Università degli studi di Padova

Department of Mathematics

Master Thesis in Data science

Dynamical low-rank training of neural networks

Supervisor
Prof. Francesco Rinaldi
Università degli studi di Padova

Co-supervisor
Prof. Francesco Tudisco
Gran Sasso science institute

Master Candidate
Emanuele Zangrando 2027817

Academic Year
2021-2022

To my parents, who always supported me during my path. In memory of my uncle Massimo and of my high-school professor Antonio, without whom I would not have inherited my passion for mathematics.

Abstract

Neural networks have achieved tremendous success in a large variety of applications. However, their space and time computational demand can limit their usage in resource limited devices. At the same time, overparameterization seems to be necessary in order to overcome the highly non-convex nature of the training optimization problem. An optimal trade-off is then to be found in order to reduce networks' dimension while mantaining high performance. Popular approaches in the current literature are based on pruning techniques that look for subnetworks able to mantain approximately the initial performance.

Nevertheless, these tecniques are often not able to reduce the memory footprint of the training phase. In this thesis we will present DLRT, a training algorithm that looks for "low-rank subnetworks" by using DLRA theory and techniques. These subnetworks and their ranks are determined and adapted already during the training phase, allowing the overall time and memory resources required by both training and evaluation phases to be reduced significantly.

Contents

Abstract v

List of figures ix

List of tables xi

Listing of acronyms xiii
0.1 Notation . . . 1

1 Introduction 3

2 Theoretical background 5
2.1 Differential geometry basics . . . 5
2.2 Neural network basics . . . 9
2.3 Theory introduction . . . 10
2.4 Training as gradient flow . . . 12
2.5 Constraints and regularization . . . 12
2.6 Flow in the low-rank manifold . . . 15
2.7 Low rank manifold constraint . . . 16
2.7.1 Derivation of the projection operator . . . 16
2.7.2 Projector-splitting integrator . . . 21
2.7.3 Unconventional robust integrator . . . 22
2.7.4 Rank-adaptive unconventional integrator . . . 23

3 The lottery ticket hypothesis and dynamical low-rank training 25
3.0.1 Pruning and lottery tickets . . . 25
3.0.2 Overview of DLRT method [27] . . . 26
3.1 Efficient gradient taping . . . 28
3.2 Training procedure description . . . 31
3.3 Low-rank lottery tickets . . . 33

4 Comparison with other methods 35
4.1 Structured sparse learning . . . 35
4.2 Rethinking smaller-norm less-informative assumption . . . 36
4.3 Pruning via GAL . . . 36
4.4 Low-rank compression of neural networks . . . 37
4.5 Singular Vector Orthogonality Regularization and Singular Value Sparsification . . . 39

5 Cost analysis and experiments 41
5.1 Cost analysis . . . 41
5.1.1 Efficient gradient taping . . . 41
5.1.2 Efficient forward phase . . . 44
5.2 Compression effects . . . 45
5.2.1 Timings . . . 45

5.2.2 Compression ratio and accuracy 48
5.3 Lenet on MNIST 48
5.4 Cifar10 and Cifar100 50
5.5 Robustness to small singular values 51
5.6 Fine tuning effectiveness 52

6 Implementation 55
6.1 Low-rank Module 55
6.1.1 Populate_gradients method 55
6.2 Optimizer class 56
6.2.1 Preprocess steps 57
6.2.2 Integration steps 57
6.2.3 Postprocess steps 57
6.2.4 Step method 57
6.2.5 Fine tuning 57

7 Conclusions and future improvements 59

References 61

Acknowledgments 65

Listing of figures

2.1 coordinate representation of a map between manifolds. Image taken from [26]. 7
2.2 Graphical representation of Galerkin condition for projecting the vector field on the tangent space of low-rank matrices. Image taken from [27]. 17

3.1 Full and low-rank representation of neural networks with the associated gradient flow problems. 29
3.2 Lenet5 representations of the K,L and S steps for the gradient efficient implementation of Alg.3.1. 34

5.1 Cpu timings of different operations on full Mnist as a function of the layerwise compression ratio. The coloured area around each line indicates one standard deviation interval. 47
5.2 Layerwise compression ratio against overall compression ratio during different phases, to compare with Fig.5.1 47
5.3 layerwise compression ratio against test accuracy after training Lenet5 on MNIST. The blue line represents the overall memory compression (counting the gradients). 48
5.4 Relative singular value threshold τ of the adaptive version of Alg.3.1 against test compression ratio, train compression ratio (counting the gradients) and test accuracy (with relative full rank vanilla training baseline). Detailed numerical results are presented in Tab.5.2. . . 49
5.5 Accuracy over epochs of DLRT and and layer factorization [13] on Lenet5 architecture trained on Mnist. Decay with powers of 10 (top left), with powers of 2 (top right), and no decay (bottom). 52
5.6 Training loss over epochs of DLRT and and layer factorization [13] on Lenet5 architecture trained on Mnist. Decay with powers of 10 (top left), with powers of 2 (top right), and no decay (bottom). 53

Listing of tables

5.1 Fully connected architecture was used for the timing experiment. The input is a flattened image of Mnist, of size 784. 46
5.2 Results of the training of LeNet5 on MNIST dataset. "Params" represent the number of parameters we have to save (in train and inference phases) using DLRT Alg.3.1. The compression ratio (c.r.) is the percentage of parameter reduction with respect to the full model ($< 0\%$ indicates that the ratio is negative), as explained in section 5.1.1 (for the train compression ratio we took the gradients into account). "ft" indicates that the model has been fine tuned. "LeNet5" line indicates the baseline, trained using stochastic gradient descent. 50
5.3 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. The "difference with baseline" column indicates the difference in final test accuracy between each method and the baseline (full rank,trained with stochastic gradient descent with the same hyperparameters). 51
5.4 Compression results of on Cifar100. DLRT with $\tau = 0.08$ is used. The "difference with baseline" column indicates the difference in final test accuracy between each method and the baseline (full rank,trained with stochastic gradient descent with the same hyperparameters). 51
5.5 Effects of fine tuning on underparametrized networks. In the table we report the final test accuracy without fine tuning (first column), and after fine tuning (second column). In the third column, we report the mean improvement after fine tuning. In the last column we report the overall test compression ratio achieved during the training. In each column a standard deviation across five independent runs is reported. 53

