Low-rank lottery tickets: finding efficient low-rank neural networks via matrix differential equations

Anonymous Author(s)
Affiliation
Address
email

Abstract

Neural networks have achieved tremendous success in a large variety of applications. However, their memory footprint and computational demand can render them impractical in application settings with limited hardware or energy resources. In this work, we propose a novel algorithm to find efficient low-rank subnetworks. Remarkably, these subnetworks are determined and adapted already during the training phase and the overall time and memory resources required by both training and evaluating them are significantly reduced. The main idea is to restrict the weight matrices to a low-rank manifold and to update the low-rank factors rather than the full matrix during training. To derive training updates that are restricted to the prescribed manifold, we employ techniques from dynamic model order reduction for matrix differential equations. Moreover, our method automatically and dynamically adapts the ranks during training to achieve the desired approximation accuracy. The efficiency of the proposed method is demonstrated through a variety of numerical experiments on fully-connected and convolutional networks.

1 Introduction

While showing great performance in terms of classification records, most state-of-the-art neural networks require an enormous amount of computation and memory storage both for the training and the evaluation phases [26]. These requirements not only increase infrastructure costs and energy consumption, but also make the deployment of artificial neural networks to infrastructures with limited resources such as mobile phones or smart devices prohibitive. On the other hand, it is well-known that networks' weights contain structures and redundancies that can be exploited for reducing the parameter space dimension without significantly affecting the overall accuracy [4, 10, 18].

Network pruning is a popular line of research that addresses this problem by removing redundant parameters from pre-trained models. Typically, the initial network is large and accurate, and the goal is to produce a smaller network with similar accuracy. Methods within this area include weight sparsification [21, 24, 46] and quantization [11, 57], with different pruning techniques, including search-based heuristics [24], reinforcement learning [3, 23] and genetic algorithms [41]. More recent work has considered pruning during training, by formulating pruning as a data-driven optimization problem [21, 25, 26]. The resulting "dynamical pruning" boils down to a parameter-constrained training phase which, however, has been mostly focused on requiring sparse or binary weights so far.

Rather than enforcing sparsity or binary variables, in this work we constrain the parameter space to the manifold of low-rank matrices. Neural networks' parameter matrices and large data matrices in general are seldom full rank [16, 45, 50, 53]. Constraining these parameters to lie on a manifold defined by low-rank matrices is thus a quite natural approach. By interpreting the training problem as a continuous-time gradient flow, we propose a training algorithm based on the extension of recent Dynamical Low-Rank Approximation (DLRA) algorithms [5, 6, 7]. This approach allows us to

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

use low-rank numerical integrators for matrix Ordinary Differential Equations (ODEs) to obtain modified forward and backward training phases that only use the small-rank factors in the low-rank representation of the parameter matrices.

We perform several experimental evaluations on fully-connected and convolutional networks showing that the resulting dynamical low-rank training paradigm yields low-parametric neural network architectures which compared to their full-rank counterparts are both remarkably less-demanding in terms of memory storage and require much less computational cost to be trained. Moreover, the trained low-rank neural networks achieve comparable accuracy to the original full architecture. This observation is reminiscent of the so-called lottery tickets hypothesis — dense neural networks contain sparse subnetworks that achieve high accuracy [18] — and suggests the presence of *low-rank winning tickets*: highly-performing low-rank subnetworks of dense networks. Remarkably, our dynamical low-rank training strategy seems to be able to find the low-rank winning tickets directly during the training phase independent of initialization.

2 Related work on low-rank methods

Low-rank factorization using the SVD and other matrix decomposition techniques have been extensively studied in the scientific computing and machine learning communities. The challenge of compressing and speeding up large-scale neural networks using low-rank methods has sparked wide-spread research interest in recent years and significant effort has been put towards developing low-rank factorization strategies for deep neural networks.

Previous works can roughly be categorized in approaches with fixed low rank and variable low rank during training time. Fixed rank approaches decompose weight matrices using SVD or tensor decompositions of pre-trained networks and fine-tune the factorized network [13, 37, 48, 52], constrain weight matrices to have a fixed low rank during training [29], or create layers as a linear combination of layers of different rank [28]. Hence, these methods introduce the rank of the matrix decomposition as another hyperparameter to be fine-tuned. Rank-adaptive methods mitigate this issue by automatic determination and adaption of the low-rank structure after training. In particular, [32, 33] apply heuristics to determine the rank of the matrix decomposition ahead of time, whereas [56] encourages low-rank weight matrices by means of a penalized loss that depends on approximated matrix ranks.

Few methods have been proposed recently that adapt the ranks of the weight matrix alongside the main network training phase. In [39], the authors set up the neural network training as a constrained optimization problem with an upper bound on the ranks of the weights, which is solved in an alternating approach resulting in an NP-hard mixed integer program. The authors of [27] formulate a similar constrained optimization problem resulting in a mixed discrete-continuous optimization scheme which jointly addresses the ranks and the elements of the matrices. However, both these approaches require knowledge of the full weight matrix (and of its singular value decomposition) during training and overall are more computational demanding than standard training.

In this work we overcome the above issues and propose a training algorithm with reduced memory and computational requirements. To this end, we reinterpret the optimization problem of a neural network as a gradient flow of the network weight matrices and thus as a matrix ODE. This continuous formulation allows us to use recent advances in DLRA methods for matrix ODEs which aim at evolving the solution of the differential equation on a low-rank manifold. The main idea of DLRA [34], which originates from the Dirac-Frenkel variational principle [15, 19], is to approximate the solution through a low-rank factorization and derive evolution equations for the individual factors. Thereby, the full solution does not need to be stored and the computational costs can be significantly reduced. To ensure robustness of the method, stable integrators have been proposed in [42] and [7]. Instead of evolving individual low-rank factors in time, these methods evolve products of low-rank factors, which yields remarkable stability and exactness properties [31], both in the matrix and the tensor settings [9, 35, 43, 44]. In this work, we employ the "unconventional" basis update & Galerkin step integrator [7] as well as its rank-adaptive extension [5], see also [8, 36]. The rank-adaptive unconventional integrator chooses the approximation ranks according to the continuous-time training dynamics and allows us to find highly-performing low-rank subnetworks directly during the training phase, while requiring reduced training cost and memory storage.

3 Low-rank training via gradient flow

Consider a feed-forward fully-connected neural network $\mathcal{N}(x) = z_M$, with $z_0 = x \in \mathbb{R}^{n_0}$, $z_k = \sigma_k(W_k z_{k-1} + b_k) \in \mathbb{R}^{n_k}$, $k = 1, \dots, M$ (the convolutional setting is discussed in the supplementary material §6.8). We consider the training of $\mathcal{N}$ based on the optimization of a loss function $\mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y)$ by means of a gradient-based descent algorithm. For example, when using gradient descent, the weights of $\mathcal{N}$ at iteration $t \in \mathbb{N}$ are updated via

$$W_k^{t+1} = W_k^t - \lambda \nabla_{W_k} \mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y) \qquad \forall k = 1, \dots, M \tag{1}$$

with a learning rate λ. When the weight matrices W_k are full, both the forward and gradient evaluations of the network require a large number of full matrix multiplications, with high computational expense and large memory footprint. This renders the training and the use of large-scale neural networks a difficult challenge on limited-resource devices. At the same time, a wealth of evidence shows that dense networks are typically overparameterized and that most of the weights learned this way are unnecessary [16, 45]. In order to reduce the memory and computation costs of training, we propose a method that performs the minimization over the manifold of low-rank matrices.

To this end, we assume that the ideal W_k can be well-approximated by a matrix of rank $r_k \ll n_k, n_{k+1}$ of the form $U_k S_k V_k^\top \in \mathbb{R}^{n_k \times n_{k-1}}$, where $U_k \in \mathbb{R}^{n_k \times r_k}$, $V_k \in \mathbb{R}^{n_{k-1} \times r_k}$ are thin and tall matrices having orthonormal columns spanning optimal subspaces which capture essential properties of parameters, and $S_k \in \mathbb{R}^{r_k \times r_k}$ is a tiny full-rank matrix that allows us to extrapolate the useful information from the learned subspaces U_k and V_k.

Traditional descent algorithms such as (1) do not guarantee the preservation of the low-rank structure $U_k S_k V_k^\top$ when updating the weights during training and require knowledge of the whole W_k rather than the factors U_k, S_k, V_k. To overcome these limitations, we move to a gradient flow formulation of the loss-minimization problem, and we derive a robust and efficient training method that preserves the low-rank structure and works only with the factors U_k, S_k, V_k, rather than their product.

Minimizing the loss function with respect to the k-th weight matrix is equivalent to evaluating the long time behaviour of the matrix ODE

$$\dot{W}_k(t) = -\nabla_{W_k} \mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y), \tag{2}$$

where the "dot" denotes the time-derivative. Thus, using (2) we can interpret the training phase as a continuous evolution of the weights, rather than a discrete update. Let $\mathcal{M}_{r_k}$ denote the manifold of matrices with rank r_k and assume at a certain time t_0 the weights are in the manifold, i.e., $W_k(t_0) \in \mathcal{M}_{r_k}$. Using this continuous-time interpretation allows us to derive a strategy to evolve the weights according to the dynamics in (2) so that $W_k(t) \in \mathcal{M}_{r_k}$ for all $t \geq t_0$. To this end, in §3.1 we exploit the fact that $W_k(t)$ admits a time-dependent factorization [14] $W_k(t) = U_k(t) S_k(t) V_k(t)^\top$ to rewrite (2) as a system of matrix ODEs for each of the individual factors U_k,S_k and V_k. Then, in §4 we propose an algorithm to efficiently integrate the system of ODEs. We show in §4.1 that such algorithm reduces the loss monotonically and is accurate, in the sense that $U_k S_k V_k^\top \approx W_k$, i.e. the learned low-dimensional subspaces U_k and V_k well-match the behaviour of the full-rank network W_k solution of (2), through the action of the learned S_k. Remarkably, using the dynamics of the individual factors will also allow us to adaptively adjust the rank r_k throughout the continuous-time training process.

3.1 Coupled dynamics of the low-rank factors via DLRA

We consider the dynamical system of a single weight matrix W_k, while the remaining weight matrices are fixed in time and are treated as parameters for the gradient. In the following, we omit writing theses parameters down for efficiency of exposition. Assuming $W_k(t) \in \mathcal{M}_{r_k}$, we can formulate (2) as

$$\min \left\{ \|\dot{W}_k(t) + \nabla_{W_k} \mathcal{L}(W_k(t))\|_F \,:\, \dot{W}_k(t) \in \mathcal{T}_{W_k(t)} \mathcal{M}_{r_k} \right\} \tag{3}$$

where $\mathcal{T}_{W_k(t)} \mathcal{M}_{r_k}$ is the tangent space of $\mathcal{M}_{r_k}$ at position $W_k(t)$. In order to solve (3), we further observe that (3) can be equivalently formulated as the following Galerkin condition [34]:

$$\langle \dot{W}_k(t) + \nabla_{W_k} \mathcal{L}(W_k(t)), \delta W_k \rangle = 0 \qquad \forall \delta W_k \in \mathcal{T}_{W_k(t)} \mathcal{M}_{r_k} \,. \tag{4}$$

From $W_k = U_k S_k V_k^\top$, a generic element δW_k of the tangent space $\mathcal{T}_{W_k(t)}\mathcal{M}_{r_k}$ can be written as

$$\delta W_k = \delta U_k S_k V_k^\top + U_k \delta S_k V_k^\top + U_k S_k \delta V_k^\top \ ,$$

where δU_k and δV_k are generic elements of the tangent space of the Stiefel manifold with r_k orthonormal columns at the points U_k and V_k, respectively, and δS_k is a generic $r_k \times r_k$ matrix, see e.g. [34, §2] for details. Additionally, the Gauge conditions $U_k^\top \delta U_k = 0$ and $V_k^\top \delta V_k = 0$ must be imposed to ensure orthogonality of the basis matrices, and the uniqueness of the representation of the tangent space elements. Similarly, by the chain rule applied several times we have

$$\dot{W}_k = \frac{d}{dt}\{U_k S_k V_k^\top\} = \dot{U}_k S_k V_k^\top + U_k \dot{S}_k V_k^\top + U_k S_k \dot{V}_k^\top \ .$$

