dimensional case, the outer product $\circ$ of two vectors $x \in \mathbb{R}^i, y \in \mathbb{R}^j$,

$$\mathbf{x} \circ \mathbf{y} = \mathbf{x}\mathbf{y}^T = \mathbf{A}, \tag{1}$$

results in a matrix $\mathbf{A} \in \mathbb{R}^{i,j}$. If we wanted to turn $\mathbf{A}$ into tensor $\mathcal{A}$ we could simply add additional vectors to the chain of outer products. For example, using $\mathbf{z} \in \mathbb{R}^k$ we could produce $\mathbf{x} \circ \mathbf{y} \circ \mathbf{z} = \mathcal{A} \in \mathbb{R}^{i,j,k}$. Formally speaking, a $n$-dimensional tensor $\mathcal{X} \in \mathbb{R}^{d_1,\dots,d_n}$ *of rank one* can be rewritten as an outer product of $n$ vectors [14]

$$\mathcal{X} = \mathbf{x}^{(1)} \circ \mathbf{x}^{(2)} \circ \mathbf{x}^{(3)} \dots \circ \mathbf{x}^{(n)}, \tag{2}$$

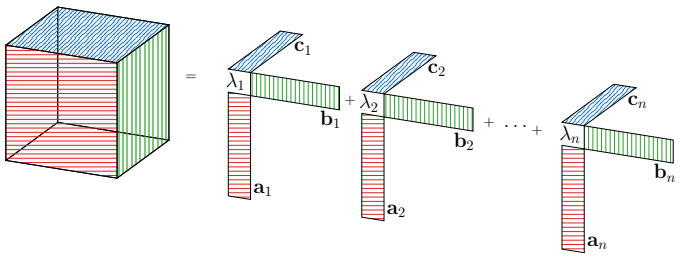


Fig. 1: Visualization of a canonical decomposition of a tensor in $\mathbb{R}^3$. For a third order tensor of size $\mathbb{R}^{i,j,k}$ we expect three vectors of shape $\mathbf{a} \in \mathbb{R}^i, \mathbf{b} \in \mathbb{R}^j, \mathbf{c} \in \mathbb{R}^k$ in the outer products of each summand. Vectors are expressed by colored rectangles. Orthogonally arranged rectangles symbolize outer products. Assuming unit length vectors, we include $\lambda_r$ in each element of the sum. The summation runs until $n$ or the total CP-tensor rank is reached.

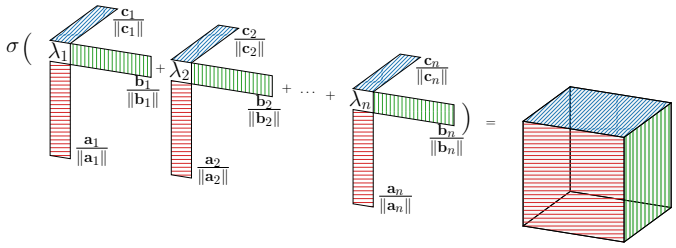


Fig. 2: Visualization of the proposed canonical weight representation. The tensor weights are expressed as sums of scaled outer products, here illustrated as orthogonal stripes. In addition to the rank scales $\lambda$ we use a global length weight $\sigma$. After every update step, the weight vectors are re-normalized. The rank scales $\lambda_r$ and the global scale $\sigma$ allow learning. After convergence, the rank scales let us assess the relative importance of each rank. Compressing the network means discarding the least important terms.

where $\mathcal{X}$ and $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \ldots, \mathbf{x}^{(n)}$ are tensor and vectors respectively. The bracket powers denote series elements. The vectors have the size of the dimension at their position in the sequence from Eq. (2).