Listing of acronyms

DLRT .... Dynamical low-rank training

GAL .... Generative adversarial learning

GAN .... Generative adversarial network

SGD .... Stochastic gradient descent

SVD .... Singular value decomposition

0.1 Notation

1. $W_{ij}^{(\alpha)}$ the (i, j) entry of the weight matrix in layer α;
2. for data points we used upper indices (e.g., $(x^{(i)}, y^{(i)})$) to indicate the number of the sample;
3. In general, vectors will be indicated with lower-case letters, while matrices and tensors will be indicated with upper-case ones;
4. For any couple of matrices $A, B \in \mathbb{R}^{n \times m}$ we will indicate with $\langle A, B \rangle = Tr(A^\top B)$ the Frobenius inner product. Since it is symmetric, it holds $\langle A, B \rangle = Tr(B^\top A) = Tr(AB^\top) = Tr(BA^\top)$;
5. throughout the thesis, we will indicate with $\|W\|_p$ the L^p norm for matrices $W \in \mathbb{R}^{n \times m}$ seen as elements of $\mathbb{R}^{nm}$;
6. we will indicate with $\times_i$ the $i - th$ mode product between a tensor and a matrix. If $T \in \mathbb{R}^{n_1 \times \dots n_m}$ and $M \in \mathbb{R}^{k \times n_i}$, the $i - th$ mode product $T \times_i M \in \mathbb{R}^{n_1 \times \dots n_{i-1} \times k \times n_{i+1} \times \dots \times n_m}$ is defined as:

$$(T \times_i M)_{j_1, \dots, j_{i-1}, t, j_{i+1}, \dots, j_m} = \sum_{\ell=1}^{n_i} T_{j_1, \dots, j_{i-1}, \ell, j_{i+1}, \dots, j_m} M_{t, \ell}$$

7. we will indicate with $GL_r(\mathbb{R})$ the general linear group of order r, i.e. the set of real invertible matrices of size $r \times r$:

$$GL_r(\mathbb{R}) = \{M \in \mathbb{R}^{r \times r} \,|\, det(M) \neq 0\}$$

8. we will indicate with $\mathcal{O}_{n \times r}(\mathbb{R})$ the set of $n \times r$ matrices with orthonormal columns:

$$\mathcal{O}_{n \times r}(\mathbb{R}) = \{U \in \mathbb{R}^{n \times r} \,|\, U^\top U = I_{r \times r}\}$$

1
Introduction

Since the development of the necessary computational tools, neural networks have been ubiquitous in society, allowing to solve problems whose solutions were otherwise really difficult to approach algorithmically. Despite the wealth of recent improvements, artificial neural networks and deep learning are still highly active macro-areas of research, both due to the growing dimension of the networks in use and to the limited theoretical understanding of their underlying functioning.

It has been known for long [1] that in the majority of cases neural networks are strongly overparametrized. This has also been shown "constructively" in recent works, such as [9]. In the latter work, the authors showed experimentally that any network possesses a heavily sparse trainable subnetwork whose performance is at least as good as the full one. This observation opens really interesting improvement scenarios: if one were able to identify the "lottery ticket" subnetworks, it would be possible to save space and time resources both during training and inference phases, allowing bigger architectures to run on limited resource devices and to potentially avoid overfitting. Nevertheless, it is really difficult to identify good-performing sparse subnetworks without having to train the full network, and even though this can improve a lot in the inference phase, it does not help in reducing the need of resources to train deep networks. Moreover, sparsity is just one instance of possible compression strategies. Potentially, any "well-structured" low-dimensional constraint in the hypothesis space can give rise to a similar result. However, if one aims at reducing training costs, a key requirement is that the constraint has to give rise to an easily treatable training optimization problem. The specification of a nice geometric structure for the constraint is a big help, since it allows us to design efficient algorithms that exploit the geometry. From a Bayesian perspective, this constraint is often tackled by using strong priors in the hypothesis space, and these techniques have already been heavily explored in research [8, 37, 40]. Despite their good performance and simplicity of usage, these techniques do not allow for dimensionality reduction during the training phase, and given the growing size of modern architectures this poses strong limits of training under computationally restrictive conditions.

The general idea of our work is to assume a suitable geometrical structure on the constraint space, that in our case will be a manifold embedded in the hypothesis space, and use the fact that we are able to put a nice

coordinate system on it, that in our case will be differentiable. Thus, the key of our proposal is to reinterpret the training optimization problem as a system of tensor gradient flow problems on the manifold. Instead of following the unconstrained gradient flow and pointwise projecting on the manifold, we can project the vector field of the ODE to the tangent bundle. After having done that, we can "pullback"the ODE using the charts on the manifold and follow the flow in an Euclidean space without constraints. More than this, we would like to have a whole family of manifolds which are able to capture increasingly large constraints covering the whole hypothesis space, and we would like our gradient flow to adapt and stabilize on the "correct" manifold of this family.

In this context, the family of manifolds under consideration is $\{\mathcal{M}_{r_1,..,r_L}\}_{r_1,..,r_L}$ where $\mathcal{M}_{r_1,..,r_L}$ is the product manifold of matrices of rank respectively $r_1, ..., r_L$. The goal of this thesis is to apply methods coming from the theory of model order reduction of ODEs in order to train a neural network constrained to a manifold of this kind. Moreover, the resulting algorithm will be able to automatically adapt the ranks $r_1, .., r_L$ and stabilize on the "right manifold".

During this thesis' work, I have been focusing in particular on methods specifically constructed to tackle problems of this kind. These methods are presented in [5, 15, 18]. In our work, we applied techniques coming from these papers in the world of neural networks, with a particular attention towards [18]. The resulting approach allows us to significantly reduce space and time complexity both during the training and evaluation phases. Moreover, the ranks of the layers are selected adaptively by looking at the relative weight of the tail of singular values in each layer.