Now, the Galerkin condition (4) becomes

$$\langle \dot{U}_k S_k V_k^\top + U_k \dot{S}_k V_k^\top + U_k S_k \dot{V}_k^\top + \nabla_{W_k}\mathcal{L}(W_k(t)), \delta W_k \rangle = 0, \qquad \forall \delta W_k \in \mathcal{T}_{W_k(t)}\mathcal{M}_{r_k} \quad (5)$$

with $U_k^\top \dot{U}_k = 0$ and $V_k^\top \dot{V}_k = 0$. If we choose $\delta W_k = U_k \delta S_k V_k^\top$ in (5), we obtain

$$\langle U_k^\top \dot{U}_k S_k V_k^\top V_k + U_k^\top U_k \dot{S}_k V_k^\top V_k + U_k^\top U_k S_k \dot{V}_k^\top V_k + U_k^\top \nabla_{W_k}\mathcal{L}(W_k(t)) V_k, \delta S_k \rangle = 0 \, .$$

Thus, using the Gauge conditions, we obtain $\langle \dot{S}_k + U_k^\top \nabla_{W_k}\mathcal{L}(W_k(t))V_k, \delta S_k \rangle = 0$, which has to hold for a generic $r_k \times r_k$ matrix δS_k. We obtain this way an evolution equation for the $S_k(t)$ factor. Similarly, specifying (5) for the two choices $\delta W_k = \delta U_k S_k V_k^\top$ and $\delta W_k = U_k S_k \delta V_k^\top$, allows us to obtain the following system of differential equations for the individual factors of W_k:

$$\begin{cases} \dot{S}_k = -U_k^\top \nabla_{W_k}\mathcal{L}(W_k(t))V_k \ , \\ \dot{U}_k = -(I - U_k U_k^\top)\nabla_{W_k}\mathcal{L}(W_k(t))V_k S_k^{-1} \ , \\ \dot{V}_k = -(I - V_k V_k^\top)\nabla_{W_k}\mathcal{L}(W_k(t))^\top U_k S_k^{-\top} \ . \end{cases} \quad (6)$$

4 KLS-based training algorithm

In order to perform an efficient and robust rank-constrained training step, we numerically integrate the system of ODEs (6). Our approach is based on the "unconventional KLS integrator" [7] and its rank-adaptive version [5]. The pseudocode of the proposed training strategy is presented in Algorithm 1.

The main idea of the KLS algorithm is to alternately represent the product $W_k = U_k S_k V_k^\top$ as $W_k = K_k V_k^\top$ and $W_k = U_k L_k^\top$, consider the corresponding coupled ODEs from (6), and then perform three main steps:

1,2. **K&L-steps** (in parallel). Update the current K_k and L_k by integrating the differential equations

$$\begin{cases} \dot{K}_k(t) = -\nabla_{W_k}\mathcal{L}(K_k(t)V_k^\top)V_k, \quad K_k(0) = U_k S_k, \\ \dot{L}_k(t) = -\nabla_{W_k}\mathcal{L}(U_k L_k(t)^\top)^\top U_k, \quad L_k(0) = V_k S_k^\top, \end{cases} \quad (7)$$

from $t = 0$ to $t = \eta$; then form new orthonormal basis matrices $\widetilde{U}_k$ and $\widetilde{V}_k$ spanning the range of the computed $K_k(\eta)$ and $L_k(\eta)$.

3. **S-step**. Update the current S_k by integrating the differential equation

$$\dot{S}_k(t) = -\widetilde{U}_k^\top \nabla_{W_k}\mathcal{L}(\widetilde{U}_k S_k(t) \widetilde{V}_k^\top)\widetilde{V}_k \quad (8)$$

from $t = 0$ to $t = \eta$, with initial value condition $S_k(0) = \widetilde{U}_k^\top U_k S_k V_k^\top \widetilde{V}_k$.

An important feature of this algorithm is that it can be extended to rank adaptivity in a relatively straightforward manner [5], letting us dynamically evolve the rank of S_k (and thus the rank of W_k) during the computation. This is particularly useful, as we may expect the weight matrices to have low ranks but we may not know what the "best" ranks for each layer are. Typically, dynamically adapting the ranks of a low-rank optimization scheme is a challenging problem as moving from the manifold $\mathcal{M}_{r_k}$ to $\mathcal{M}_{r_k \pm 1}$ introduces singular points [2, 20]. Instead, treating the training problem as a system of matrix differential equations allows us to overcome this issue with a simple trick: at each step of the KLS integrator we double the dimension of the basis matrices $\widetilde{U}_k$ and $\widetilde{V}_k$ computed in the K- and

L-steps by computing orthonormal bases spanning $[K_k(\eta) \mid U_k]$ and $[L_k(\eta) \mid V_k]$, respectively, i.e. by augmenting the current basis with the basis computed in the previous time step. Then, after the new S_k matrix is computed via the S-step, a truncation step is performed, removing from the newly computed S_k matrix all the singular values that are under a certain threshold ϑ.

Of course, adding the rank-adaptivity to the integrator comes at a cost. In that case, each step requires to perform an SVD decomposition of twice the size of the current rank of S_k in order to be able to threshold the singular values. Moreover, the dimension of the bases U_k and V_k may grow, which also may require additional computational effort. However, if the ranks remain small throughout the dynamics, this computational overhead is negligible, as we will further discuss in §4.3 and §5.

4.1 Error analysis and convergence

In this section we present our main theoretical results, showing that (a) the low-rank matrices $U_k S_k V_k^\top$ formed by the weights' factors computed with Alg. 1 are close to the true solution of (2), and (b) that the loss function decreases during DLRT, provided the singular value threshold ϑ is not too large, i.e., is bounded by a constant times the square of the time-step size η (see Theorem 1). In the version we present here, part of the statements are presented in an informal way for the sake of brevity. We refer to the supplementary material §6.1 for details and for the proofs.

Assume the gradient flow $\mathcal{F}_k(Z) = -\nabla_{W_k}\mathcal{L}(W_1, \dots, Z, \dots, W_M, \mathcal{N}(x), y)$ in (2) is locally bounded and locally Lipschitz continuous, with constants C_1 and C_2, respectively. Then,

Theorem 1. *Fixed x and y, let $W_k(t)$ be the (full-rank) continuous-time solution of* (2) *and let U_k, S_k, V_k be the factors computed with Algorithm 1 after t steps. Assume that the K,L,S steps* (7) *and* (8) *are integrated exactly from* 0 *to η. Assume moreover that, for any $Z \in \mathcal{M}_{r_k}$ sufficiently close to $W_k(t\eta)$, the whole gradient flow $\mathcal{F}_k(Z)$ is "ε-close" to $\mathcal{M}_{r_k}$. Then,*

$$\|U_k S_k V_k^\top - W_k(t\eta)\|_F \leq c_1\varepsilon + c_2\eta + c_3\vartheta/\eta \qquad k = 1, \dots, M$$

where the constants c_1, c_2 and c_3 depend only on C_1 and C_2. In particular, the approximation bound does not depend on the singular values of the exact nor the approximate solution.

Observe that, while the loss function $\mathcal{L}$ decreases monotonically along any continuous-time solution $W_k(t)$ of (2), it is not obvious that the loss decreases when the integration is done onto the low-rank manifold via Algorithm 1. The next result shows that this is indeed the case, up to terms of the order of the truncation tolerance ϑ. More precisely, we have

Theorem 2. *Let $W_k^t = U_k^t S_k^t (V_k^t)^\top$ be the low rank weight matrix computed at step t of Algorithm 1 and let $\mathcal{L}(t) = \mathcal{L}(W_1^t, \dots, W_M^t, \mathcal{N}(x), y)$. Then, for a small enough time-step η we have*

$$\mathcal{L}(t+1) \leq \mathcal{L}(t) - \alpha\eta + \beta\vartheta$$

where α and β are positive constants that do not depend on t, η and ϑ.

4.2 Efficient implementation of the gradients

All the three K,L,S-steps require the evaluation of the gradient flow of the loss function with respect to the whole weight matrix W_k. Different approaches to compute this gradient in an efficient way may be used. The strategy we discuss below aims at reducing memory requirements and computational costs by avoiding the computation of the full gradient $\nabla_{W_k}\mathcal{L}$ and instead working with the gradient with respect to the low-rank factors.

To this end, we note that for the K-step it holds $\nabla_{W_k}\mathcal{L}(K_k(t)V_k^\top)V_k = \nabla_{K_k}\mathcal{L}(K_k(t)V_k^\top)$. Hence, the whole gradient can be computed through a forward run of the network with respect to K_k

$$z_k = \sigma_k\left(K_k(t)V_k^\top z_{k-1} + b_k\right), \qquad k = 1, \dots, M \tag{9}$$

and taping the gradient with respect to K_k. In this way, the full gradient does not need to be computed and the overall computational costs are comprised of running a forward evaluation while taping gradients with respect to K_k, analogously to the traditional back-propagation algorithm. The L- and S-steps can be evaluated efficiently in the same manner, by evaluating the network while taping the gradients with respect to L_k and S_k, respectively. Hence, instead of a single gradient tape (or chain rule evaluation) of the full weight matrix network, we have three gradient tapes, one for each low rank step, whose combined computational footprint is less than the full matrix tape. We provide detailed formulas for all the three gradient tapes in the supplementary material §6.7.

Algorithm 1: Dynamic Low Rank Training Scheme (DLRT)

Input : Initial low-rank factors $S_k^0 \sim r_k^0 \times r_k^0$; $U_k^0 \sim n_k \times r_k^0$;$V_k^0 \sim n_{k-1} \times r_k^0$ for $k = 1, \dots, M$;
`iter`: maximal number of descent iterations per epoch;
`adaptive`: Boolean flag that decides whether or not to dynamically update the ranks;
ϑ: singular value threshold for adaptive procedure.

1 **for** *each* `epoch` **do**
2 **for** $t = 0$ *to* $t =$ `iter` **do**
3   **for** *each layer* k **do**
4    $K_k^t \leftarrow U_k^t S_k^t$ `/* K-step */`
5    $K_k^{t+1} \leftarrow$ one-step-integrate$\big\{\dot{K}(t) = -\nabla_K \mathcal{L}(K(t)(V_k^t)^\top z_{k-1} + b_k^t), K(0) = K_k^t\big\}$
6    $L_k^t \leftarrow V_k^t (S_k^t)^\top$ `/* L-step */`
7    $L_k^{t+1} \leftarrow$ one-step-integrate$\big\{\dot{L}(t) = -\nabla_L \mathcal{L}(U_k^t L(t)^\top z_{k-1} + b_k^t), L(0) = L_k^t\big\}$
8   **if** `adaptive` **then** `/* Basis augmentation step */`
9     $K_k^{t+1} \leftarrow [K_k^{t+1} \mid U_k^t]$
10     $L_k^{t+1} \leftarrow [L_k^{t+1} \mid V_k^t]$
11    $U_k^{t+1} \leftarrow$ orthonormal basis for the range of K_k^{t+1} `/* S-step */`
12    $M_k \leftarrow (U_k^{t+1})^\top U_k^t$
13    $V_k^{t+1} \leftarrow$ orthonormal basis for the range of L_k^{t+1}
14    $N_k \leftarrow (V_k^{t+1})^\top V_k^t$
15    $\widetilde{S}_k^t \leftarrow M_k S_k^t N_k^\top$
16    $S_k^{t+1} \leftarrow$ one-step-integrate$\big\{\dot{S}(t) = -\nabla_S \mathcal{L}\big(U_k^{t+1} S(t) (V_k^{t+1})^\top z_{k-1} + b_k^t\big), S(0) = \widetilde{S}_k^t\big\}$
17   **if** `adaptive` **then** `/* Rank compression step */`
18     $P, \Sigma, Q \leftarrow \mathrm{SVD}(S_k^{t+1})$
19     $S_k^{t+1} \leftarrow$ truncate Σ using the singular value threshold ϑ
20     $U_k^{t+1} \leftarrow U_k^{t+1} \widetilde{P}$ where $\widetilde{P} = [$first r_k^{t+1} columns of $P]$
21     $V_k^{t+1} \leftarrow V_k^{t+1} \widetilde{Q}$ where $\widetilde{Q} = [$first r_k^{t+1} columns of $Q]$
`/* Bias update step */`
22    $b_k^{t+1} \leftarrow$ one-step-integrate$\big\{\dot{b}(t) = -\nabla_b \mathcal{L}(U_k^{t+1} S_k^{t+1} (V_k^{t+1})^\top z_{k-1} + b(t)), b(0) = b_k^t\big\}$

