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 is 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 a 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 [23]. 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, 16].

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 [19, 21, 41] and quantization [11, 50], with different pruning techniques, including search-based heuristics [21], reinforcement learning [3, 20] and genetic algorithms [37]. More recent work has considered pruning during training, by formulating pruning as a data-driven optimization problem [19, 22, 23]. 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 [45, 47]. 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 [16] — 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, 33, 43, 46], constrain weight matrices to have a fixed low rank during training [26], or create layers as a linear combination of layers of different rank [25]. 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, [28, 29] apply heuristics to determine the rank of the matrix decomposition ahead of time, whereas [49] 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 [35], 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 [24] 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 [30], which originates from the Dirac-Frenkel variational principle [15, 17], 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 [38] 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 [27], both in the matrix and the tensor settings [9, 31, 39, 40]. In this work, we employ the "unconventional" basis update & Galerkin step integrator [7] as well as its rank-adaptive extension [5], see also [8, 32]. 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 $\mathcal{N}$ (the convolutional setting is discussed in the supplementary material §7.5). Given an input $x \in \mathbb{R}^{n_0}$, the output $\mathcal{N}(x) \in \mathbb{R}^{n_M}$ is computed as

$$\begin{aligned} z_0 &= x, \\ z_k &= \sigma_k(W_k z_{k-1} + b_k), \qquad k = 1, \dots, M \\ \mathcal{N}(x) &= z_M, \end{aligned} \tag{1}$$

where $W_k \in \mathbb{R}^{n_k \times n_{k-1}}$ denotes the weight matrix, $b_k \in \mathbb{R}^{n_k}$ the bias vector and σ_k the nonlinear activation function of layer k. For the sake of brevity, we omit the bias vector in the following.

Given a set of data-target pairs $\{(x, y)\}$, the neural network $\mathcal{N}$ is trained using a loss function $\mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y)$ which depends on the weights, the output and the target data. The training of the neural network is based on the efficient (back)propagation of weights and biases to compute the gradients, which exploits the adjoint implementation of the chain rule:

$$\nabla_{W_k} \mathcal{L} = \sigma_k' \circ \nabla_{z_k} \mathcal{L}\, z_{k-1}^\top, \qquad \nabla_{z_k} \mathcal{L} = \prod_{l=k}^{M} \left(W_l^\top \circ \sigma_l' \right) \nabla_{z_M} \mathcal{L}, \tag{2}$$

where $\sigma_k' := \sigma_k'(W_k z_{k-1} + b_k) \in \mathbb{R}^{n_k}$ denotes the derivative of the k-th activation function and $\circ$ denotes the Hadamard product $y \circ A \circ x = (y_i A_{ij} x_j)_{ij}$, for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$.

The gradient is then employed in a gradient-based optimization algorithm. For example, in the gradient descent case the weights of the neural network at iteration $t \in \mathbb{N}$ are updated via

$$W_k^{t+1} = W_k^t - \lambda \nabla_{W_k} \mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y) \qquad \forall k = 1, \dots, M \tag{3}$$

with a learning rate λ. From (1),(2) it is clear that both the forward and gradient evaluations of the network require a large amount 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 experimental evidence shows that dense networks are typically overparameterized and that most of the weights learned this way are unnecessary. In order to reduce the number of parameters without affecting the dimension of the layers, we propose a method performing the minimization over the manifold of low-rank matrices.