My thesis' work was mainly focused on the implementation of a Pytorch optimizer for this training algorithm. We also collaborated with other universities to produce a first version of a preprint that is meant to be expanded in future. The preprint of our work is publicly available at [27] along with the implementations at https://github.com/compile-gssi-lab/DLRT/tree/efficient_gradient (the one used in this thesis) and its first version at https://github.com/compile-gssi-lab/DLRT.

2
Theoretical background

In this chapter we will introduce some theoretical background concerning the main idea behind dynamical low-rank training (DLRT [27]).
This chapter is meant to be an introduction to the minimum amount of tools needed to formalize the dynamical low-rank training algorithm, with a particular attention on “non-standard” topics for a data scientist.

2.1 Differential geometry basics

Differential geometry is one of the cornerstones of modern science and deep learning is no exception. Its applications in this area are wide, going from unsupervised manifold learning [12], autoencoders [20], normalizing flows [14], homology inference [6] to information geometry [24].
The particular application we are interested in this context is to describe constraints in the hypothesis space of a neural network. In this chapter we will introduce the basic notation and embedded differential geometry concepts that will be needed in the next sections. Because some definitions may differ from one book to another, we will refer to the terminology used in [19] .
The main motivation behind the development of differential geometry was the need to extend analysis from Euclidean “flat” spaces to “curved” ones. Since the majority of concepts in analysis is “local”, it is reasonable to extend this theory to spaces that are locally “similar” to Euclidean ones. With no other structure added and some other technical requirement, this lead to the general definition of **topological manifold**.
In simple words, a topological space is a set with the weakest possible definition of closeness between its points. A topological manifold of dimension n is a topological space whose topological properties a locally indistinguishable from an open set of the euclidean space $\mathbb{R}^n$, even though globally it can be quite different. More formally:

Definition 2.1.1 (Topological manifold) *A manifold $\mathcal{M}$ is a second countable $T2$ topological space $(\mathcal{M}, \tau)$ such that for every $p \in \mathcal{M}$ there exists an open neighborhood of it $\mathcal{U}_p$ and a map ϕ such that*

$$\phi\colon \mathcal{U}_p \to \phi(\mathcal{U}_p) \subset \mathbb{R}^n$$

is an homeomorphism. The map ϕ is said to be a local coordinate chart around p and n is said to be the dimension of the manifold.
Moreover, the manifold $\mathcal{M}$ is said to be embedded in an ambient space $\mathcal{A}$ if it is a subset of it.

Up to now there's no notion of differentiability in this definition. In order to be able to do analysis, we need to introduce a notion of "smoothness" in system of local coordinates.

Definition 2.1.2 (Atlas) *Let $\mathcal{M}$ be a topological manifold and let $\{(\mathcal{U}_\alpha, \phi_\alpha)\}_\alpha$ be a set of charts such that the union of the open sets $\mathcal{U}_\alpha$ covers the whole manifold. Then the set $\{(\mathcal{U}_\alpha, \phi_\alpha)\}_\alpha$ is said to be an **atlas** of $\mathcal{M}$.*
Moreover, the atlas is said to be of class $\mathcal{C}^\ell$ if for any couples of open sets of the covering $\mathcal{U}_\alpha, \mathcal{U}_\beta$ with $\mathcal{U}_\alpha \cap \mathcal{U}_\beta \neq \emptyset$, the transition map

$$\phi_\alpha \circ \phi_\beta^{-1}\colon \phi_\beta(U_\alpha \cap U_\beta) \subset \mathbb{R}^n \to \phi_\alpha(U_\alpha \cap U_\beta) \subset \mathbb{R}^n$$

is a diffeomorphism of class $\mathcal{C}^\ell$.

This additional requirement on the atlas can be interpreted as a smoothness in the change of coordinates on the manifold, or as a sort of "compatibility" condition on the charts of the atlas.

Definition 2.1.3 (Differentiable manifold) *A topological manifold $\mathcal{M}$ endowed with a $\mathcal{C}^1$ atlas is said to be a differentiable manifold.*

To not complicate the definitions, since it is of no interest for the future discussion, we are going to present all the next definition/results thinking about the embedded scenario.
The next definition is of huge importance when it comes to studying ordinary differential equations on manifolds. As Definition 2.1.3 states, a differentiable manifold is essentially just a topological manifold endowed with an additional atlas in the which change of coordinates between one patch and the other behave "well". As in Euclidean spaces, every curve lying on the manifold has a pointwise velocity vector, tangential to the curve. In a flat euclidean space $\mathbb{E}$, where any curve has no constraint, the space spanned by the velocities of all curves passing through a particular point p is called tangential space of $\mathbb{E}$ at the point p, and in this simple case it is just isomorphic to the vector space underlying $\mathbb{E}$. The same definition can be given for manifolds, and the elements of the tangent space can be interpreted as the "possible directions of movement" from a point in order to stay on the manifold. This premise leads to the following definition:

Definition 2.1.4 (Tangent space) *Let $\mathbb{E}$ be an Euclidean space with V as underlying vector space. Let $\mathcal{M} \subset \mathbb{E}$ be a differentiable manifold and let $p \in \mathcal{M}$. The tangent space of $\mathcal{M}$ at the point p, is defined as:*

$$T_p\mathcal{M} = \{\dot\gamma(0) \in V \mid \gamma \in \mathcal{C}^1((-1,1);\mathcal{M}), \gamma(0) = p\} \subset T_p\mathbb{E} \tag{2.1}$$

It is convenient for some other definitions to introduce an equivalence relation in the set

$$\Gamma_p(\mathcal{M}) := \{\gamma \mid \gamma \in \mathcal{C}^1((-1,1);\mathcal{M}), \gamma(0) = p\} \tag{2.2}$$

by saying $\gamma_1 \sim \gamma_2$ *if* $\dot{\gamma}_1(0) = \dot{\gamma}_2(0)$.
With this additional definition, we can identify $T_p\mathcal{M}$ *with the quotient* $\Gamma_p\mathcal{M}/\sim$.