4.3 Implementation details, computational costs and limitations

Each step of Alg. 1 requires the computation of two orthonormal bases for the ranges of K_k^{t+1} and L_k^{t+1}. There are of course different techniques to compute such orthonormal matrices. In our implementation we use the QR algorithm, which is known to be one of the most efficient and stable approaches for this purpose. In the adaptive strategy the singular values of S_k^{t+1} are truncated according to a parameter ϑ. To this end, in our implementation, we use the Frobenius norm of Σ. Precisely, we truncate $\Sigma = \mathrm{diag}(\sigma_i)$ at step 19 of Alg. 1 by selecting the smallest principal $r \times r$ submatrix such that $(\sum_{i \geq r+1} \sigma_i^2)^{1/2} \leq \vartheta$. Finally, one-step-integrate denotes a numerical procedure that integrates the corresponding ODE from time $t = 0$ to $t = \eta$. In practice one can employ different numerical integrators, without affecting the ability of the algorithm to reduce the loss function (see [5, Thm. 5]) while maintaining the low-rank structure. In our implementation we used two methods:

1. Explicit Euler. This method applied to the gradient flow coincides with one step of Stochastic Gradient Descent (SGD), applied to the three factors K_k, L_k, S_k independently.
2. Adam. Here we formally compute the new factors by modifying the explicit Euler step as in the Adam optimization method. Note that Nesterov accelerated SGD is known to coincide with a particular linear multistep ODE integrator [49]. While Adam does not directly correspond to a numerical integrator to our knowledge, in our tests it resulted in a faster decrease of the loss than both Euler (SGD) and Nesterov accelerated SGD.

For both choices, the target time step η corresponds to the value of the learning rate, which we set to 0.2 for Euler. For Adam, we use the default dynamical update, setting 0.001 as starting value.

Computational cost. To obtain minimal computational costs and memory requirements for the K-step, the ordering of evaluating $K_k V_k^\top z_{k-1}$ in (9) is important. First, we compute $\widetilde{z} := V_k^\top z_{k-1} \in \mathbb{R}^{r_k}$ which requires $O(r_k n_{k-1})$ operations. Second, we compute $K_k \widetilde{z}$ which requires $O(r_k n_k)$ operations. Adding the bias term and evaluating the activation function requires $O(n_k)$ operations. Hence, combined over all layers we have asymptotic cost of $O\left(\sum_k r_k(n_k + n_{k+1})\right)$. Taping the forward evaluation to compute the gradient with respect to K_k as discussed in §4.2 does not affect the asymptotic costs, i.e. the costs of computing the K-step at layer k assuming a single data point x requires $C_K \lesssim \sum_k r_k(n_k + n_{k+1})$ operations. In a similar manner, we obtain the computational costs of the L- and S-steps, which are again $C_{L,S} \lesssim \sum_k r_k(n_k + n_{k+1})$. Moreover, the QR decompositions used in the K- and L-step require $O\left(\sum_k r_k^2(n_k + n_{k-1})\right)$ operations and computing the SVD in the truncation step has worst-case cost of $O\left(\sum_k r_k^3\right)$. Hence, assuming $r_k \ll n_k, n_{k+1}$, the cost per step of our low-rank method is $C_{\text{DLRA}} \lesssim \sum_k r_k^2(n_k + n_{k-1})$, opposed to the dense network training, which requires $C_{\text{dense}} \lesssim \sum_k n_k n_{k+1}$ operations. In terms of memory cost, note that we only need to store $r_k(1 + n_k + n_{k+1})$ parameters per layer during the algorithm, corresponding to the matrices S_k^t, U_k^t, V_k^t. Moreover, at the end of the training we can further compress memory by storing the product of the trained weight factors $U_k S_k$, rather than the individual matrices.

Limitations. A requirement for DLRT's efficiency is that $r_k \ll n_k, n_{k+1}$. When the truncation threshold ϑ is too small, Alg. 1 does not provide advantages with respect to standard training. This is also shown by Fig. 1. Moreover, in the terminology of [51], DLRT is designed to reduce training costs corresponding to model parameters and to the optimizer. To additionally decrease activation costs, DLRT can be combined with micro-batching or checkpointing approaches. Finally, the choice of ϑ introduces one additional hyperparameter which at the moment requires external knowledge for tuning. However, our experiments in §5 show that relatively large values of ϑ yield competing performance as compared to a number of baselines, including standard training.

5 Numerical Results

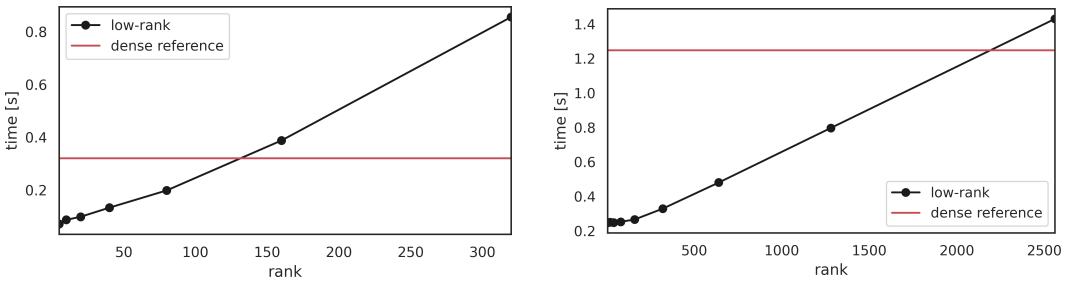
We illustrate the performance of DLRT Algorithm 1 on several test cases. The code is implemented in both Tensorflow [1] and PyTorch [47], and it is available in the supplementary material. The networks are trained on an AMD Ryzen 9 3950X CPU and a Nvidia RTX 3090 GPU. Timings are measured on pure CPU execution. We inspect and compare the performance of the dynamical low-rank training approach on the MNIST data-set [12], which we partition in randomly sampled train-validation-test sets of size $50K$-$10K$-$10K$: The images are pixelwise normalized and no further data augmentation or weight regularization has been used. The MNIST dataset is made available under the terms of the Creative Commons Attribution-Share Alike 3.0 license.

5.1 Fixed-rank fully-connected feed-forward network timings

First, we compare the training time of the adaptive DLRT Algorithm 1 on a 5-layer fully-connected $[5120, 5120, 5120, 5120, 10]$ network fixing the ranks of layers 1-4, i.e. choosing a specific starting rank r_k^0 for the input weight factors and truncating Σ at line 19 of Alg. 1 to the principal $r_k^0 \times r_k^0$ submatrix, rather than via a threshold. Next, we measure the average prediction time of the whole MNIST dataset over 1000 runs. Figures 1(a) and 1(b) show that both timings scale linearly with the rank of the factorizations, and that for sufficiently small ranks the dynamical low-rank approach is faster than the full baseline both in terms of training and of prediction.

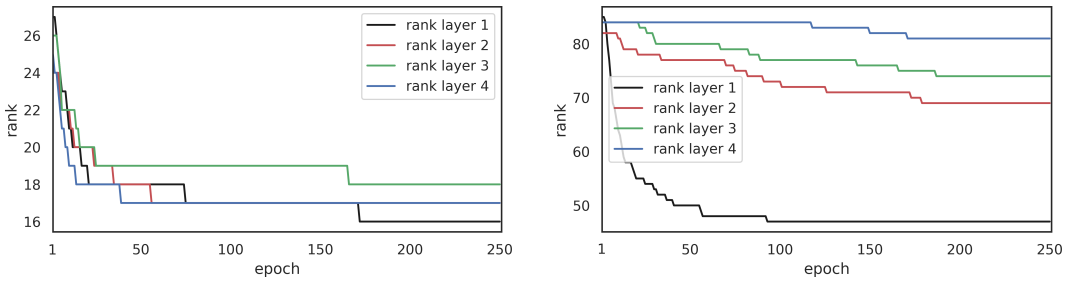
5.2 Rank evolution and performance of DLRT for different singular value thresholds

Next, we demonstrate the capabilities of DLRT to determine the rank of the network's weight matrices automatically during the network training using Algorithm 1. The Adam optimizer with default learning rate is used for the gradient update. We train fully connected 5-layer networks, of which the first 4 are replaced by low-rank layers in the subsequent tests. The activation function is chosen to be ReLU for the hidden layers, and softmax for the output layer. The training loss is sparse categorical cross entropy and we additionally measure the model's accuracy. We use batch size 256 and train for 250 epochs. We choose $\vartheta = \tau \|\Sigma\|$, thus we truncate the singular values of the current S_k^t by a fraction τ of the total Frobenius norm. The smaller τ, the more singular values are kept.



(a) Training timings (b) Prediction timings

Figure 1: Comparison of batch execution and training times of 5-layer, 5120-neurons low-rank networks of different ranks and a non-factorized reference network with the same architecture on the MNIST dataset. Training times shown correspond to one epoch for a batch of 256 datapoints. Prediction times refer instead to the whole dataset. All the times are the average over 1000 runs.



(a) Rank evolution for $\tau = 0.15$ (b) Rank evolution of $\tau = 0.05$

Figure 2: Rank evolution (layers 1-4) of 5-layer [500,500,500,500,10] fully-connected net on MNIST.

Figures 2 (a) and (b) show the evolution of the rank adaptive layers of a 5-layer 500-neuron network in a long time case study for $\tau = 0.05$ and $\tau = 0.15$. We can see that within the first epoch the initial full matrix ranks are reduced significantly, to 27 for $\tau = 0.15$, and to ~ 85 for $\tau = 0.05$ respectively. Within the first 50 epochs, the layer ranks are already close to their final ranks. This indicates that the rank adaptive algorithm is only needed for the first few training epochs, and can then be replaced by the computationally cheaper fixed-low-rank training (by setting the Boolean variable `adaptive` to `False` in Algorithm 1). Figure 3 compares the mean test accuracy of 5-layer networks with 500 and 784 neurons with different levels of low-rank compression, over five independent runs with randomly sampled train-test-val sets. The networks can be compressed via dynamical low-rank training by more than 95%, while only losing little more than 1% test accuracy compared to the dense reference network marked in red. Remark that restricting the space of possible networks to a given rank regularizes the problem, since such a restriction can be understood as adding a PCR regularization term to the loss function. This can be seen from the tendency of not overfitting and reaching improved test accuracies compared to the corresponding dense network for moderate compression ratios. Also note that adaptive-low rank training eliminates the need for hyperparameter grid search in terms of layer-weights, due to automatic rank adaptation. The rank dynamics for all configurations can be seen in the supplementary material §6.3. Finally, in the supplementary material §6.6 we compare the use of DLRT with the vanilla approach which simply thresholds the singular values of the full-rank network. Our results show that advantageous low-rank winning tickets exist, but are not easy to find. In fact, the vanilla low-rank subnetworks perform very poorly. From this point of view, our approach can be seen as an efficient dynamical pruning technique, able to determine high-performing low-rank subnetworks in a given dense network. Remarkably, our numerical experiments suggest that low-rank winning tickets can be trained from the start and do not to heavily depend on the initial weight guess.