To this end, we assume that every W_k is of rank $r_k \ll n_k, n_{k+1}$ and that it admits an SVD-like decomposition

$$W_k = U_k S_k V_k^\top \in \mathbb{R}^{n_k \times n_{k-1}}, \tag{4}$$

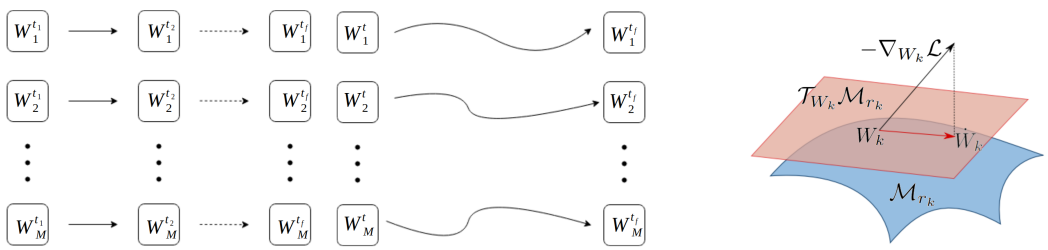
where $U_k \in \mathbb{R}^{n_k \times r_k}$, $V_k \in \mathbb{R}^{n_{k-1} \times r_k}$ are thin and tall matrices having orthonormal columns, and $S_k \in \mathbb{R}^{r_k \times r_k}$ if of full rank. Unlike standard SVD, here S_k is not assumed to be diagonal.

Traditional descent algorithms such as (3) do not guarantee the preservation of the low-rank structure (4) when updating the weight matrix W_k and require knowledge of the whole W_k rather than the low-rank 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, without forming the matrix W_k.

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

$$\dot{W}_k(t) = -\nabla_{W_k} \mathcal{L}(W_1, \dots, W_M; \mathcal{N}(x), y), \tag{5}$$

where the "dot" denotes the time-derivative. Its solution space contains all elements of the layer's weight matrix. Thus, rather than a discrete update of the weights, we interpret the training phase as a continuous evolution of the weights according to (5), as illustrated in Fig. 1(a-b). Let $\mathcal{M}_{r_k}$ denote the manifold of matrices with rank r_k and assume the input weights are in the manifold, i.e. $W_k(0) \in \mathcal{M}_{r_k}$. Using this continuous-time interpretation allows us to derive a strategy to evolve the weights according to the dynamics in (5) so that $W_k(t) \in \mathcal{M}_{r_k}$ for all t. To this end, in §3.1 we exploit the fact that $W_k(t)$ admits a time-dependent SVD-like factorization [14] $W_k(t) = U_k(t) S_k(t) V_k(t)^\top$ to rewrite (5) as a system of matrix ODEs for each of the individual factors U_k,S_k and V_k. Then, in §4 we discuss an algorithm to efficiently integrate the system of ODEs. Remarkably, using the dynamics of the individual factors will allow us to adaptively adjust the rank r_k throughout the continuous-time training process.



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

Figure 1: 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 $W_k(t)$.

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 wish to find

$$\dot{W}_k(t) \in \mathcal{T}_{W_k(t)}\mathcal{M}_{r_k} \quad \text{such that} \quad \|\dot{W}_k(t) + \nabla_{W_k}\mathcal{L}(W_k(t))\|_F = \text{min!} \tag{6}$$

where $\mathcal{T}_{W_k(t)}\mathcal{M}_{r_k}$ is the tangent space of $\mathcal{M}_{r_k}$ at position $W_k(t)$. That is, we wish to make sure that our update remains of rank r_k and at the same time minimizes the residual, i.e. solves (5). The rank r attractor of the above system is optimal in the sense that it minimizes the gradient norm $\|\nabla_{W_k}\mathcal{L}\|_F$ for $k \in \{1, \cdots, M\}$, as $\dot{W}_k = 0$. A key observation of DLRA is that (6) can be restated as a Galerkin condition, graphically illustrated in Fig. 1(c) and given by [30]:

$$\langle \dot{W}_k(t) + \nabla_{W_k}\mathcal{L}(W_k(t)), \delta W_k \rangle = 0 \qquad \forall \delta W_k \in \mathcal{T}_{W_k(t)}\mathcal{M}_{r_k} \ . \tag{7}$$

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. [30, §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 (7) becomes

$$\langle \dot{U}_k S_k V_k^\top + U_k \dot{S}_k V_k^\top + U_k S_k \dot{V}_k^\top + \nabla_{W_k}\mathcal{L}(W_k(t)), \delta W_k \rangle = 0, \qquad \forall \delta W_k \in \mathcal{T}_{W_k(t)}\mathcal{M}_{r_k} \tag{8}$$