Observation 2.1.1 *The last definition 2.1.4 is given in the context of manifolds embedded in Euclidean spaces. For the general case one can either follow the same approach but using charts, or otherwise more commonly use the derivations approach [19].*

Observation 2.1.2 *(Tangent spaces are vector spaces)*
The main idea of the proof can be seen by taking two tangent vectors at a point p, $v_1 = \dot{\gamma}_1(0), v_2 = \dot{\gamma}_2(0)$. *Without loss of generality, we can choose a chart containing p such that* $\phi(p) = 0$ *(or otherwise we can consider* $\tilde{\phi}(q) = \phi(q) - \phi(p)$*). Then consider the curve* $t \mapsto \sigma(t) := \phi^{-1}\Big(\phi(\gamma_1(t)) + \lambda\phi(\gamma_2(t))\Big)$. *It is* $\sigma(0) = \phi^{-1}\Big(\phi(p) + \lambda\phi(p)\Big) = \phi^{-1}\Big(0\Big) = p$.
Now, any tangent vector can be represented as an equivalence class of curves up to reparametrization. It can be shown that the map

$$\begin{aligned}\phi_*\colon T_p\mathcal{M} &\to T_0\mathbb{R}^n\\ [\gamma] &\mapsto [\phi\circ\gamma]\end{aligned}$$

is bijective up to equivalence class of curves. Moreover, $T_0\mathbb{R}^n$ *is naturally endowed with a vector space structured. There is only one possible way to endow* $T_p\mathcal{M}$ *with a linear structure such that* ϕ_* *is an isomorphism, and it is done by pulling back with* ϕ_*^{-1}. *This shows that* $T_p\mathcal{M}$ *has a rather natural vector space structure, and it can moreover be shown that this does not depend on the choice of the chart.*

Example 2.1.1 *Let* $\mathcal{M} = \mathbb{R}^n$ *with the usual differentiable structure* $\{(\mathbb{R}^n, id)\}$ *and let* $p \in \mathbb{R}^n$. *Consider the set of coordinate curves* $\{\gamma(t) = p + te_i | i = 1, ..n\}$ *where* e_i *are the canonical basis vectors. Then the derivatives of this curves at* $t = 0$ *span the whole* $\mathbb{R}^n$*, so we can say that* $T_p\mathbb{R}^n \cong \mathbb{R}^n$.

The next definition extents the definition of differential of maps between coordinate spaces.

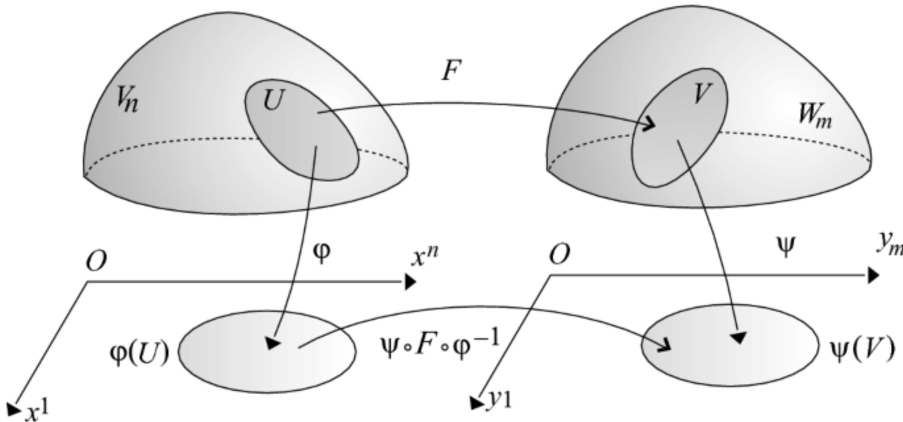

Figure 2.1: coordinate representation of a map between manifolds. Image taken from [26].

Definition 2.1.5 (Pushforward) *Let $\mathcal{M}, \mathcal{N}$ be two differentiable manifolds and let*

$$F \colon \mathcal{M} \to \mathcal{N}$$

be a map. F is said to be of class $\mathcal{C}^k$ if for any $p \in \mathcal{M}$ and any couple of charts $(\mathcal{U}_p, \phi)$, $(\mathcal{U}_{F(p)}, \psi)$, the coordinate representation of F, $\tilde{F} = \psi \circ F \circ \phi^{-1}$, is of class $\mathcal{C}^k$.
*The **pusforward** (or **differential**) of F at a point p, denoted with dF_p is the linear map defined as follows:*

$$dF_p \colon T_p\mathcal{M} \to T_{F(p)}\mathcal{N}$$
$$[\gamma] \mapsto (F \circ \gamma)^{'}(0)$$

Moreover, the definition is well posed (does not depend on the representative of the class).

This last definition is important for the next two examples, in which we will present the two most common ways in which manifolds are represented.

Example 2.1.2 *(Zeros of regular functions) Let V and W be two finite dimensional real vector spaces. Without loss of generality, we can choose basis for these two spaces and think of them as $V \cong \mathbb{R}^n$ and $W \cong \mathbb{R}^m$. Let*

$$F \colon \mathbb{R}^n \to \mathbb{R}^m$$

*be a map of class $\mathcal{C}^1$. Define $\mathcal{M} := F^{-1}(0)$. Then if $dF(p)$ has the same rank r for each $p \in F^{-1}(0)$ (in this case we call F a **submersion**), $\mathcal{M}$ is a $\mathcal{C}^1$ differentiable manifold of dimension $n - r$.*
It can be seen easily that the tangent space $T_p\mathcal{M}$ is exactly the kernel of the differential $dF(p)$.