5.3 Convolutional layers: LeNet5

Here we compare the proposed dynamical low-rank training scheme on LeNet5 [38] on MNIST, against the full-rank reference and several baselines. SVD prune [58] and LRNN [27] are the closest approaches to our DLRT: they dynamically train low-rank layers by adding a rank-penalty to the loss

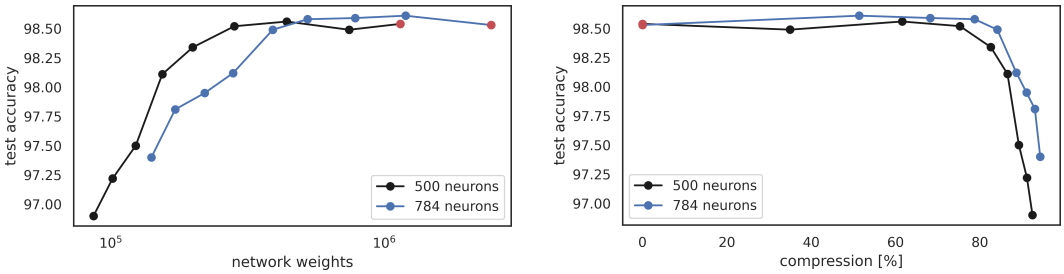


Figure 3: Mean test accuracy over parameters' number and compression rate for 5 runs with randomly sampled train-test-val sets on 5-layer fully-connected nets. Red dots denote the full-rank baseline.

Table 1: Results of the training of LeNet5 on MNIST dataset. Effective parameters represent the number of parameters we have to save for evaluating the network and those we need in order to train via the DLRT Alg.1. The compression ratio (c.r.) is the percentage of parameter reduction with respect to the full model ($< 0\%$ indicates that the ratio is negative). "ft" indicates that the model has been fine tuned. "LeNet5" denotes the standard LeNet5 architecture trained with SGD.

		NN metrics		Evaluation		Train	
method		test acc.	ranks	params	c.r.	params	c.r.
LeNet5		**99.2**%	[20, 50, 500, 10]	430500	0%	430500	0%
DLRT	$\tau = 0.11$	98.0%	[15, 46, 13, 10]	47975	88.86%	50585	88.25%
	$\tau = 0.15$	97.8%	[13, 31, 9, 10]	34435	92.0%	35746	91.7%
	$\tau = 0.2$	97.2%	[10, 20, 7, 10]	25650	94.04%	26299	93.89%
	$\tau = 0.3$	95.3%	[6, 9, 4, 10]	15520	**96.4**%	15753	**96.34**%
SSL [59] (ft)		99.18%		110000	74.4%		$< 0\%$
NISP [55] (ft)		99.0%		100000	76.5%		$< 0\%$
GAL [40]		98.97%		30000	93.0%		$< 0\%$
LRNN [27]		98.67%	[3, 3, 9, 9]	18075	95.8%		$< 0\%$
SVD prune [58]		94.0%	[2, 5, 89, 10]	123646	71.2%		$< 0\%$

function, and by complementing the standard training step via an SVD projection step in the latter and a pruning step in the former. While computing low-rank factors for each layer, thus reducing memory storage of the network, this training approach is more expensive than training the full network. GAL [40], SSL [59], and NISP [55] are pruning methods which aim at learning optimal sparse weights (rather than low-rank) by adding sparsity-promoting regularization terms to the training loss. As for LRNN, these methods do not reduce the computational cost of the training phase (as indicated with the $< 0\%$ in Table 1). Analogously to [27], our adaptive low-rank training technique is applied to the convolutional layers by flattening the tensor representing the convolutional kernel into a matrix. Details are provided in the supplementary material §6.8. All the models are trained for 120 epochs using SGD with a fixed learning rate of 0.2. Results in Table 1 show that the DLRT algorithm is able to find low-rank subnetworks with up to 96.4% less parameters than the full reference, while keeping the test accuracy above 95%. Compared to the baseline methods, we achieve better compression rates but observe lower accuracy. However, unlike the baseline references, DLRT automatically prunes the singular values during training, without requirement to solve any additional optimization problem, thus significantly improving time and memory efficiency of both forward and backward phase, with respect to the full reference.

5.4 Results on Cifar10 and Cifar100 with AlexNet and VGG16

We compare DLRT with standard full-rank training and a number of low-rank alternative baselines on the VGG16 and AlexNet models on the Cifar10 and Cifar100 datasets using the SGD optimizer with momentum factor 0.1. The last layers of the networks have been adapted to match the corresponding classification tasks. Detailed results are reported in Tables 7 and 8 in the supplementary material. In Cifar10, see Table 7, we archive a train compression of 77.5% with an accuracy loss of just 1.1% for VGG16 and 84.2% compression at 1.79% accuracy loss. For the Cifar100 test case, see Table 8, we have 72% train compression and a 2% gain of accuracy for VGG16, and 85% compression at 3.88% accuracy loss over the corresponding baseline for AlexNet.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009.

[3] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani. N2n learning: Network to network compression via policy gradient reinforcement learning. In *International Conference on Learning Representations*, 2018.

[4] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[5] G. Ceruti, J. Kusch, and C. Lubich. A rank-adaptive robust integrator for dynamical low-rank approximation. *BIT Numerical Mathematics*, 2022.

[6] G. Ceruti and C. Lubich. Time integration of symmetric and anti-symmetric low-rank matrices and Tucker tensors. *BIT Numerical Mathematics*, 60(3):591–614, 2020.

[7] G. Ceruti and C. Lubich. An unconventional robust integrator for dynamical low-rank approximation. *BIT. Numerical Mathematics*, 62(1):23–44, 2022.

[8] G. Ceruti, C. Lubich, and D. Sulz. Rank-adaptive time integration of tree tensor networks. *arXiv:2201.10291*, 2022.

[9] G. Ceruti, C. Lubich, and H. Walach. Time integration of tree tensor networks. *SIAM Journal on Numerical Analysis*, 59(1):289–313, 2021.

[10] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE international conference on computer vision*, pages 2857–2865, 2015.

[11] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *Advances in neural information processing systems*, 2016.

[12] L. Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[13] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

[14] L. Dieci and T. Eirola. On smooth decompositions of matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(3):800–819, 1999.

[15] P. A. M. Dirac et al. *The principles of quantum mechanics*. Number 27. Oxford university press, 1981.

[16] R. Feng, K. Zheng, Y. Huang, D. Zhao, M. Jordan, and Z.-J. Zha. Rank diminishing in deep neural networks. *arXiv:2206.06072*, 2022.

[17] F. Feppon and P. F. Lermusiaux. A geometric approach to dynamical model order reduction. *SIAM Journal on Matrix Analysis and Applications*, 39(1):510–538, 2018.

[18] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[19] J. Frenkel. *Wave mechanics, advanced general theory*, volume 1. Oxford, 1934.

[20] B. Gao and P.-A. Absil. A riemannian rank-adaptive method for low-rank matrix completion. *Computational Optimization and Applications*, 81(1):67–90, 2022.

[21] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. *Advances in neural information processing systems*, 29, 2016.

[22] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations. I. Nonstiff problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993.

[23] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision*, pages 784–800, 2018.

[24] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.

[25] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[26] Z. Huang and N. Wang. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 304–320, 2018.

[27] Y. Idelbayev and M. A. Carreira-Perpiñán. Low-rank compression of neural nets: Learning the rank of each layer. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8046–8056, 2020.

[28] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi. Training CNNs with low-rank filters for efficient image classification. In *International Conference on Learning Representations*, 2016.

[29] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014.

[30] M. Khodak, N. Tenenholtz, L. Mackey, and N. Fusi. Initialization and regularization of factorized neural layers. *arXiv preprint arXiv:2105.01029*, 2021.

[31] E. Kieri, C. Lubich, and H. Walach. Discretized dynamical low-rank approximation in the presence of small singular values. *SIAM Journal on Numerical Analysis*, 54(2):1020–1038, 2016.

[32] H. Kim, M. U. K. Khan, and C.-M. Kyung. Efficient neural network compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12569–12577, 2019.

[33] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications, 2015.

[34] O. Koch and C. Lubich. Dynamical low-rank approximation. *SIAM Journal on Matrix Analysis and Applications*, 29(2):434–454, 2007.

[35] O. Koch and C. Lubich. Dynamical tensor approximation. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2360–2375, 2010.

[36] J. Kusch and P. Stammer. A robust collision source method for rank adaptive dynamical low-rank approximation in radiation therapy. *arXiv:2111.07160*, 2021.

[37] 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*, 2015.

[38] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[39] C. Li and C. J. R. Shi. Constrained optimization based low-rank approximation of deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[40] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann. Towards optimal structured CNN pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2790–2799, 2019.

[41] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018.

[42] C. Lubich and I. V. Oseledets. A projector-splitting integrator for dynamical low-rank approximation. *BIT Numerical Mathematics*, 54(1):171–188, 2014.

[43] C. Lubich, T. Rohwedder, R. Schneider, and B. Vandereycken. Dynamical approximation by hierarchical Tucker and tensor-train tensors. *SIAM Journal on Matrix Analysis and Applications*, 34(2):470–494, 2013.

[44] C. Lubich, B. Vandereycken, and H. Walach. Time integration of rank-constrained Tucker tensors. *SIAM Journal on Numerical Analysis*, 56(3):1273–1290, 2018.

[45] C. H. Martin and M. W. Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *Journal of Machine Learning Research*, 22(165):1–73, 2021.

[46] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. In *International Conference on Learning Representations*, 2017.

[47] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[48] 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 *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6655–6659, 2013.

[49] D. Scieur, V. Roulet, F. Bach, and A. d'Aspremont. Integration methods and accelerated optimization algorithms. In *Advances In Neural Information Processing Systems*, 2017.

[50] S. P. Singh, G. Bachmann, and T. Hofmann. Analytic insights into structure and rank of neural network Hessian maps. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

[51] N. S. Sohoni, C. R. Aberger, M. Leszczynski, J. Zhang, and C. Ré. Low-memory neural network training: A technical report. *arXiv:1904.10631*, 2019.

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

[53] M. Udell and A. Townsend. Why are big data matrices approximately low rank? *SIAM Journal on Mathematics of Data Science*, 1(1):144–160, 2019.

[54] H. Wang, S. Agarwal, and D. Papailiopoulos. Pufferfish: communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3:365–386, 2021.

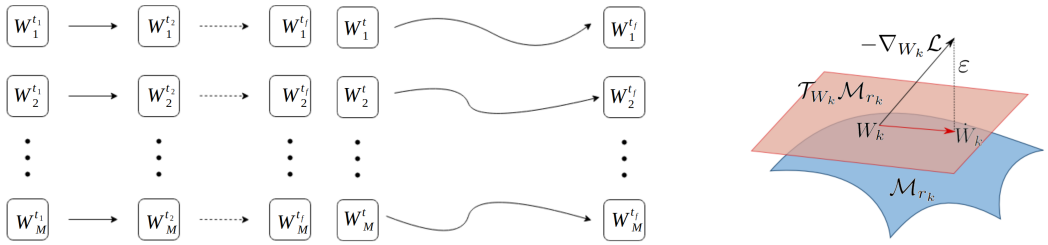
[55] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[56] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li. Coordinating filters for faster deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 658–666, 2017.

[57] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4820–4828, 2016.

[58] H. Yang, M. Tang, W. Wen, F. Yan, D. Hu, A. Li, H. Li, and Y. Chen. Learning low-rank deep neural networks via singular vector orthogonality regularization and singular value sparsification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 678–679, 2020.

[59] J. Ye, X. Lu, Z. Lin, and J. Z. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *International Conference on Learning Representations*, 2018.