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 (8), 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 (8) 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:

$$\begin{cases} \dot{S}_k = -U_k^\top \nabla_{W_k}\mathcal{L}(W_k(t)) V_k \ , \\ \dot{U}_k = -(I - U_k U_k^\top)\nabla_{W_k}\mathcal{L}(W_k(t)) V_k S_k^{-1} \ , \\ \dot{V}_k = -(I - V_k V_k^\top)\nabla_{W_k}\mathcal{L}(W_k(t))^\top U_k S_k^{-\top} \ . \end{cases} \tag{9}$$

4 KLS-based training algorithm

In order to perform an efficient rank-constrained training step, we numerically integrate the system of ODEs (9). 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 (9), 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

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

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

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

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

from $t = 0$ to $t = t^*$, 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, 18]. 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 $[K_k(t^*) \mid U_k]$ and $[L_k(t^*) \mid V_k]$, 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 §4.2 and §5.

4.1 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 gradient flow of 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

$$z_k = \sigma_k\left(K_k(t)V_k^\top z_{k-1} + b_k\right), \qquad k = 1, \dots, M \tag{10}$$

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 §7.4.

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$\big\{\dot{b}(t) = -\nabla_b \mathcal{L}(U_k^{t+1} S_k^{t+1} (V_k^{t+1})^\top z_{k-1} + b(t)), b(0) = b_k^t\big\}$

4.2 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 $t = t^*$. 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 [44]. 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 t^* 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 (10) 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 §4.1 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 our algorithm's efficiency is that all weight matrices are of rank $r_k \ll n_k, n_{k+1}$. When the truncation threshold ϑ is too small Alg. 1 does not provide advantages with respect to standard training. This is also emphasized in Fig. 2. Moreover, 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 perform well, yielding competing performance with respect to competing methods and the full-rank baselines.

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 [42], 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.

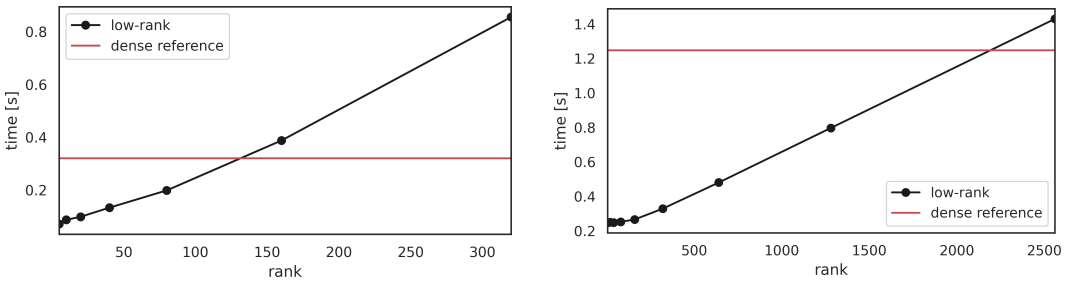
5.1 Fixed-rank fully-connected feed-forward network timings

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 2(a) and 2(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

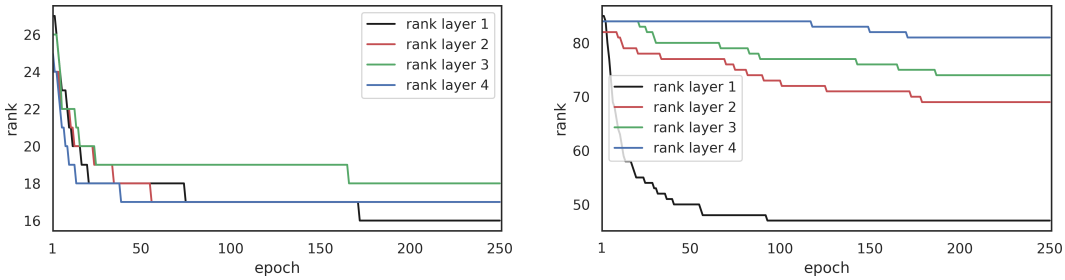
Next, we demonstrate the capabilities of the dynamical low-rank approach to determine the rank of the network's weight matrices automatically during the network training using Algorithm 1. The Adam optimizer with default learning rate is used for the gradient update. We train fully connected 5-layer 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.