Example 2.1.3 *(Trough parametrization) By definition 2.1.3, every differentiable manifold is endowed of a atlas structure with local charts. Given a local chart ϕ we can consider its inverse ϕ^{-1}, that is still a diffeomorphism. Since ϕ^{-1} is a local parametric representation of the manifold, we call the inverse of a local chart a **local parametrization**. With this definition we can describe properties of the manifold locally using the parametrization. Let Ω be an open subset of $\mathbb{R}^k$ and let*

$$\sigma \colon \Omega \subset \mathbb{R}^k \to \mathbb{R}^n$$

*be a diffeomorphism of class $\mathcal{C}^1$. Define $\mathcal{M} := im(\sigma)$. Then if $d\sigma(p)$ has the same rank r for each $p \in \Omega$ (in this case we call σ an **immersion**), $\mathcal{M}$ is a differentiable manifold of dimension r inside $\mathbb{R}^n$. In can be shown that the tangent space $T_p\mathcal{M}$ is exactly the image of the differential $d\sigma(p)$.*

It is important to notice that, even though not all manifolds can be represented globally through a parametrization or as set of zeros of a regular function, this can always be done locally.

Example 2.1.4 *(rank-r matrices) An example of our interest is the manifold $\mathcal{M}$ of rank r matrices. As embedding space, let's consider the space of matrices $\mathbb{R}^{n \times m}$ and let $r \leq \min\{n, m\}$. A known characterization of the rank is the one using the largest order of a non-vanishing minor. In particular, for any rank r matrix A, all minors of order $r + 1$ are zero.*

Defined

$$\mathcal{S} := \left\{ X = \left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right) \in \mathbb{R}^{n\times m} \mid A \in GL_r(\mathbb{R}) \right\} \tag{2.3}$$

we can consider the Schur complement of A in X for a matrix $X \in \mathcal{S}$, $X/A := D - CA^{-1}B$. *Since the top left submatrix A is invertible, the rank of X is exactly r if and only if* $X/A = 0$. *This can be rephrased equivalently by saying that, defined* $F(X) = X/A$, *it holds* $\mathcal{M}_r \cap \mathcal{S} = F^{-1}(0)$.
Now, note that F is surjective from $\mathbb{R}^{n\times m}$ *to* $\mathbb{R}^{n-r\times m-r}$, *since already considering the image of matrices with* $C = 0, B = 0$ *we obtain the full codomain. Moreover,* F *is a submersion since* $\frac{\partial F}{\partial D} = I_{n-r\times m-r}$, *thus the differential is surjective everywhere. Using example 2.1.2, we can conclude that* $F^{-1}(0)$ *is a manifold of dimension* $nm - rank(dF) = nm - (n-r)(m-r)$. *The union of these level sets for all the possible positions of A as a submatrix of X gives the whole manifold of rank r matrices.*

2.2 Neural network basics

The historical origin of neural networks is collocated in the early 1940s, when different psychologists like Hebb, McCulloch and Pitts were trying to formulate computational models to simulate the mechanisms of learning. Thanks to the later developing of Automatic differentiation, backpropagation algorithm to the advent of GPUs, these methods started becoming computationally feasible only in the late 1980s.
Nowadays neural networks are heavily used to (approximately) describe complex processes whose algorithmic description is not clear or too complicated. In other words, neural networks are used for model identification. As a matter of fact, the foundation of neural networks lies in statistics. Indeed, the definition of neural network itself has a lot in common with the definition of statistical model, and the majority of common uses of neural networks can be regarded as statistical models. Even if a precise definition of neural network can be given in a lot of different ways, all most commonly used ones (e.g.,CNN,RNN, dense neural networks) fall in the following definition:

Definition 2.2.1 (Parametric model) *A parametric model is a parametric family of functions*

$$\begin{aligned} f\colon \mathcal{X} \times \Theta &\to \mathcal{Y} \\ (x, W) &\mapsto f(x\,; W) \end{aligned}$$

mapping some input space $\mathcal{X}$ *to some output space* $\mathcal{Y}$. *In this context* Θ *is called* ***hypothesis space****.*

Typically every set in Def.2.2.1 has a vector space structure.

Example 2.2.1 *(Feedforward neural networks)*
Feedforward neural networks are parametric models whose typical representation is obtained when $\mathcal{X} = \mathbb{R}^n, \mathcal{Y} = \mathbb{R}^m$. *This family of neural networks are often represented as compositions of L parametric affine functions* $T^\alpha(\bullet; W^{(\alpha)})$ *and L entry-wise non linear activations* $\sigma^{(\alpha)}$:

$$f(x, \{W^{(\alpha)}\}_{\alpha=1,\dots,L}) = \sigma^{(L)} \circ T^L(\bullet; W^{(L)}) \circ \cdots \circ \sigma^{(1)} \circ T^1(x; W^{(1)}) \tag{2.4}$$

In this context $W^{(\alpha)}$ *are often represented by matrices, thus the hypothesis space can be represented as a cartesian product of vector spaces.*

The pecularity of this kind of representation is that, under some hypothesis on the dimension of the matrices $W^{(\alpha)}$ and on the activation functions $\sigma^{(\alpha)}$, a family of functions of this kind can satisfy some universal approximation property. This peculiarity guarantees that, if the neural network is big enough, it can potentially approximate arbitrarly well an entire class of function with enough regularity.

For sake of clarity in the next we will refer to feed forward neural networks.
Let's suppose we want to describe a complex process (e.g., the classification of a picture according to what is represented in it), but have nothing more than a set of observations $\{(x^{(i)}, y^{(i)})\}_{i=1,\dots,n}$, where $x^{(i)}$ is to be thought as the "input" of the process (the picture) and y its corresponding "output" (the content). This kind of learning problems are called **supervised**. If the outputs $y^{(i)}$ have a discrete nature, the problem is said to be a **classification** problem, if the outputs are continuous it is said to be a **regression** problem.
Exactly as in statistical models, the goal is to exploit the "malleability" of the neural network to extract knowledge about the underlying complex process just by using the observations. This procedure boils down to the solution of a non-convex optimization problem, as better explained in the next section.