For an individual element in $\mathcal{X}$ at position $i_1, i_2, \ldots, i_n$ in the tensor this means that [14],

$$x_{i_1,i_2,\ldots,i_n} = x_{i_1}^{(1)} x_{i_2}^{(2)} x_{i_3}^{(3)} \ldots x_{i_n}^{(n)}. \tag{3}$$

### B. CP-Decomposition

The CP-decomposition expresses a tensor $\mathcal{X}$ as a sum of $R$ rank one tensors [14]. A three dimensional tensor requires three vectors in each rank product. Consequently

$$\mathcal{X} \approx [\mathbf{A}, \mathbf{B}, \mathbf{C}] = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \tag{4}$$

approximates the tensor. Adding an additional scaling weight $\lambda$ [14] allows normalizing the vectors to have unit length in the two-norm, we obtain

$$\mathcal{X} \approx [\mathbf{A}, \mathbf{B}, \mathbf{C}] = \sum_{r=1}^{R} \lambda_r \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r, \tag{5}$$

where $\mathcal{X}$ is the input tensor and $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$ are the used vectors representing it, see Figure 1. The matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ contain the vectors $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$ in their columns.

Multiple algorithms exist to obtain the CP-form of a tensor: Alternating Least Squares (ALS) [14], the tensor power method [35] and Non-linear Least Squares (NLS) [31]. We compare the ALS and the power method to random initialization for our networks. The ALS approach iteratively and alternatingly updates $\mathbf{A}, \mathbf{B}, \mathbf{C}$, i.e. separately each mode matrix of the tensor. The power method starts from a random initialization and relies on repeated multiplications to find the CP-decomposition. The procedure is similar to the matrix case. We refer the interested reader to [14]. Once initialized, we optimize our networks in the CP-form.

### C. Canonical Weight Normalization (CPNorm)

Instead of working with a single flat weight vector and a global length parameter, we aim to conserve the tensor structure. Figure 2 illustrates our alternative approach. In the figure, outer products appear as orthogonally arranged squares. This paper, therefore, explores a CP-weight formulation. Taking a cue from weight normalization, we introduce the parameter $\sigma$. For a $\mathbb{R}^3$ tensor we choose to represent our weights as

$$\mathcal{W} = \sigma \left( \sum_{r=1}^{R} \lambda_r \frac{\mathbf{a}_r}{\|\mathbf{a}_r\|_2} \circ \frac{\mathbf{b}_r}{\|\mathbf{b}_r\|_2} \circ \frac{\mathbf{c}_r}{\|\mathbf{c}_r\|_2} \right). \tag{6}$$

The rank scales $\lambda_r$, the parameter vectors $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$, as well as the global length $\sigma$, are all optimized. In tensor numerics for the CP-decomposition and related approaches, normalized vectors often appear in the outer vector product for numerical stability and convenience [4], [8], [14]. Regular renormalization should consequently improve stability. For an $\mathbb{R}^3$ tensor, we divide $\mathbf{a}_r$, $\mathbf{b}_r$, $\mathbf{c}_r$ by their norm after each weight update. In other words: All weight vectors are normalized after each update. Renormalization preserves their unit length.

Enforcing unit weight vectors ensures that the rank weight $\lambda_r$ scales the rank. We can now optimize the global and the per rank scales separately. Furthermore, by sorting the scales, we can truncate the sum and preserve the essential terms.

We found our weight formulation to be differentiable in PyTorch. The upcoming section will verify its stability and convergence properties empirically.

## IV. Experiments

This section describes CP-normalization and compression experiments on LeNet and AlexNet-like architectures. The implementation relies on Pytorch [28] and TensorLy [15]. We work with the MNIST [23], CIFAR-10 [17], and SVHN [25] datasets.

The CIFAR10 dataset consists of 60k images, which we split into 50k train and 10k test images. Ten different categories

(a) CIFAR10 (b) SVHN (c) MNIST

Fig. 3: Visualization of the CIFAR-10 [17], SVHN [25] and MNIST [23] data sets we used to train and evaluate our networks.

exist within each split. The networks have to identify planes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Figure 3a shows samples randomly drawn from the training set.

Cropped Street View House Numbers (SVHN) sample images appear in Figure 3b. The dataset contains 73K train and 26K test images. Ten digits from 0 to 9 have to be classified correctly in photos of house numbers obtained from Google Street View.

Finally, the MNIST dataset has 60k elements. Fifty thousand are used to train and 10 to test our networks. Similar to SVHN, handwritten digit numbers have to be classified correctly. Figure 3c shows sample digits.

We train all our networks multiple times to account for the effect of local minima in non-convex optimization. In total, all architectures in this paper are trained eight times. We report mean values $\mu$ and a single standard deviation $\pm\sigma$ every time.