(a) Discrete time weight update (b) Continuous time weight update (c) Galerkin condition

Figure 4: Panels (a)-(b): Graphical re-interpretation of the weight update step as a time-continuous process. Panel (c): Orthogonal projection onto the tangent space of the low-rank manifold $\mathcal{M}_r$. The dashed line depicts the projection resulting in $\dot{W}_k(t)$, which is the tangent element minimizing the distance between $\nabla_{W_k}\mathcal{L}(W_k(t))$ and the tangent space $\mathcal{T}_{W_k(t)}\mathcal{M}_r$ at the approximation $W_k(t)$.

6 Appendix

6.1 Proofs of the main results

We provide here a proof of Theorems 1 and 2. The proof is based on a number of classical results as well as recent advances in DLRA theory, including [5, 7, 31, 34, 42].

Recall that, for a fixed layer k, we reinterpret the training phase as a continuous-time evolution of the weights on the manifold of low-rank matrices, as illustrated in Fig. 4(a-b). This boils down to solving the manifold-constrained matrix differential equation

$$\min\left\{\|\dot{W}_k(t) - \mathcal{F}_k(W_k(t))\| \,:\, \dot{W}_k(t) \in \mathcal{T}_{W_k(t)}\mathcal{M}_{r_k}\right\}, \tag{10}$$

where $\|\cdot\|$ is the Frobenius norm and $\mathcal{F}_k$ denotes the gradient flow of the loss with respect to the k-th matrix variable, namely

$$\mathcal{F}_k(Z) = -\nabla_{W_k}\mathcal{L}(W_1, \dots, Z, \dots, W_M; \mathcal{N}(x), y)\,.$$

For the sake of simplicity and for a cleaner notation, as all the results we will present hold for a generic k, we drop the subscript k from now on. In particular, we assume W is the weight matrix of a generic hidden layer with n input and m output neurons.

In order for our derivation to hold, we require the following two properties:

P1. The gradient flow $\mathcal{F}$ is locally bounded and locally Lipschitz continuous, with constants C_1 and C_2, respectively. Namely, we assume there exist $C_1, C_2 > 0$ (independent of k) such that

$$\|\mathcal{F}(Z)\| \leq C_1 \qquad \|\mathcal{F}(Z) - \mathcal{F}(\tilde{Z})\| \leq C_2\|Z - \tilde{Z}\|$$

for all $Z, \tilde{Z} \in \mathbb{R}^{m\times n}$.

P2. The whole gradient flow is "not too far" from the rank-r manifold $\mathcal{M}_r$. Precisely, we assume that for any $Z \in \mathcal{M}_r$ arbitrary close to $W(t)$, the whole gradient flow $\mathcal{F}(Z)$ near t is such that

$$\|(I - \mathrm{P}(Z))\mathcal{F}(Z)\| \leq \varepsilon\,,$$

where $\mathrm{P}(Z)$ denotes the orthogonal projection onto $\mathcal{T}_Z\mathcal{M}_r$.

Note that both assumptions are valid for low-rank neural network training. In particular, Lipschitz continuity and boundedness of the gradient are standard assumptions in optimization and are satisfied by the gradient of commonly used neural networks' losses. Moreover, assuming the gradient flow to be close to the low-rank manifold is an often encountered empirical observation in neural networks [16, 45, 50].

In order to derive the proof of Theorems 1 and 2 we first present a number of relevant background lemmas. The first lemma shows that the subspace generated by the K-step in Algorithm 1 after the QR-decomposition is $\mathcal{O}(\eta(\eta + \varepsilon))$ close to the range of the exact solution.

Lemma 1 ([7, Lemma 2])**.** *Let W^1 be the solution at time $t = \eta$ of the full problem (2) with initial condition W^0. Let U^1 be the matrix obtained with the K-step of the fixed-rank Algorithm 1, after one step. Under the assumptions P1 and P2 above, we have*

$$\|U^1 U^{1,\top} W^1 - W^1\| \leq \theta$$

where

$$\theta = C_1 C_2 (4e^{C_2\eta} + 9)\eta^2 + (3e^{C_2\eta} + 4)\varepsilon\eta\,.$$

Proof. The local error analysis of [31] shows that there exists L^1 such that

$$\|U^1 L^{1,\top} - W^1\| \leq \theta.$$

It follows that,

$$\begin{aligned}
\|U^1 L^{1,\top} - W^1\|^2 &= \|U^1 L^{1,\top} - U^1 U^{1,\top} W^1 + U^1 U^{1,\top} W^1 - W^1\|^2 \\
&= \|U^1 U^{1,\top}(U^1 L^{1,\top} - W^1) + (I - U^1 U^{1,\top})(-W^1)\|^2 \\
&= \|U^1 U^{1,\top}(U^1 L^{1,\top} - W^1)\|^2 + \|(I - U^1 U^{1,\top})W^1\|^2.
\end{aligned}$$

Therefore,

$$\|U^1 U^{1,\top}(U^1 L^{1,\top} - W^1)\|^2 + \|(I - U^1 U^{1,\top})W^1\|^2 \leq \theta^2.$$

Hence, since both terms must be bounded by θ^2 individually, we obtain the stated result. □

In the next lemma we show that also the space generated by the L step is close by the exact solution. Namely, combined with the previous result, we have

Lemma 2 ([7, Lemma 3])**.** *Let W^1, U^1 be defined as above. Let V^1 be the matrix obtained from the L-step of the fixed-rank Algorithm 1, after one step. The following estimate holds:*

$$\|U^1 U^{1,\top} W^1 V^1 V^{1,\top} - W^1\| \leq 2\theta.$$

Proof. The L-step is obtained as the K-step applied to the transposed function $\mathcal{G}(Y) = \mathcal{F}(Y^\top)^\top$. Due to the invariance of the Frobenius norm under transposition, property P1 holds. Similarly, property P2 continues to be satisfied because

$$\|(I - \mathrm{P}(Y))\mathcal{G}(Y)\| = \|(I - \mathrm{P}(Y^\top))\mathcal{F}(Y^\top)\| \leq \varepsilon,$$

where the equality $\mathrm{P}(Y)Z^\top = \left[\mathrm{P}(Y^\top)Z\right]^\top$ has been used [34, §4]. It follows from Lemma 1 that

$$\begin{aligned}
&\|U^1 U^{1,\top} W^1 - W^1\| \leq \theta, \\
&\|V^1 V^{1,\top} W^{1,\top} - W^{1,\top}\| \leq \theta.
\end{aligned} \tag{11}$$

This implies that

$$\begin{aligned}
\|U^1 U^{1,\top} W^1 V^1 V^{1,\top} - W^1\| &\leq \|U^1 U^{1,\top} W^1 V^1 V^{1,\top} - W^1 V^1 V^{1,\top} + W^1 V^1 V^{1,\top} - W^1\| \\
&\leq \|U^1 U^{1,\top} W^1 V^1 V^{1,\top} - W^1 V^1 V^{1,\top}\| + \|W^1 V^1 V^{1,\top} - W^1\| \\
&\leq \|\big(U^1 U^{1,\top} W^1 - W^1\big) V^1 V^{1,\top}\| + \|V^1 V^{1,\top} W^{1,\top} - W^{1,\top}\| \\
&\leq \|U^1 U^{1,\top} W^1 - W^1\| \cdot \|V^1 V^{1,\top}\|_2 + \|V^1 V^{1,\top} W^{1,\top} - W^{1,\top}\|.
\end{aligned}$$

Because $\|V^1 V^{1,\top}\|_2 = 1$, the stated result follows from (11). □

With the previous lemmas, we are in the position to derive the local error bound for the fixed-rank KLS integrator of Section 4.

Lemma 3 (Local Error, [7, Lemma 4])**.** *Let W^1, U^1, V^1 be defined as above and let S^1 be the matrix obtained with the S-step of Algorithm 1 after one step. The following local error bound holds:*

$$\|U^1 S^1 V^{1,\top} - W^1\| \leq \eta(\hat{c}_1\varepsilon + \hat{c}_2\eta),$$

where the constants $\hat{c}_i$ are independent of the singular values of W^1 and S^1.

Proof. From Lemma 2 and the equality $Y^1 = U^1 S^1 V^{1,\top}$, we have that

$$\begin{aligned}\|Y^1 - W^1\| &\le \|Y^1 - U^1 U^{1,\top} W^1 V^1 V^{1,\top}\| + \|U^1 U^{1,\top} W^1 V^1 V^{1,\top} - W^1\| \\ &\le \|U^1 (S^1 - U^{1,\top} W^1 V^1) V^{1,\top}\| + 2\theta \\ &\le \|S^1 - U^{1,\top} W^1 V^1\| + 2\theta.\end{aligned}$$

The local error's analysis is reduced to bound the term $\|S^1 - U^{1,\top} W^1 V^1\|$. For $0 \le t \le \eta$, we thus introduce the following auxiliary quantity:

$$\widetilde{S}(t) := U^{1,\top} W(t) V^1.$$

We observe that the term $W(t)$ can be re-written as

$$W(t) = U^1 U^{1,\top} W(t) V^1 V^{1,\top} + \Big(W(t) - U^1 U^{1,\top} W(t) V^1 V^{1,\top}\Big) = U^1 \widetilde{S}(t) V^{1,\top} + \mathcal{R}(t),$$

where $\mathcal{R}(t)$ denotes the term in big brackets. For $0 \le t \le \eta$, it follows from Lemma 2 and the bound C_1 of the function $\mathcal{F}$ that

$$\|W(t) - W(\eta)\| \le \int_0^\eta \|\dot{W}(s)\|\, ds = \int_0^\eta \|\mathcal{F}(W(s))\|\, ds \le C_1 \eta.$$

Therefore, the term $\mathcal{R}(t)$ is bounded by

$$\|\mathcal{R}(t)\| \le \|\mathcal{R}(t) - \mathcal{R}(\eta)\| + \|\mathcal{R}(\eta)\| \le 2C_1\eta + 2\theta.$$

We re-cast the function $\mathcal{F}\big(W(t)\big)$ as

$$\begin{aligned}\mathcal{F}\big(W(t)\big) &= \mathcal{F}\big(U^1 \widetilde{S}(t) V^{1,\top} + \mathcal{R}(t)\big) \\ &= \mathcal{F}\big(U^1 \widetilde{S}(t) V^{1,\top}\big) + \mathcal{D}(t)\end{aligned}$$

where the defect $\mathcal{D}(t)$ is given by

$$\mathcal{D}(t) := \mathcal{F}\big(U^1 \widetilde{S}(t) V^{1,\top} + \mathcal{R}(t)\big) - \mathcal{F}\big(U^1 \widetilde{S}(t) V^{1,\top}\big).$$

Via the Lipschitz constant C_2 of the function $\mathcal{F}$, the defect is bounded by

$$\|\mathcal{D}(t)\| \le C_2 \|\mathcal{R}(t)\| \le 2C_2(C_1\eta + \theta).$$

Now, we compare the two differential equations

$$\begin{aligned}\dot{\widetilde{S}}(t) &= U^{1,\top} \mathcal{F}\big(U^1 \widetilde{S}(t) V^{1,\top}\big) V^1 + U^{1,\top} \mathcal{D}(t) V^1, & \widetilde{S}(0) &= U^{1,\top} W^0 V^1, \\ \dot{S}(t) &= U^{1,\top} \mathcal{F}\big(U^1 S(t) V^{1,\top}\big) V^1, & S(0) &= U^{1,\top} W^0 V^1.\end{aligned}$$

The solution S^1 obtained in the second differential equation is the same as given by the S-step of the KLS integrator of Section 4. By construction, the solution obtained in the first differential equation at time $t = \eta$ is $\widetilde{S}(\eta) = U^{1,\top} W^1 V^1$. With the Gronwall inequality we obtain

$$\|S^1 - U^{1,\top} W^1 V^1\| \le \int_0^\eta e^{C_2(\eta - s)}\, \|\mathcal{D}(s)\|\, ds \le e^{L\eta}\, 2C_2(C_1\eta + \theta)\eta.$$

The result yields the statement of the theorem using the definition of θ. □

We are now in the position to conclude the proof of Theorem 1.