2.3 Theory introduction

In this section we introduce the main general theory of supervised learning that will be needed in the next chapters. The notation presented in this chapter will be consistently used throughout the thesis.
As always happens in the supervised learning setting, the goal is to fit a parametric family of functions to some dataset $D = (X, Y)$ where $X \in \mathbb{R}^{N \times D_1 \times \dots \times D_{\text{in}}}$ is the independent variable tensor, $Y \in \mathbb{R}^{N \times d_1 \times \dots \times d_{\text{out}}}$ is the dependent variable data tensor and N is the number of samples.
The most general assumption one can do in this kind of scenario is that each couple of data points (x, y) comes from an unknown joint probability distribution,i.e., $(x, y) \sim p_{(x,y)}$. This joint probability distribution encodes both the stochasticity in the sampling of x (described by the marginal on x) and in the regression (encoded in the conditional $p_{y|x}$).
Typically in this scenario there's some kind of sense in which one wants the model to approximate y from x, and this information can be enforced in a loss function

$$\begin{aligned} \ell \colon \mathcal{Y} \times \mathcal{Y} &\to \mathbb{R} \\ (y, \widehat{y}) &\mapsto \ell(y, \widehat{y}) \end{aligned}$$

Often, by doing some a priori assumptions on the functional form of the conditional distribution $p_{y|x}$, the loss function ℓ can be naturally chosen as the corresponding negative log-likelihood.
Let's suppose we want to fit this dataset D with a neural network:

$$\begin{aligned} f \colon X \times \Theta &\to Y \\ (x, W) &\mapsto f(x\,; W) \end{aligned}$$

In theory what one would really do is minimize the average loss with respect to the real data generating probability distribution, i.e. one would like to solve the optimization problem:

$$\arg\min_{W} \mathop{\mathbb{E}}_{(x,y)\sim p_{(x,y)}} \Big[\ell(f(x;W),y)\Big] \tag{2.5}$$

In practice, there's usually no information at all about the data generating probability distribution, apart from the samples we have in D. A solution to this problem is trying to approximate the data generating probability distribution, and the easiest way is to approximate it with a mixture of Dirac distributions concentrated in the data points, namely:

$$p_{\text{data}}(x,y) = \frac{1}{N}\sum_{j=1}^{N} \delta\Big((x,y) - (x^{(j)}, y^{(j)})\Big)$$

If we think D is representative enough for $p_{(x,y)}$, we have decent pointwise estimator for the expected loss function and solve the optimization problem:

$$\arg\min_{W} \mathop{\mathbb{E}}_{(x,y)\sim p_{\text{data}}} \Big[\ell(f(x;W),y)\Big] \tag{2.6}$$

This translates into the average loss being just the mean of the loss calculated on the dataset D. Despite the difficulty of finding a global minimum (the loss function is almost never convex apart from trivial cases), one can try to find a good local minimum of Eq.(2.6) in different ways. One of the simplest and most used technique, which can outperform more complicated approaches [41], is (minibatch) gradient descent:

$$\begin{cases} W_{t+1} = W_t - \eta\nabla \mathop{\mathbb{E}}_{(x,y)\sim p_{\text{batch}(t)}} \Big[\ell(f(x,W_t),y)\Big] \\ W_0 \quad \text{given} \end{cases} . \tag{2.7}$$

Where at each step the expectation is taken over a small subset of the whole dataset (called batch), which changes at each step. The rationale behind this technique is statistical: the gradients we are calculating are estimators of the true gradients (to which we don't have access). Using a small subsample the whole dataset is often enough to obtain a good approximation. Also, sometimes it is computationally not feasible to use the entire dataset, leaving no other option but to use smaller batches. Moreover, using the minibatch has some advantages with respect to full batch gradient step:

1. the first advantage is computational. Calculating the gradient of the loss with full batch is much more expensive than calculating it on a batch. Moreover, often already medium sized batches give good estimations;

2. The second advantage can be attributed to the variance of the gradient estimator. The bigger the batch size, the smaller the variance of the gradient estimation. Having a small variance is not often a good choice, since a more noisy estimation can help escaping "bad" local minima [30].

3. Again, as shown in [30], a small batch size can have a regularization effect, avoiding overfit on the train set.

In order to simplify notation, from now on we will indicate with $J(W;X,Y)$ the expectation of the loss with respect to p_{data}.
In the next section we will take a more careful look at Eq.(2.7), since it's the bridge between optimization and ordinary differential equations.

2.4 Training as gradient flow

The full batch gradient descent version of equation (2.7) can be seen as a forward Euler step applied to the gradient flow ODE:

$$\begin{cases} \dot{W}(t) = -\nabla J(W(t);X,Y) \\ W(0) = W_0 \end{cases} . \tag{2.8}$$

The situation complicates a bit more when we're using minibatches, since the vector field inherits a stochastic component given by the fluctuations of the estimator on each subsample of the original dataset. Nevertheless, if the minibatches are choosen from the dataset in a deterministic way, the problem can be rephrased in a similar way by loosing the autonomous property of the system of ODE:

$$\begin{cases} \dot{W}(t) = -\nabla J(W(t);X(t),Y(t)) \\ W(0) = W_0 \end{cases} . \tag{2.9}$$

where $(X(t),Y(t))$ is the minibatch choosen at time t.
This observation is really important, since it allows us to apply any numerical integration method in order to solve (2.9). In fact, there's a strong analogy between numerical integration methods for the gradient flow and optimization methods for deep learning [28].