### A. Evaluating Canonical Weight Normalization (CPNorm)

*1) AlexNet-CIFAR10:* This section empirically evaluates canonical normalization using an AlexNet-inspired network. Similar to the classic architecture [18] we work with five convolutional layers (kernel size-3) and three max pool layers (kernel size-2) followed by three fully connected layers. The last three fully connected layers act as a classifier for the network with dropout applied as in [18].

Application of CP-normalization to convolutional or linear layers requires at least approximate prior knowledge of the corresponding layer's rank.

Following [20], we estimate the tensor-rank of a layer by computing CP-decompositions using the alternating least squares method with increasing ranks until

$$1 - \frac{||\mathcal{W} - \bar{\mathcal{W}}||_2}{||\mathcal{W}||_2} \approx 1. \tag{7}$$

$\mathcal{W}$ represents the original tensor, and $\bar{\mathcal{W}}$ denotes the reconstruction from the CP-form. Table I lists the full rank of every layer in the AlexNet-like architecture along with every tensor shape. We estimate the ranks of the initialized tensors before training. Patterns and redundant structures are likely to appear during training. The resulting tensors will probably have a lower rank compared to the original tensor. We accept the larger estimated rank based on the random initialization, and the limited over-parameterization it causes. We believe it tends to help with initial convergence [7]. We will revisit this question in Section IV-C. Knowing the rank, we can now convert the weight tensors into their canonical form.

TABLE I: Ranks of every layer in our AlexNet-like architecture right after initialization. 4D tensors represent convolutional layers. 2D tensors represent linear layers. We iteratively estimated the ranks using the alternating least squares (ALS) algorithm.

| layer size | tensor rank |
|---|---|
| 3x64x3x3 | 36 |
| 64x192x3x3 | 571 |
| 192x384x3x3 | 1626 |
| 384x256x3x3 | 1948 |
| 256x256x3x3 | 1644 |
| 4096x1024 | 1024 |
| 1024x512 | 512 |
| 512x10 | 10 |

TABLE II: The test accuracies of an AlexNet inspired network on CIFAR10. We tabulate mean values and a single standard deviation. Experiments without normalization, with weight normalization, and with CP-normalization are compared. For the SGD experiments, we choose a learning rate of 0.01. The RMSProp optimizer ran with a learning rate of 0.001. * represents the early stopping. We find that our CP-normalization approach performs competitively.

| | | accuracy [%] | | |
|---|---|---|---|---|
| method | optimizer | max | $\mu \pm \sigma$ | weights |
| none | SGD | 88.01 | $87.06 \pm 0.36$ | $6.98 \times 10^6$ |
| weight | SGD | 88.63 | $87.38 \pm 0.37$ | $6.98 \times 10^6$ |
| CP | SGD | **89.05** | $\mathbf{88.32 \pm 0.21}$ | $9.25 \times 10^6$ |
| none* | RMSProp | 83.94 | $82.51 \pm 1.40$ | $6.98 \times 10^6$ |
| weight | RMSProp | 87.98 | $87.17 \pm 0.35$ | $6.98 \times 10^6$ |
| CP | RMSProp | **90.38** | $\mathbf{88.70 \pm 1.09}$ | $9.25 \times 10^6$ |

Having established the tensor ranks, we evaluate the stability of the new representation. To this end, we train our AlexNet-like network on CIFAR 10 for 150 epochs. We repeat identical experiments with weight normalization [30], CP-normalization (see section III-C), and without normalization using SGD and RMSprop optimizers. We also apply both weight and canonical normalization to all layers. The Stochastic Gradient Descent (SGD) optimizer ran with a learning rate of 0.01. We compare the SGD optimization result to multiple runs with an RMSprop optimizer and a learning rate of 0.001.

We found empirically that the power method provides better initializations than the alternating least squares approach. In this and all subsequent experiments, the power method will be used to initialize the weights.