Proof of Theorem 1. In Lemma 3, the local error for the fixed-rank integrator of §4 has been provided. The local error in time of the rank-adaptive version is directly obtained via a triangle inequality:

$$\|U^1 S^1 V^{1,\top} - W(\eta)\| \le \hat{c}_1 \varepsilon \eta + \hat{c}_2 \eta^2 + \vartheta\,,$$

where ϑ is the tolerance parameter chosen for the truncation procedure. Here, we abuse the notation and we maintain the same nomenclature U^1, S^1, and V^1 also for the novel low-rank approximation obtained via the truncation procedure.

Thus, we conclude the proof using the Lipschitz continuity of the function $\mathcal{F}$. We move from the local error in time to the global error in time by a standard argument of Lady Windermere's fan [22, Section II.3]. Therefore, the error after t steps of the rank-adaptive Algorithm 1 is given by

$$\|U^t S^t V^{t,\top} - W(t\eta)\| \le c_1 \varepsilon + c_2 \eta + c_3 \vartheta/\eta\,.$$

□

To conclude with, we prove that after one step the proposed rank-adaptive DLRT algorithm decreases along the low-rank approximations. We remind that only property P1 needs to be assumed here.

Proof of Theorem 2. Let $\widehat{Y}(t) = U^1 S(t) V^{1,\top}$. Here, $S(t)$ denotes the solution for $t \in [0, \eta]$ of the S-step of the rank-adaptive integrator . It follows that

$$\begin{aligned}\frac{d}{dt}\mathcal{L}(\widehat{Y}(t)) &= \langle \nabla \mathcal{L}(\widehat{Y}(t)), \dot{\widehat{Y}}(t)\rangle \\ &= \langle \nabla \mathcal{L}(\widehat{Y}(t)), U^1 \dot{S}(t) V^{1,\top}\rangle \\ &= \langle U^{1,\top} \nabla \mathcal{L}(\widehat{Y}(t)) V^1, \dot{S}(t)\rangle \\ &= \langle U^{1,\top} \nabla \mathcal{L}(\widehat{Y}(t)) V^1,\ -U^{1,\top} \nabla \mathcal{L}(\widehat{Y}(t)) V^1\rangle = -\|U^{1,\top} \nabla \mathcal{L}(\widehat{Y}(t)) V^1\|^2 .\end{aligned}$$

The last identities follow by definition of the S-step. For $t \in [0, \eta]$ we have

$$\frac{d}{dt}\mathcal{L}(\widehat{Y}(t)) \leq -\alpha^2 \tag{12}$$

where $\alpha = \min_{0\leq\tau\leq 1} \|U^{1,\top} \nabla \mathcal{L}\big(\widehat{Y}(\tau\eta)\big) V^1\|$. Integrating (12) from $t = 0$ until $t = \eta$, we obtain

$$\mathcal{L}(\widehat{Y}^1) \leq \mathcal{L}(\widehat{Y}^0) - \alpha^2 \eta.$$

Because the subspace U^1 and V^1 contain by construction the range and co-range of the initial value, we have that $\widehat{Y}^0 = U^0 S^0 V^{0,\top}$ [5, Lemma 1]. The truncation is such that $\|Y^1 - \widehat{Y}^1\| \leq \vartheta$. Therefore,

$$\mathcal{L}(Y^1) \leq \mathcal{L}(\widehat{Y}^1) + \beta\vartheta$$

where $\beta = \max_{0\leq\tau\leq 1} \|\nabla \mathcal{L}\big(\tau Y^1 + (1-\tau)\widehat{Y}^1\big)\|$. Hence, the stated result is obtained. □

6.2 Detailed timing measurements

Table 2 displays the average batch training times of a 5-layer, 5120-neuron dense network on the MNIST dataset, with a batch size of 500 samples. We average the timings over 200 batches and additionally display the standard deviation of the timings corresponding to the layer ranks. The batch timing measures the full K,L and S steps, including back-propagation and gradient updates, as well as the loss and metric evaluations.

Table 2: Average batch training times for fixed low-rank training of a 5-layer fully-connected network with layer widths $[5120, 5120, 5120, 5120, 10]$. Different low-rank factorizations are compared

ranks	mean time [s]	std. deviation [s]
full-rank	0.320	± 0.005227
$[320, 320, 320, 320, 320]$	0.855	± 0.006547
$[160, 160, 160, 160, 10]$	0.387	± 0.005657
$[80, 80, 80, 80, 10]$	0.198	± 0.004816
$[40, 40, 40, 40, 10]$	0.133	± 0.005984
$[20, 20, 20, 20, 10]$	0.098	± 0.005650
$[10, 10, 10, 10, 10]$	0.087	± 0.005734
$[5, 5, 5, 5, 10]$	0.071	± 0.005369

Table 3 shows the average test time of a 5-layer, 5120-neuron dense network, for different low-rank factorizations and the full rank reference network. The timings are averaged over 100 evaluations of the $60K$ sample MNIST training data set. We measure the K step forward evaluation of the low-rank networks as well as the loss and prediction accuracy evaluations.

6.3 Detailed training performance of adaptive low-rank networks

Tables 4 and 5 display a detailed overview of the adaptive low-rank results of §5.2. The displayed ranks are the ranks of the converged algorithm. The rank evolution of the 5-Layer, 500-Neuron test

Table 3: Average dataset prediction times for fixed low-rank training of a 5-layer fully-connected network with layer widths $[5120, 5120, 5120, 5120, 10]$. Different low-rank factorizations are compared.

ranks	mean time [s]	std. deviation [s]
full-rank	1.2476	± 0.0471
$[2560, 2560, 2560, 2560, 10]$	1.4297	± 0.0400
$[1280, 1280, 1280, 1280, 10]$	0.7966	± 0.0438
$[640, 640, 640, 640, 10]$	0.4802	± 0.0436
$[320, 320, 320, 320, 10]$	0.3286	± 0.0442
$[160, 160, 160, 160, 10]$	0.2659	± 0.0380
$[80, 80, 80, 80, 10]$	0.2522	± 0.0346
$[40, 40, 40, 40, 10]$	0.2480	± 0.0354
$[20, 20, 20, 20, 10]$	0.2501	± 0.0274
$[10, 10, 10, 10, 10]$	0.2487	± 0.0276
$[5, 5, 5, 5, 10]$	0.2472	± 0.0322

case can be seen in Fig. 5. The Evaluation parameter count corresponds to the parameters of the K step of the dynamical low-rank algorithm, since all other matrices are no longer needed in the evaluation phase. The training parameter count is evaluated as the number of parameters of the S step of the adaptive dynamical low rank training, with maximal basis expansion by $2r$, where r is the current rank of the network. We use the converged ranks of the adaptive low-rank training to compute the training parameters. Note that during the very first training epochs, the parameter count is typically higher until the rank reduction has reached a sufficiently low level.

Table 4: Dynamical low rank training for 5-layer 500-neurons network. c.r. denotes the compression rate relative to the full rank dense network.

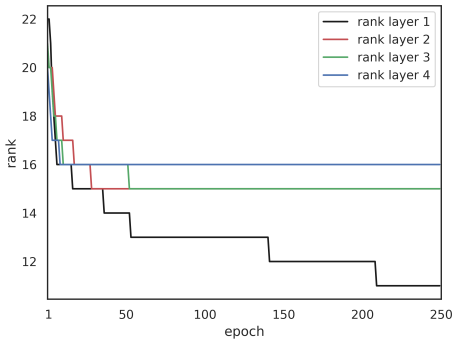
	NN metrics		Evaluation		Train	
τ	test acc.	ranks	params	c.r.	params	c.r.
full-rank	$98.54 \pm 0.03\%$	$[500, 500, 500, 500, 10]$	1147000	0%	1147000	0%
0.03	$98.49 \pm 0.02\%$	$[176, 170, 171, 174, 10]$	745984	34.97%	1964540	-71.27%
0.05	$98.56 \pm 0.02\%$	$[81, 104, 111, 117, 10]$	441004	61.56%	1050556	8.40%
0.07	$98.52 \pm 0.08\%$	$[52, 67, 73, 72, 10]$	283768	75.26%	633360	44.78%
0.09	$98.34 \pm 0.14\%$	$[35, 53, 51, 46, 10]$	199940	82.57%	429884	62.52%
0.11	$98.11 \pm 0.46\%$	$[27, 40, 37, 38, 10]$	154668	86.52%	324904	71.67%
0.13	$97.50 \pm 0.23\%$	$[20, 31, 32, 30, 10]$	123680	89.22%	255500	77.72%
0.15	$97.22 \pm 0.29\%$	$[17, 25, 26, 24, 10]$	101828	91.13%	207320	81.92%
0.17	$96.90 \pm 0.45\%$	$[13, 21, 24, 20, 10]$	86692	92.45%	174728	84.76%

Table 5: Dynamical low rank training for 5-layer 784-neurons network. c.r. denotes the compression rate relative to the full rank dense network.

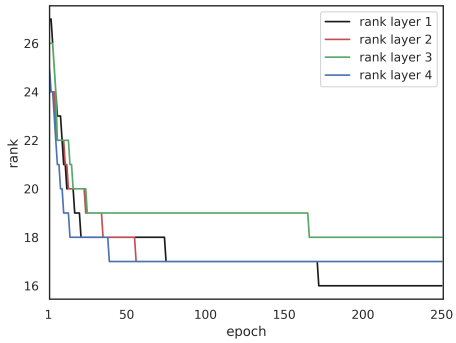
	NN metrics		Evaluation		Train	
τ	test acc.	ranks	params	c.r.	params	c.r.
full-rank	$98.53 \pm 0.04\%$	$[784, 784, 784, 784, 10]$	2466464	0%	2466464	0%
0.03	$98.61 \pm 0.07\%$	$[190, 190, 190, 190, 10]$	1199520	51.37%	2968800	-20.36%
0.05	$98.59 \pm 0.06\%$	$[124, 120, 125, 126, 10]$	784000	68.22%	1805268	26.80%
0.07	$98.58 \pm 0.03\%$	$[76, 86, 85, 83, 10]$	525280	78.71%	1151864	53.29%
0.09	$98.49 \pm 0.05\%$	$[56, 67, 63, 59, 10]$	392000	84.41%	836460	66.08%
0.11	$98.12 \pm 0.21\%$	$[35, 49, 47, 43, 10]$	280672	88.63%	584240	76.31%
0.13	$97.95 \pm 0.23\%$	$[29, 35, 38, 34, 10]$	221088	91.04%	453000	81.63%
0.15	$97.81 \pm 0.17\%$	$[22, 29, 27, 27, 10]$	172480	93.01%	348252	85.88%
0.17	$97.40 \pm 0.25\%$	$[17, 23, 22, 23, 10]$	141120	94.28%	281724	88.57%

6.3.1 Lenet5 experiment

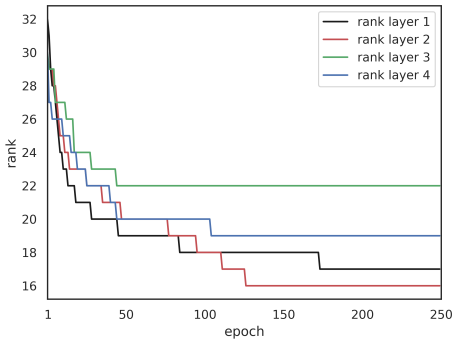
In Table 6 we report the results of five independent runs of the dynamic low-rank training scheme on Lenet5; we refer to §5.3 for further details. For each column of the table, we report the mean value