2.5 Constraints and regularization

In this section we discuss about constrained optimization problems and regularization. This discussion will be a useful remainder since these approaches are used in the works we decided to compare our approach with.
A pretty common approach in order to avoid overfitting and/or forcing the fitted model to live near particular regions of the hypothesis space Θ is to use regularization. This tecnique consists of adding another term to the loss function, $\lambda\Omega(W)$, in order to penalize regions in the hypothesis space in which $\Omega(W)$ is large. The regularization parameter λ controls how much one wants to penalize these particular regions of the hypothesis space.
Almost always in supervised learning, it happens that the loss function can be regarded as the negative log-likelihood of some conditional probability distribution on the output, $p(y|x,W)$. In this particular cases, if $\Omega(W) > 0\,\forall\, W \in \Theta$, the regularized problem can be interpreted by using a Bayesian lens: we can define an unnormalized probability distribution $p(W) \propto \exp\big(-\lambda\Omega(W)\big)$ and by using Bayes rule we can obtain something proportional to the posterior distribution over the hypothesis space, namely:

$$-\log p(W|X,Y) = -\log p(X,Y|W) - \log p(W) = -\log p(X,Y|W) + \lambda\Omega(W)$$

Oftentimes it is also the case of the so called "improper" priors, whose support is a zero-measure Lebesgue set.

Example 2.5.1 *(Improper prior)*
Convolution is an example of this kind of regularization. In fact, discrete convolution with a fixed kernel is a linear map between finite dimensional vector spaces V and W, but the set of convolutions is a zero measure set in $Hom_{\mathbb{R}}(V,W)$. This can be seen easily by observing that the matrix associated with a convolution has a Toeplitz structure, and in particular the corresponding matrix would be forced to satisfy a linear equation of the kind $M(K)_{tj} - M(K)_{t+1,j+1} = 0$ for some particular j. This already is sufficient to reduce the subspace

$$T = \{M \in Hom_R(V,W) | M \text{ convolution operator}\}$$

onto an hyperplane of $Hom_{\mathbb{R}}(V,W)$ that has zero Lebesgue measure. One can define an improper probability measure on $Hom_{\mathbb{R}}(V,W)$ defined piecewise by:

$$p(M) = \mathbb{1}_T(M)$$

Of course, since T is not compact for the usual topology (it is a non trivial subspace of a finite dimensional vector space) in general, there's no uniform probability distribution so this last function is not normalizable. One can nevertheless avoid normalizability conditions, cause it would lead just to the adding of a constant to the optimization problem. Notice that, a negative log-likelihood applied to $p(M)$ would lead to a function that is evaluated 0 on T and $+\infty$ outside. This forces an hard type constrain on T, so training a fully connected network whose i^{th} layer is convolutional is equivalent to solve the optimization problem:

$$\begin{cases} \arg\min_{W} J(W;X,Y) \\ W^{(i)} \in T \end{cases} \tag{2.10}$$

This idea is the main one behind penalty methods, which we will need to present methods in chapter 4.
An hard constraint can be difficult to handle in practice if the its geometry is not that easy, so often a concentrated prior is used instead.

In the next two examples we will present the most common examples of regularization in deep learning.

Example 2.5.2 *(L^2 regularization)*
*L^2 regularization (also called **Ridge** regularization) is one of the most common types of regularization used in modern deep learning, but its origin lies in statistics, where it is used to handle correlated covariates.*
This approach applied to a weight matrix corresponds to $\Omega(W) = \|W\|_F^2 = \sum_{i,j} W_{ij}^2$.
This regularization shrinks all the entries of the weight matrix W to be near zero. This can be seen also reinterpreting the regularization term $\lambda\Omega(W)$ as a negative log-likelihood: in fact, given the discussion in this section, the prior correspoding to this penalty is $p(W) \propto \exp(-\lambda\|W\|_F^2)$. Using the prior distribution interpretation,

by reparametrizing the penalization coefficient as $\lambda = \frac{1}{2\sigma^2}$ *we can notice that the imposed prior has the functional form of a multivariate isotropic mean zero normal distribution. Thus, this is an intuitive way through which we can see why the weights are shrinked towards zero: the bigger* λ *the smaller the variance of the prior and thus the stronger the prior information we are enforcing in the model.*
The shrinkage property can also be seen with a gradient descent update:

$$W_{ij} \leftarrow W_{ij}(1 - 2\eta\lambda) - \eta \frac{\partial J}{\partial W_{ij}} \tag{2.11}$$

Example 2.5.3 *(L^1 regularization)*
L^1 *regularization (also called* ***Lasso*** *regularization) is the most common regularization used with the purpose of variable selection. It was firstly introduced by Robert Tibshirani in* 1996 *[31] as a method for performing variable selection in Linear regression to obtain more interpretable models.*
This approach applied to a weight matrix corresponds to $\Omega(W) = \|W\|_1 = \sum_{i,j} |W_{ij}|$.
The remarkable property of this regularizer is due to the geometry of the level curves. Suppose we have a starting weight W *that has all positive entries for simplicity (the penalty function is not differentiable in all points in which an entry is zero, but it is a set of measure zero), then it is easy to see that the gradient of* $\Omega(W)$ *does not depend on the magnitude of the entries. This shows that contrary to Ridge regression, in which the shrinkage is proportional to the magnitude, all entries are shrinked towards zero of the same amount, regardless their magnitude.*
From a Bayesian perspective, the prior associated to the Lasso is a Laplace probability distribution $p(W) \propto \exp(-\lambda\|W\|_1)$*, that is sharper around zero than the Ridge's Gaussian.*
Different variations of this regularization has been proposed, from graphical Lasso (Lasso on graphical models), Fused Lasso (exploiting spatial information in the weights), adaptive Lasso to group Lasso. This last one in particular has been used in one of the papers presented in chapter 4. The main idea of group Lasso is to select groups of variables instead of single ones. This is attained by partitioning a weight W *in disjoint sets of indeces* $\{G_k\}_k$*. The penalty used in group Lasso is the following:*

$$\sum_k \sqrt{\sum_{i,j \in G_k} W_{ij}^2} \tag{2.12}$$

This penalty function serves as a "group variable selection", setting to zero entire groups of variables mutually. Intuitively, this is happening because Eq.(2.12) *can be seen as a Lasso on the vector containing the norms of the groups. More details about group Lasso can be found in [33].*

Example 2.5.4 *(L^1/L^2 regularization)*
L^1/L^2 *regularization (also called* ***elastic net****) is obtained as a convex combination of Ridge and Lasso regularizations. This approach was introduced to solve some problems of Lasso. In fact,thinking about the linear model in the high dimensional scenario (number of regressors* p *bigger than number of samples* n*), Lasso selects only* n *variables before it saturates. Moreover, if different regressors are correlated, Lasso tends to select only one and discart the other. These two problems are actually the two main reasons for which Ridge regression was introduced. In fact, in the linear regression case, the quadradic penalty makes the loss strongly convex, ensuring the uniqueness of the local minima. Moreover, the quadratic penalty also allows to tackle the problem of correlated*

regressors.