Results are tabulated in Table II. For both SGD and RMSProp, our canonical formulation outperforms weight normalization and the un-normalized network. In the CP-case, we additionally observe faster convergence. Unfortunately, driving the initial approximation error of the CP-form close to zero increases the network size. We will revisit this issue in

TABLE III: Test accuracies of the AlexNet-like network performance on the SVHN dataset. Experiments with weight-normalization, CP-normalization, and without being shown. We train using SGD with a learning rate of 0.08. For the RMSProp experiments, we work with a step size of 0.0001.

| | | accuracy [%] | | |
|---|---|---|---|---|
| method | optimizer | max | $\mu \pm \sigma$ | weights |
| none | SGD | 95.39 | $94.61 \pm 0.40$ | $6.98 \times 10^6$ |
| weight | SGD | 95.43 | $94.30 \pm 0.70$ | $6.98 \times 10^6$ |
| CP | SGD | **95.62** | $\mathbf{94.82 \pm 0.29}$ | $9.25 \times 10^6$ |
| none | RMSProp | 94.76 | $94.00 \pm 0.33$ | $6.98 \times 10^6$ |
| weight | RMSProp | 94.98 | $94.06 \pm 0.32$ | $6.98 \times 10^6$ |
| CP | RMSProp | **95.03** | $\mathbf{94.52 \pm 0.30}$ | $9.25 \times 10^6$ |

TABLE IV: Ranks of every layer in our LeNet-like architecture. Convolutional layers have 4D tensors, and 2D tensor represent linear layers.

| Layer size | Tensor rank |
|---|---|
| 1x32x3x3 | 11 |
| 32x64x3x3 | 270 |
| 9216x128 | 128 |
| 128x10 | 10 |

section IV-C.

*2) AlexNet-SVHN:* We now repeat similar experiments on the Street View House Numbers (SVHN) data set. Our AlexNet-like architecture remains the same. The network is trained for 80 epochs by SGD with a learning rate of 0.08. Similarly, with RMSProp, we optimize for 80 epochs with a relatively small learning rate of 0.0001. Again the ranks from Table 1 are used to apply CP-normalization.

Table III shows the performance of our AlexNet-like architecture on the SVHN dataset. Once more, we compare weight- and CP-normalization, as well as no normalization. Once more, CP-normalization improved performance with both optimizers. Without normalization and higher learning rates, RMSprop was occasionally unstable. Both normalization approaches stabilized these runs successfully.

*3) LeNet-MNIST:* The implemented LeNet-inspired architecture has two convolutional layers (kernel size-3) followed by two fully connected layers with dropout. Just like we did for our previous experiments, we again compare the performance with SGD, and RMSProp on MNIST. We set the learning rate to 0.001 and train over 50 epochs for both optimizers. We repeat the rank estimation procedure as described in Section IV-A1, Table IV shows the ranks we measured. Table V depicts the results of LeNet over different normalization methods using SGD and RMSprop optimizers. Similar to the AlexNet-case, LeNet with CP-normalization converges to competitive mean accuracies. The parameter gain caused by the full rank CP-form is less pronounced in this case.

We show the evolution of the CP-parameters from Equation 5 in Figure 4. The $\sigma$ and $\lambda$s are all one initially. We observe that the rate of change for all parameters is initially very high. Towards the end of the training process, all values

TABLE V: Test accuracies of the LeNet-like architecture without normalization, with weight normalization, and with CP-normalization. We compare training with a SGD optimizer to training with RMSProp with a learning rate of 0.001.

| | | accuracy [%] | | |
|---|---|---|---|---|
| method | optimizer | max | $\mu \pm \sigma$ | # weights |
| none | SGD | 97.91 | $97.79 \pm 0.07$ | $1.19 \times 10^6$ |
| weight | SGD | 98.00 | $97.91 \pm 0.07$ | $1.20 \times 10^6$ |
| CP | SGD | **98.77** | $98.66 \pm 0.05$ | $1.22 \times 10^6$ |
| none | RMSProp | 99.33 | $99.10 \pm 0.05$ | $1.19 \times 10^6$ |
| weight | RMSProp | **99.40** | $99.30 \pm 0.04$ | $1.20 \times 10^6$ |
| CP | RMSProp | 99.35 | $99.21 \pm 0.05$ | $1.22 \times 10^6$ |