Figures 3 (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



(a) Training timings (b) Prediction timings

Figure 2: 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.



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

Figure 3: Rank evolution for layers 1-4 of a 5-layer $[500, 500, 500, 500, 10]$ fully-connected network on MNIST.

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 4 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 §7.2. Finally, in the supplementary material §7.3 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 [34] on MNIST, against the full-rank reference and several baselines. LRNN [24] is the closest approach to our DLRT: it dynamically trains low-rank layers by adding a rank-penalty to the loss function, and by complementing the standard training step via an SVD projection step. While computing low-rank factors for each layer, thus reducing memory storage of the network, this training approach is more

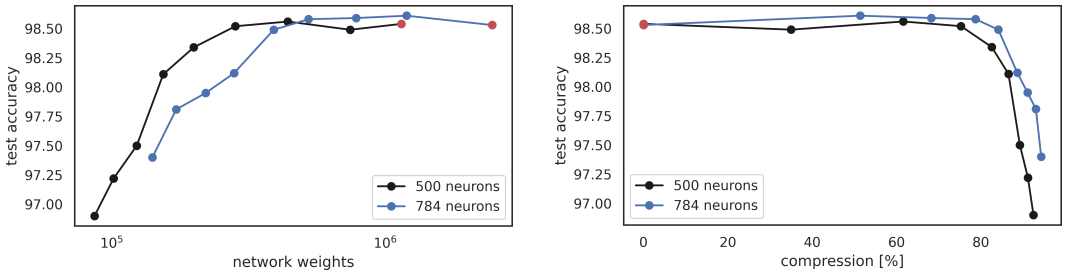


Figure 4: Mean test accuracy over network parameter count and compression rate for 5 runs with randomly sampled train-test-val sets. The red dot denotes the baseline full-rank network.

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 [51] (ft)		99.18%		110000	74.4%		$< 0\%$
NISP [48] (ft)		99.0%		100000	76.5%		$< 0\%$
GAL [36]		98.97%		30000	93.0%		$< 0\%$
LRNN [24]		98.67%	$[3, 3, 9, 9]$	18075	95.8%		$< 0\%$

expensive than training the full network. GAL [36], SSL [51], and NISP [48] 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 LNRR, these methods do not reduce the computational cost of the training phase. Analogously to [24], 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 §7.5. 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.

6 Summary and Conclusion

In this work, we present a dynamical, adaptive low-rank training algorithm for deep neural networks that allow us to find efficient low-rank subnetworks directly during training time. For large networks, the proposed algorithm has lower memory footprint than traditional full rank training and is computationally more efficient, while maintaining competitive levels of neural network accuracy. Demonstration on test cases show that the approach is able to find low-rank manifolds in the parameter space of the network and train the network without knowledge of the full parameter space. This enables online training of large networks on limited-resources devices.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Y. Idelbayev and M. 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Algorithm 1, Figure 2 and Table 1.
   (b) Did you describe the limitations of your work? [Yes] See Section 4.2
   (c) Did you discuss any potential negative societal impacts of your work? [No] This paper is on foundations and proposes a novel learning paradigm based on ODE, so not applicable.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] All the authors have read the etichs review guidelines and confirm that the proposed paper conform to them.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Everything has been included.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] We provided all the training details.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We report mean performance in the main paper and error bars showing standard deviation in the supplementary material.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 5.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Section 5 and Section 5.3 .
   (b) Did you mention the license of the assets? [Yes] See Section 5.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]