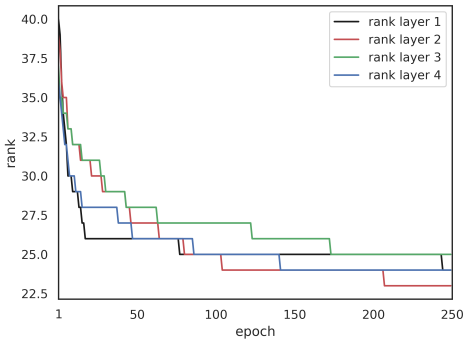
a) Rank evolution for $\tau = 0.17$



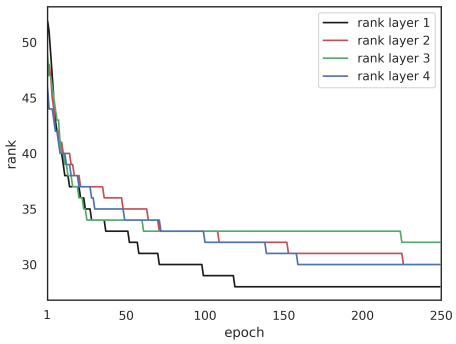
b) Rank evolution for $\tau = 0.15$



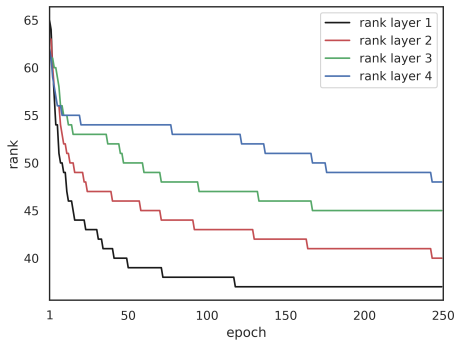
c) Rank evolution for $\tau = 0.13$



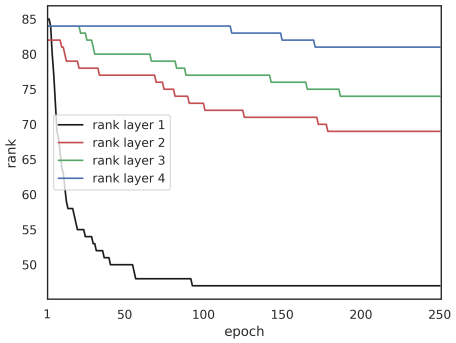
d) Rank evolution for $\tau = 0.11$



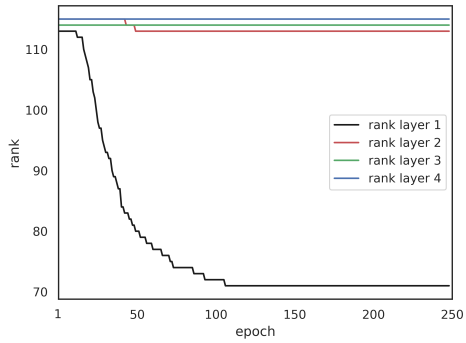
e) Rank evolution for $\tau = 0.09$



f) Rank evolution for $\tau = 0.07$



g) Rank evolution for $\tau = 0.05$



h) Rank evolution for $\tau = 0.03$

Figure 5: Rank evolution of the dynamic adaptive low-rank training algorithm for the 5-layer, 500-neuron dense architecture.

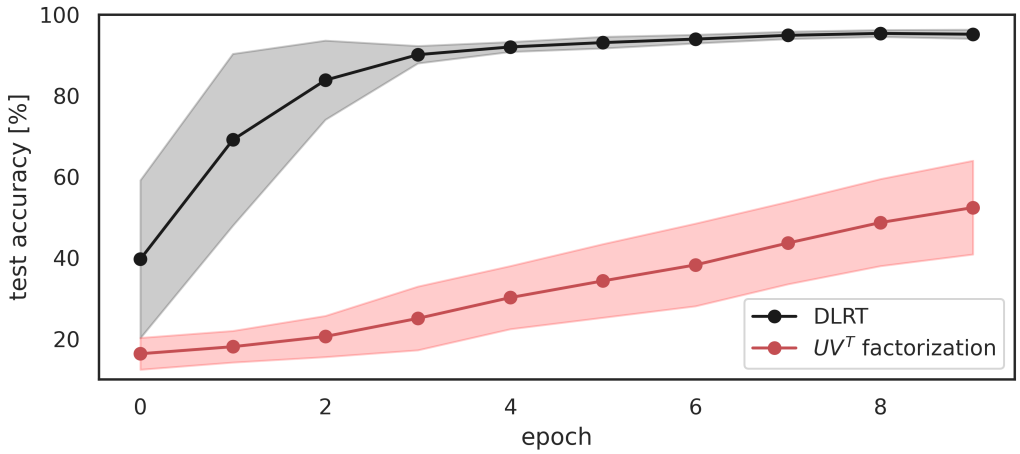
together with its relative standard deviations. No seed has been applied for splitting the dataset and generating the initial weights configuration.

Table 6: Mean results and standard relative deviations of the dynamic low-rank training algorithm over five independent runs on Lenet5. Adaptive learning rate of 0.05 with $0.96-$exponentially decaying tax.

	NN metrics		Evaluation		Train	
τ	test acc.	ranks	params	c.r.	params	c.r.
0.11	$95.420 \pm 1.865\%$	$[15, 46, 13, 10]$	47975	88.9%	50585	88.2%
0.15	$95.527 \pm 1.297\%$	$[13, 31, 9, 10]$	34435	92.0%	35746	91.69%
0.2	$95.009 \pm 1.465\%$	$[10, 20, 7, 10]$	25650	94.04%	26299	93.89%
0.3	$92.434 \pm 1.757\%$	$[6, 9, 4, 10]$	15520	96.39%	15753	96.34%

6.4 Comparison with training of $UV^\top$ factorization

In Figure 6, we study the influence of the SVD-like decomposition used for the derivation of the DLRT algorithm on the test accuracy. Contrary to the proposed strategy used for the DLRT algorithm, performing a low-rank training on a factorization of the form $W_k = U_k V_k^\top$ as used in [30, 54] is not robust with respect to the presence of small singular values. The problem is peculiar to the low-rank manifold itself, as discussed in [17, 34]. In fact, the local curvature of the manifold of fixed-rank matrices is proportional to the inverse of the smallest singular value of the solution (or weight matrices in the neural network methodology). Therefore, a naive approach for training the weights on the low-rank manifold might translate in an ill-conditioned optimization which requires reduced learning rates. In contrast to this approach, the DLRT algorithm takes advantage of the ruled structure of the manifold, and it moves along flat sub-spaces mitigating problems arising from the high-curvature of the manifold [31]. This can be seen from the bound of Theorem 1, where the constants are independent of the singular values of the weight-matrices. In the following, we provide an ad-hoc example to illustrate these situations in the context of low-rank neural network's training. Therefore, we compare the DLRT algorithm with a training performed directly on the factors of an $UV^\top$ decomposition[30], where the initial value for the training exhibits exponentially decaying singular values. The resulting test accuracy over the epoch number can be found in Figure 6. One observes that for the chosen learning rate, the $UV^\top$ based training converges significantly slower than DLRT.



(a) test accuracy over epochs

Figure 6: Mean learning curves of Lenet5 on MNIST dataset with their relative standard deviations over 10 runs of DLRT compared to a vanilla layer factorization $W = UV^\top$. The initialization of the weight matrices has exponentially decaying singular values. The chosen learning rate is 0.01, the batch-size of 128 is used. .

6.5 Cifar10 and Cifar100 experiments with AlexNet and VGG16

Table 7 shows the low-rank compression results of AlexNet and VGG16 for the Cifar10 dataset. Baseline and DLRT training was computed with SGD and momentum factor of 0.1. We compare to literature result, which compress the same baseline-networks. Table 8 shows the corresponding results for the Cifar100 test case. The same optimizer settings as in Cifar10 are used.

Table 7: Compression results of VGG16 and AlexNet on Cifar10. The compression ratio (c.r.) is the percentage of parameter reduction with respect to the full model ($< 0\%$ indicates that the ratio is negative). "ft" indicates that the model has been fine tuned.

method	test acc.	c.r. [eval]	c.r. [train]
VGG16	**93.9**%	0%	0%
DLRT $\tau = 0.1$	92.01%	56%	77.5%
GAL [40]	92.03%	77%	$< 0\%$
LRNN [27]	92%	60%	$< 0\%$
AlexNet	**83.16**%	0%	0%
DLRT $\tau = 0.1$	81.37%	86.3%	84.2%
NISP [55] (ft)	82.1%	$n.a.$	$< 0\%$

Table 8: Compression results of on Cifar100. DLRT with $\tau = 0.08$ is used.

method	baseline acc.	test acc.	c.r. [eval]	c.r. [train]
VGG16	58.5%	60.02%	51%	72%
AlexNet	55%	51.12%	85.1%	79.2%

6.6 Detailed training performance of low-rank pruning

The proposed low-rank training algorithm does not need to be applied to train a network from random initial weight guesses. When an already trained network is available, the proposed method can be employed as a memory-efficient pruning strategy. A straightforward approach to reduce a trained fully-connected network to a rank r network is to compute an SVD for all weight matrices and to truncate those decompositions at rank r. However, while this choice is optimal to present weight matrices, it might significantly reduce the accuracy of the network. Hence, retraining the determined low-rank subnetwork is commonly necessary to obtain desirable accuracy properties. Three key aspects are important to obtain an efficient pruning method for low-rank methods:

1. Retraining preserves the low-rank structure of the subnetwork.
2. Retraining does not exhibit the memory footprint of the fully connected network.
3. Retraining finds the optimal network among possible low rank networks.

Let us note that the attractor of the proposed dynamical low-rank evolution equations fulfills these three requirements. Recall that for the evolution equations we have (3):

$$\min \left\{ \|\dot{W}_k(t) + \nabla_{W_k} \mathcal{L}(W_k(t))\|_F \,:\, \dot{W}_k(t) \in \mathcal{T}_{W_k(t)} \mathcal{M}_{r_k} \right\}. \tag{13}$$

The condition $\dot{W}_k(t) \in \mathcal{T}_{W_k(t)} \mathcal{M}_{r_k}$ ensures that the weight matrices remain of low-rank. Moreover, as previously discussed, the training method only requires memory capacities to store low-rank factors. At the attractor, i.e., when $\dot{W}_k = 0$, the last condition ensures that the attractor minimizes $\|\nabla_{W_k} \mathcal{L}(W_k(t))\|_F$. That is, the attractor is the optimal low-rank subnetwork in the sense that it picks the network with minimal gradient. To underline the effectiveness of our low-rank method as a pruning technique, we take the fully connected network from Table 5. To demonstrate the poor validation accuracy when simply doing an SVD on the full 784 by 784 weight matrices and truncating at a given smaller rank, we perform this experiment for ranks $r \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. It turns out that though reducing memory requirements, this strategy leads to unsatisfactory accuracy of about 10%, see the first column of Table 9. Then, we use the proposed low-rank training methods with fixed rank r to retrain the network. As starting points, we use the low-rank networks which have been determined by the truncated SVD. Retraining then reaches desired accuracies that are comparable to the previously determined low-rank networks in Table 5.