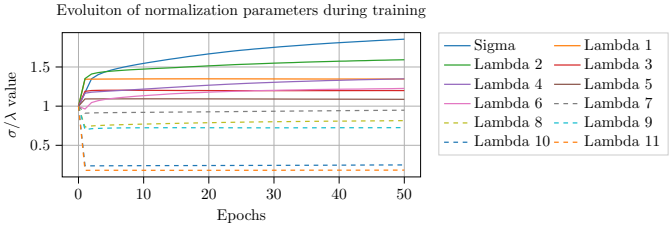


Fig. 4: Plot of the progression of $\sigma$ and $\lambda$'s during training for the first convolutional layer in the LeNet-like architecture. The first layer has a tensor shape of 1x32x3x3 and a rank of 11. Solid lines portray weights with positive growth. We use dashed lines for $\lambda$ values with negative growth.

cease to change. We conclude that gradients are applied, and backpropagation is successful. Figure 4 also displays significant differences between the various rank weights, a prerequisite for our truncation approach to be meaningful.

### *B. Random Initialization of the CP-Form*

All networks started in an initialized tensor form thus far. Running the power method converted the original tensor into the CP-form. Evaluation of the power method causes additional overhead. This section will explore alternatives to reduce the computational cost. Working with the setup described in Section IV-A1, we now directly populate the CP-form with random values drawn from standardized distributions.

Table I shows layer sizes of all AlexNet-Layers. Since we store the individual vectors in matrices, the four-dimensional convolutional layers will have four matrices $\mathbf{A}, \mathbf{D}, \mathbf{C}, \mathbf{D}$ per layer. For the fully connected layers, we have two matrices $\mathbf{A}$, and $\mathbf{B}$. Since the rank of each layer determines the shape, we can construct the CP-shape directly and initialize the matrices using Kaiming normal or uniform distributions. In a first series of experiments we set $\lambda_r = 1$ for all ranks $r$ initially. Table VI compares direct initialization with Kaiming normal or uniformly distributed values to initialization through the CP-decomposition via the power method. We find the performance with stochastic gradient descent competitive.

An additional series of experiments explores normal instead of constant initialization for the rank-scales $\lambda$. Standard SGD worked better with initialization to ones. The Adam-optimizer,

TABLE VI: Test accuracies of the AlexNet-like network performance on the CIFAR10 dataset with various initializations. We train using SGD with a learning rate of 0.01 and initially $\lambda_r = 1$ for all ranks.

| | accuracy [%] | |
|---|---|---|
| factor initialization | max | $\mu \pm \sigma$ |
| CP-decomposition | 89.05 | $88.32 \pm 0.21$ |
| Kaiming normal | 88.95 | $87.76 \pm 0.36$ |
| Kaiming uniform | 88.68 | $87.63 \pm 0.43$ |

TABLE VII: Test accuracies of the AlexNet-like network performance on the CIFAR10 dataset with various initializations. We train using ADAM with a learning rate of 0.001. A normal distribution $\mathcal{N}(0, 1)$ initialized the rank-scales $\lambda$.

| | accuracy [%] | |
|---|---|---|
| factor initialization | max | $\mu \pm \sigma$ |
| CP-decomposition | 90.27 | $89.59 \pm 0.21$ |
| Kaiming normal | 90.61 | $89.70 \pm 0.37$ |
| Kaiming uniform | 90.62 | $89.74 \pm 0.35$ |

however, works with the standard normal initialization. Results are shown in Table VII. We observe a slightly improved performance in comparison to what we saw in Table VI. Previously initialization by running the power method worked slightly better. Now the maximum values are higher for the normally or uniformly initialized CP-forms. Since the standard deviations indicate that the differences are not significant, we conclude that running the power method is not required when using Adam. The progression of $\lambda$ distributions during training is shown in Figures 5 and 6. Encouragingly Adam pushes the scales towards the sides away from zero, as we would expect in a working system.

### C. Network Compression

Re-parameterizing the network in a CP-form, allows easy compression by truncating the rank sums. We compress our networks by removing the smallest rank scale lambdas and their corresponding vectors from the decomposition. The following experiments were conducted on the AlexNet and LeNet-like architectures (as discussed in the previous section) with various compression rates.

*1) AlexNet-CIFAR10:* Before compression, the full-rank networks are trained and stored. We measure compression with respect to the best fit CP-normalized network. 0% means that we are working with every outer product. We compress our networks by sorting the CP-summands for each rank according to their $\lambda_r$ weights. After sorting, we discard, for example, the 25% least important CP-summands. All summands have the same amount of parameters. Therefore, in this case, 75% of parameters are retained.

The compressed networks are fine-tuned for 20 epochs with SGD to compensate for the truncation. We use SGD for all fine-tuning as RMSprop was unstable in some cases. During tuning, we observed over-fitting problems depending on the

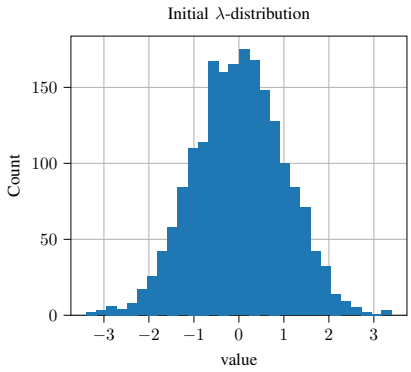


Fig. 5: Histograms depicting initial the initially normal $\mathcal{N}(0, 1)$ distribution of rank-scales ($\lambda$). The histogram shows the fourth layer of our AlexNet-architecture, before a gradient update has been applied.

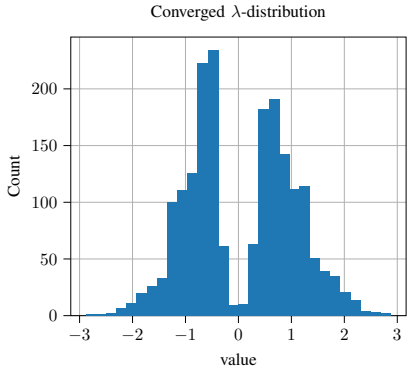


Fig. 6: Histograms showing the final distribution of the $\lambda$s from Plot 5. As the optimization process converges, we observe significant movement away from the center towards the sides.

TABLE VIII: Sum truncation for all layers of the AlexNet-like architecture on CIFAR-10. Various compression rates and the resulting accuracies are tabulated. The compressed networks have been fine-tuned using SGD with variety of different learning rates, which are specified in brackets.

| | | accuracy [%] | | |
|---|---|---|---|---|
| compression | learning rate | max | $\mu \pm \sigma$ | # weights |
| 0% | 0.001 | 89.05 | $88.32 \pm 0.21$ | $9.25 \times 10^6$ |
| 25% | 0.0001 | 89.12 | $89.10 \pm 0.02$ | $6.93 \times 10^6$ |
| 50% | 0.001 | 87.93 | $87.79 \pm 0.05$ | $4.62 \times 10^6$ |
| 75% | 0.01 | 81.61 | $81.25 \pm 0.28$ | $2.31 \times 10^6$ |

learning rates. In response, we chose the learning rates based on the validation accuracy after 20 epochs. For twenty-five percent compression, we work with a learning rate of 0.0001, which is much smaller than the optimization step size of the initial training. In the 50% compression case, a learning rate of 0.001 is used. For 75% compression, we chose an even higher learning rate of 0.01 for faster convergence. As the distance to the original network weights increases, learning rates become useful. We find this relationship intuitive since the distance

TABLE IX: Compression performance comparison between Tai compression and CP-compression. Here, we only consider compression of the convolutional layers and give the number of weights for these. All the convolutional layers in AlexNet except the first one are compressed. Compression is performed by fine tuning over 20 epochs. This is an important difference to the experiments run for table VIII.

| method | compression | accuracy [%] | # weights |
|---|---|---|---|
| low-rank [33] | 0% | 85.11 | $2.25 \times 10^6$ |
| canonical (ours) | 0% | 85.96 | $3.21 \times 10^6$ |
| low-rank [33] | 89.78% | $83.06 \pm 1.02$ | $0.23 \times 10^6$ |
| canonical (ours) | 90.91% | $81.82 \pm 0.52$ | $0.20 \times 10^6$ |

we must travel in weight space to compensate for the missing parameters increases.

Table VIII contains the compression results for our AlexNet-like structure. The number of parameters at 25% compression and the corresponding accuracy is particularly significant. Here, the number of parameters approximately equals those of the weight-normalized network. The performance improves in comparison to the full rank or perfect fit parametrization, Table II. We conclude that a near-perfect fit is not required in this case.