Table 9: Pruning methods with 784 Neurons per layer

test accuracy			Evaluation	
SVD	low-rank training	ranks	params	c.r.
98.63%	98.63%	$[784, 784, 784, 784, 10]$	2466464	0%
9.91%	98.16%	$[100, 100, 100, 100, 10]$	635040	74.25%
9.67%	98.44%	$[90, 90, 90, 90, 10]$	572320	76.80%
9.15%	98.47%	$[80, 80, 80, 80, 10]$	509600	79.34%
9.83%	98.58%	$[70, 70, 70, 70, 10]$	446880	81.88%
9.67%	98.41%	$[60, 60, 60, 60, 10]$	384160	84.42%
9.83%	98.39%	$[50, 50, 50, 50, 10]$	321440	86.97%
10.64%	98.24%	$[40, 40, 40, 40, 10]$	258720	89.51%
10.3%	98.24%	$[30, 30, 30, 30, 10]$	196000	92.05%
9.15%	97.47%	$[20, 20, 20, 20, 10]$	133280	94.60%
10.9%	95.36%	$[10, 10, 10, 10, 10]$	70560	97.14%

6.7 Detailed derivation of the gradient

In this section, we derive the computation of the gradients in the K, L and S steps in detail. For this, let us start with the full gradient, i.e., the gradient of the loss with respect to the weight matrix W_k. We have

$$\begin{aligned}\partial_{W^\ell_{jk}}\mathcal{L} &= \sum_{i_M=1}^{n_M} \partial_{z^M_{i_M}}\mathcal{L}\partial_{W^\ell_{jk}} z^M_{i_M} = \sum_{i_M=1}^{n_M} \partial_{z^M_{i_M}}\mathcal{L}\partial_{W^\ell_{jk}}\sigma_M\left(\sum_{i_{M-1}} W_{i_M i_{M-1}} z^{M-1}_{i_{M-1}} + b^M_{i_M}\right)\\ &= \sum_{i_M=1}^{n_M} \partial_{z^M_{i_M}}\mathcal{L}\sigma'_M\left(\sum_{i_{M-1}} W_{i_M i_{M-1}} z^{M-1}_{i_{M-1}} + b^M_{i_M}\right)\partial_{W^\ell_{jk}}\left(\sum_{i_{M-1}} W_{i_M i_{M-1}} z^{M-1}_{i_{M-1}}\right). \quad (14)\end{aligned}$$

For a general α, let us define

$$\sigma'_{\alpha,i_\alpha} := \sigma'_\alpha\left(\sum_{i_{\alpha-1}} W^\alpha_{i_\alpha i_{\alpha-1}} z^{\alpha-1}_{i_{\alpha-1}} + b^\alpha_{i_\alpha}\right) \quad (15)$$

and note that for $\alpha \neq \ell$

$$\partial_{W^\ell_{jk}}\left(\sum_{i_{\alpha-1}} W^\alpha_{i_\alpha i_{\alpha-1}} z^{\alpha-1}_{i_{\alpha-1}}\right) = \sum_{i_{\alpha-1}} W^\alpha_{i_\alpha i_{\alpha-1}}\partial_{W^\ell_{jk}} z^{\alpha-1}_{i_{\alpha-1}}, \quad (16)$$

whereas for $\alpha = \ell$ we have

$$\partial_{W^\ell_{jk}}\left(\sum_{i_{\alpha-1}=1}^{n_{\alpha-1}} W^\alpha_{i_\alpha i_{\alpha-1}} z^{\alpha-1}_{i_{\alpha-1}}\right) = \sum_{i_{\alpha-1}} \delta_{j i_\alpha}\delta_{k i_{\alpha-1}} z^{\alpha-1}_{i_{\alpha-1}}. \quad (17)$$

Therefore, recursively plugging (15), (16) and (17) into (14) yields

$$
\begin{aligned}
\partial_{W^\ell_{jk}}\mathcal{L} &= \sum_{i_M=1}^{n_M} \partial_{z^M_{i_M}}\mathcal{L}\sigma'_{M,i_M}\sum_{i_{M-1}} W^\alpha_{i_M i_{M-1}}\partial_{W^\ell_{jk}} z^{M-1}_{i_{M-1}} \\
&= \sum_{i_M=1}^{n_M} \partial_{z^M_{i_M}}\mathcal{L}\sigma'_{M,i_M}\sum_{i_{M-1}} W^\alpha_{i_M i_{M-1}}\sigma'_{M-1,i_{M-1}}\sum_{i_{M-2}} W^\alpha_{i_{M-1} i_{M-2}}\partial_{W^\ell_{jk}} z^{M-2}_{i_{M-2}} = \cdots \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\partial_{W^\ell_{jk}}\left(\sum_{i_{\ell-1}=1}^{n_{\ell-1}} W^\ell_{i_\ell i_{\ell-1}} z^{\ell-1}_{i_{\ell-1}}\right) \qquad (18) \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\sum_{i_{\ell-1}}\delta_{ji_\ell}\delta_{ki_{\ell-1}} z^{\ell-1}_{i_{\ell-1}} \\
&= \sum_{i_{\ell+1},\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,j}\delta_{ji_\ell} z^{\ell-1}_{k}
\end{aligned}
$$

Written in matrix notation and making use of the Hadamard product defined as $y\circ A\circ x = (y_i A_{ij} x_j)_{ij}$, for $A\in\mathbb{R}^{m\times n}$, $x\in\mathbb{R}^n$ and $y\in\mathbb{R}^m$, we have:

$$
\partial_{W^\ell}\mathcal{L} = \partial_{z^M}\mathcal{L}^\top\left(\sigma'_\ell\circ\prod_{\alpha=\ell+1}^{M} W^\top_\alpha\circ\sigma'_\alpha\right)^\top z^\top_{\ell-1}
$$

Now, let us move to deriving the K, L and S-steps for the dynamical low-rank training. For the K-step, we represent the weight matrix W_ℓ as $W^\ell_{i_\ell i_{\ell-1}} = \sum_m K^\ell_{i_\ell m}V^\ell_{i_{\ell-1}m}$. Hence, reusing the intermediate result (18) yields

$$
\begin{aligned}
\partial_{K^\ell_{jk}}\mathcal{L} &= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\partial_{K^\ell_{jk}}\left(\sum_{i_{\ell-1}=1}^{n_{\ell-1}}\sum_m K^\ell_{i_\ell m}V^\ell_{i_{\ell-1}m} z^{\ell-1}_{i_{\ell-1}}\right) \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\sum_{i_{\ell-1}=1}^{n_{\ell-1}}\sum_m \delta_{ji_\ell}\delta_{km}V^\ell_{i_{\ell-1}m} z^{\ell-1}_{i_{\ell-1}} \\
&= \sum_{i_{\ell+1},\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\sum_{i_{\ell-1}=1}^{n_{\ell-1}}\delta_{ji_\ell}V^\ell_{i_{\ell-1}k} z^{\ell-1}_{i_{\ell-1}}
\end{aligned}
$$

In matrix notation we obtain

$$
\partial_{K_\ell}\mathcal{L} = \partial_{z^M}\mathcal{L}^\top\left(\sigma'_\ell\circ\prod_{\alpha=\ell+1}^{M} W^\top_\alpha\circ\sigma'_\alpha\right)^\top\left(V_\ell^\top z_{\ell-1}\right)^\top = \partial_{W_\ell}\mathcal{L}V_\ell,
$$

which is exactly the right-hand side of the K-step. Hence, the K-step can be computed by a forward evaluation of $\mathcal{L}$ and recording the gradient tape with respect to K^ℓ. Similarly, for the L-step, we represent W_ℓ as $W^\ell_{i_\ell i_{\ell-1}} = \sum_m U^\ell_{i_\ell m}L^\ell_{i_{\ell-1}m}$. Hence,

$$
\begin{aligned}
\partial_{L^\ell_{jk}}\mathcal{L} &= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\partial_{L^\ell_{jk}}\left(\sum_{i_{\ell-1}=1}^{n_{\ell-1}}\sum_m U^\ell_{i_\ell m}L^\ell_{i_{\ell-1}m}\right) \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}\sum_{i_{\ell-1}=1}^{n_{\ell-1}}\sum_m U^\ell_{i_\ell m}\delta_{ji_{\ell-1}}\delta_{km} z^{\ell-1}_{i_{\ell-1}} \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L}\prod_{\alpha=\ell+1}^{M}\sigma'_{\alpha,i_\alpha}W^\alpha_{i_\alpha i_{\alpha-1}}\sigma'_{\ell,i_\ell}U^\ell_{i_\ell m} z^{\ell-1}_{j}.
\end{aligned}
$$

In matrix notation, we obtain

$$\partial_{L_\ell}\mathcal{L} = \left(U_\ell^\top \partial_{z^M}\mathcal{L}^\top \left(\sigma'_\ell \circ \prod_{\alpha=\ell+1}^{M} W_\alpha^\top \circ \sigma'_\alpha \right)^\top z_{\ell-1}^\top \right)^\top = (\partial_{W_\ell}\mathcal{L})^\top U_\ell.$$

Lastly, for the S-step we write $W^\ell_{i_\ell i_{\ell-1}} = \sum_{n,m} U^\ell_{i_\ell m} S_{mn} V^\ell_{i_{\ell-1} n}$. Then,

$$\begin{aligned}
\partial_{S^\ell_{jk}}\mathcal{L} &= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L} \prod_{\alpha=\ell+1}^{M} \sigma'_{\alpha,i_\alpha} W^\alpha_{i_\alpha i_{\alpha-1}} \sigma'_{\ell,i_\ell} \partial_{S^\ell_{jk}} \left(\sum_{n,m} U^\ell_{i_\ell m} S_{mn} V^\ell_{i_{\ell-1} n} \right) \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L} \prod_{\alpha=\ell+1}^{M} \sigma'_{\alpha,i_\alpha} W^\alpha_{i_\alpha i_{\alpha-1}} \sigma'_{\ell,i_\ell} \sum_{i_{\ell-1}=1}^{n_{\ell-1}} \sum_{m} U^\ell_{i_\ell m} \delta_{jm} \delta_{kn} V^\ell_{i_{\ell-1} n} z^{\ell-1}_{i_{\ell-1}} \\
&= \sum_{i_\ell,\cdots,i_M} \partial_{z^M_{i_M}}\mathcal{L} \prod_{\alpha=\ell+1}^{M} \sigma'_{\alpha,i_\alpha} W^\alpha_{i_\alpha i_{\alpha-1}} \sigma'_{\ell,i_\ell} U^\ell_{i_\ell j} V^\ell_{i_{\ell-1} k} z^{\ell-1}_{i_{\ell-1}}.
\end{aligned}$$

In matrix notation, we have

$$\partial_{S_\ell}\mathcal{L} = U_\ell^\top \partial_{z^M}\mathcal{L}^\top \left(\sigma'_\ell \circ \prod_{\alpha=\ell+1}^{M} W_\alpha^\top \circ \sigma'_\alpha \right)^\top \left(V_\ell^\top z_{\ell-1} \right)^\top = U_\ell^\top \partial_{W_\ell}\mathcal{L} V_\ell.$$

6.8 Low-rank matrix representation and implementation of convolutional layers

A generalized convolutional filter is a four-mode tensor $W \in \mathbb{R}^{F\times C\times J\times K}$ consiting of F filters of shape $C \times J \times K$, which is applied to a batch of N input $C-$ channels image signals Z of spatial dimensions $U \times V$ as the linear mapping,

$$(Z * W)(n, f, u, v) = \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} W(f, c, j, k) Z(n, c, u-j, v-k)\,. \tag{19}$$

In order to train the convolutional filter on the low-rank matrix manifold, we reshape the tensor W into a rectangular matrix $W^{\text{resh}} \in \mathbb{R}^{F\times CJK}$. This reshaping is also considered in e.g. [27].

An option is, to see the convolution as the contraction between an three-mode tensor Z^{unfolded} of patches and the reshaped kernel matrix W^{resh} using Pytorch's fold-unfold function. We can construct the unfold by stacking the vectorized version of sliding patterns of the kernel on the original input, obtaining in this way a tensor $Z^{\text{unfolded}} \in \mathbb{R}^{N\times CJK\times L}$, where L denotes the dimension of flatten version of the output of the 2-D convolution. Thus, equation 19 can be rewritten as a tensor mode product:

$$\begin{aligned}
(Z * W)(n, f, u, v) &= \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} W^{\text{resh}}(f, (c, j, k)) Z^{\text{unfolded}}(n, (c, j, k), (u, v)) \\
= \sum_{p=}^{r} U(f, p) \sum_{q=1}^{r} S(p, q) \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} V((c, j, k), q) Z^{\text{unfolded}}(n, (c, j, k), (u, v))
\end{aligned} \tag{20}$$

As it is shown in (20), we can a decompose the starting weight $W^{\text{resh}} = USV^\top$ and then do all the training procedure as a function of the factors (U, S, V), without never reconstructing the kernel. Then we can apply the considerations of fully connected layers.