With half of the parameters, we observe a $\approx 1\%$ accuracy drop compared to the full rank CP-normalized network. Finally, we cut the CP-sum short after the first quarter, effectively removing 75% of all parameters. At the same time, this drastic parameter cut results in only a $\approx 7\%$ accuracy drop.

Note, the 25% and 50% compressed networks outperform the weight normalized and un-normalized networks shown in Table II in terms of mean accuracy, the version with only 50% of the CP-summands does so with significantly fewer parameters.

*2) Comparison to Tai et al. [33]:* Table IX compares the compression performance of the CP-form and the low rank-formulation proposed in [33]. We work with the AlexNet-like network from section IV-C1 on CIFAR10. To challenge both methods, we aim for compression rates of approximately $90\%$. During the fine-tuning [33] employs batch normalization, while our CP-form does not. We observe competitive performance for our approach, with slightly fewer parameters than our re-implementation of [33]. The method proposed in this paper allows compression of convolutional and fully connected layers. Since [33] do not consider dense layers, we limit ourselves to convolutional layers here for a fair comparison. As [33] chose to work with Lua, we re-implemented their approach in PyTorch. Our source code is made available.

*3) Compression Performance on SVHN:* We now move to the compression of the network resulting from our SVHN experiments. Accuracies, compression-rates as well as parameter counts appear in Table X. We use the same compression rates as in the case of CIFAR10 and again fine-tune for 20 epochs. Once more, we find increasing learning rates more helpful when networks are compressed more aggressively.

On SVHN, we find that removing the last quarter and half of the CP-sum improves our mean results. Going further and removing three-quarters of the network parameters has a detrimental effect on the network accuracy. We deduce that the network initially had more parameters than required. We can classify the SVHN-digits with fewer weights. In this case, we settle with the upper half of the original summands.

TABLE X: Compression results on AlexNet like architecture on SVHN with various compression rates using a SGD optimizer with variety of learning rates specified in brackets.

| | | accuracy [%] | | |
|---|---|---|---|---|
| compression | learning rate | max | $\mu \pm \sigma$ | # weights |
| 0% | 0.0001 | 95.62 | $94.82 \pm 0.30$ | $9.25 \times 10^6$ |
| 25% | 0.0001 | 95.30 | $95.27 \pm 0.01$ | $6.93 \times 10^6$ |
| 50% | 0.001 | 95.42 | $95.36 \pm 0.03$ | $4.62 \times 10^6$ |
| 75% | 0.01 | 94.19 | $93.75 \pm 0.14$ | $2.31 \times 10^6$ |

TABLE XI: Compression results on LeNet like architecture with various compression rates using a RMSprop optimizer with a learning rate of 0.001.

| | accuracy [%] | | |
|---|---|---|---|
| compression | max | $\mu \pm \sigma$ | # weights |
| 0% | 99.35 | $99.21 \pm 0.05$ | $1.22 \times 10^6$ |
| 10% | 99.23 | $99.18 \pm 0.01$ | $1.10 \times 10^6$ |
| 25% | 98.83 | $98.80 \pm 0.02$ | $0.91 \times 10^6$ |
| 50% | 97.56 | $97.55 \pm 0.02$ | $0.61 \times 10^6$ |

The best performing 50% CP-normalized network improves upon the weight normalized and not normalized networks we saw in Table III in terms of both mean accuracy and parameters.

*4) LeNet-MNIST:* Finally, we compress the LeNet-like architecture we trained on MNIST with RMSprop and a learning rate of 0.001.

We remove the lower 10%, 25%, and 50% of the summands in the CP-form and collect the resulting classification accuracies in Table XI. At a 10% compression rate, the resulting mean accuracy drop we observe is fairly small 0.01% . At 25% compression, a comparatively small drop of $\approx 0.5\%$. In this case, we can not afford to discard half of the parameters since we find the performance loss significant.

## V. Conclusion

In this paper, we proposed to express network weights as a weighted sum of normalized outer vector products. Computing a canonical/parallel factor decomposition allows a direct network initialization using well-known, established approaches. We evaluated the newly formulated method experimentally and found it to rival weight normalization in terms of network convergence and stability. In contrast to weight normalization, the canonical approach simplifies network compression after training. We tested initializing the CP-form directly. Our evidence suggests that standard initialization techniques can replace the power method for network initialization.

### A. Future Work

In hindsight, we would have additionally included lower ranks during our initial training. Given the encouraging compression results, this is certainly an idea we recommend for future work. An adaptive compression based on the size of the rank weights $\lambda_r$ should also be investigated.

## REFERENCES


[1] Alexandre Araujo, Benjamin Negrevergne, Yann Chevaleyre, and Jamal Atif. Training compact deep learning models for video classification using circulant matrices. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 271–286. Springer International Publishing, 2019.

[2] Marcella Astrid and Seung-Ik Lee. Cp-decomposition with tensor power method for convolutional neural networks compression. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 115–118. IEEE, 2017.

[3] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. *Advances in neural information processing systems*, 26:3084–3092, 2013.

[4] Gregory Beylkin, Jochen Garcke, and Martin J. Mohlenkamp. Multivariate regression and machine learning with sums of separable functions. *SIAM Journal on Scientific Computing*, 31(3):1840–1857, 2009.

[5] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[6] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[7] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[8] J. Garcke. Classification with sums of separable functions. In José Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *ECML PKDD 2010*, pages 458–473, 2010.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, Berlin, Heidelberg, 2021.

[11] Song Han, Huizi Mao, and W. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 448–456. PMLR, 2015.

[13] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In Michel François Valstar, Andrew P. French, and Tony P. Pridmore, editors, *British Machine Vision Conference, BMVC 2014, Nottingham, UK, September 1-5, 2014*. BMVA Press, 2014.

[14] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[15] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019.

[16] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105. Curran Associates, Inc., 2012.

[19] Andrey Kuzmin, Markus Nagel, Saurabh Pitre, Sandeep Pendyam, Tijmen Blankevoort, and Max Welling. Taxonomy and Evaluation of Structured Compression of Convolutional Neural Networks. 2019.

[20] Brett W Larsen and Tamara G Kolda. Practical leverage-based sampling for low-rank tensor decomposition. *arXiv preprint arXiv:2006.16438*, 2020.

[21] V. Lebedev and V. Lempitsky. Fast convnets using group-wise brain damage. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2554–2564, 2016.

[22] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In Yoshua Bengio and Yann LeCun, editors, *ICLR 2015*, 2015.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] J. Ma, Qiuchen Zhang, Joyce Ho, and Li Xiong. Spatio-temporal tensor sketching via adaptive sampling. In *ECML/PKDD*, 2020.

[25] Yuval Netzer, Tiejie Wang, Adam Coates, A. Bissacco, Bo Wu, and A. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[26] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. Tensorizing neural networks. In *NIPS*, 2015.

[27] Anton Obukhov, Maxim Rakhuba, Alexander Liniger, Zhiwu Huang, Stamatios Georgoulis, Dengxin Dai, and Luc Van Gool. Spectral tensor train parameterization of deep learning layers. In *International Conference on Artificial Intelligence and Statistics*, pages 3547–3555. PMLR, 2021.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS Workshop Autodiff*, 2017.

[29] Anh Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavský, Valeriy Glukhov, Ivan V. Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXIX*, volume 12374 of *Lecture Notes in Computer Science*, pages 522–539. Springer, 2020.

[30] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[31] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. Tensorlab v2.0. *https://www.tensorlab.net/*, 2014.

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[33] Cheng Tai, Tong Xiao, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[34] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4451–4458. IEEE, 2017.

[35] Y. Wang, H. F. Tung, Alex Smola, and Anima Anandkumar. Fast and guaranteed tensor decomposition via sketching. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[36] Moritz Wolter, Shaohui Lin, and Angela Yao. Neural network compression via learnable wavelet transforms. In *29th International Conference on Artificial Neural Networks*. Springer, 2020.

[37] Bijiao Wu, Dingheng Wang, Guangshe Zhao, Lei Deng, and Guoqi Li. Hybrid tensor decomposition in neural network compression. *Neural Networks*, 132:309–320, 2020.

[38] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1476–1483, 2015.

[39] Matthew D. Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Conference Track Proceedings*, 2013.