2.6 Flow in the low-rank manifold

Let's suppose we have a neural network $f(x;W)$ and that we're interested in fitting it with some dataset (X,Y) while keeping the weights in a particular subset $\mathcal{M} \subset \Theta$. For the sake of clarity, we will for the moment think about fully connected neural networks.
Let's suppose moreover that $\mathcal{M}$ has the structure of a differentiable manifold embedded in the parameters space. In particular, the case in which we are interested is the one in which

$$\begin{aligned} &\mathcal{M}_r := \{W \mid rank(W) = r\} \\ &\mathcal{M} = \mathcal{M}_{r_1,..,r_L} = \{(W^{(1)},..,W^{(L)}) \mid W^{(k)} \in \mathcal{M}_{r_k}\, \forall\, k\} \end{aligned} \tag{2.13}$$

As already said, the training phase tries to solve the constrained optimization problem:

$$\begin{cases} \underset{W}{\arg\min} \underset{(x,y)\sim p_{\text{data}}}{\mathbb{E}}\Big[\ell(f(x,W),y)\Big] \\ W \in \mathcal{M} \end{cases} . \tag{2.14}$$

The challenges in trying to solve this problem are multiple:

- This optimization problem is non convex in general, both due to the nature of the neural network or to the complex geometry of the constraint;
- Even assuming convexity of the loss, the manifold constrain poses different problems in trying to build algorithms;

As explained in section 2.4, we would like to have a similar flow formulation for this kind of constrained problem, in order to be able to apply all integration methods to optimize the function.
The first idea could be to pointwisely project the flow on the low-rank manifold by solving at any time step the following optimization problem:

$$W_{LR}(t) = \underset{B\in\mathcal{M}}{\arg\min}\|B - W(t)\| \tag{2.15}$$

This approach has some disadvantages:

1. if the used norm is Frobenius, the singular value decomposition has to be calculated at each epoch to project;
2. this approach is not taking full advantage of the low rank constraint since full weights need to be saved, so there is no computational saving with this approach.

A more natural approach would be to keep the whole flow on the manifold, for the following reasons:

- It is the exact analogue formulation of equation (2.8);

- If one wants to constrain neural network weights to be low rank, it would be nice to be able not to save full weights during training. This would give both benefits in training and inference complexity.

In order to modify the ODE and solve it directly on the manifold, the vector field has to be projected pointwise onto the tangent bundle, leading to

$$\dot{W}_{LR} = -P(W_{LR})\nabla_W J(X, Y; W_{LR}) \tag{2.16}$$

where

$$\begin{aligned} P\colon \mathcal{M} \subset \Theta &\to \coprod_{q \in \mathcal{M}} Hom_{\mathbb{R}}(T_q\Theta, T_q\mathcal{M}) \\ W &\mapsto P(W) \end{aligned}$$

and $P(W)$ is the orthogonal projection operator onto $T_W\mathcal{M}$.
In the next section we will present the derivation of the projection operator in closed form and we will present a numerical method for solving Eq.(2.16), following the presentations in [15, 34].

2.7 Low rank manifold constraint

The main idea in order to solve Eq.(2.16) directly on the manifold is to construct a simpler parametrization of $\mathcal{M}$ and pullback the ODE on the charts. In particular, the parametrization constructed is:

$$\begin{aligned} \phi\colon \mathcal{O}_{m,r}(\mathbb{R}) \times \mathrm{GL}(\mathbb{R}^r) \times \mathcal{O}_{n\times r}(\mathbb{R}) &\to \mathcal{M}_r \\ (U, S, V) \quad &\mapsto \quad \phi(U, S, V) = USV^\top \end{aligned}$$

where $\mathcal{O}_{m,r}(\mathbb{R})$ is the set of $m \times r$ real matrices with orthonormal columns and $GL(\mathbb{R}^r)$ is the space of $r \times r$ invertible matrices. Let's remark that this map is surjective but not injective, since $\phi(UP^\top, PSQ^\top, VQ^\top) = \phi(U, S, V)$ for every $P, Q \in \mathcal{O}_{r,r}(\mathbb{R})$ and thus is not a proper parametrization in the sense of section 2.1.

Supposing to have a time dependent decomposition $W(t) = \phi(U(t), S(t), V(t))$, we can reconstruct the differential equation in terms of U, S and V by differentiating ϕ with respect to time:

$$\dot{W} = \dot{\phi} = \dot{U}SV^\top + U\dot{S}V^\top + US\dot{V}^\top = F(\phi(U, S, V), t) \tag{2.17}$$

where $F(W, t) = -\nabla J(W, t; X, Y)$ is the vector field of the ODE.

Next section is dedicated to a detailed derivation of a system of ODEs on U, S and V starting from the construction of the orthogonal projection operator.

2.7.1 Derivation of the projection operator

Since the tangent space of the manifold is naturally embedded in the tangent space of Θ, in order to obtain a functional form for the projection operator, we can use the minimization condition, i.e.:

$$\Pi(F(W, t)) = \underset{\dot{W} \in T_W\mathcal{M}}{\arg\min} \|\dot{W} - F(W, t)\|_F \tag{2.18}$$

This last condition translates into the fact that the difference between the field and the minimizer of Eq.(2.18) has to be orthogonal to $T_W\mathcal{M}$, more precisely:

$$\langle \dot{W} - F(W,t), \delta W\rangle = 0 \qquad \forall \delta W \in T_W\mathcal{M} \tag{2.19}$$

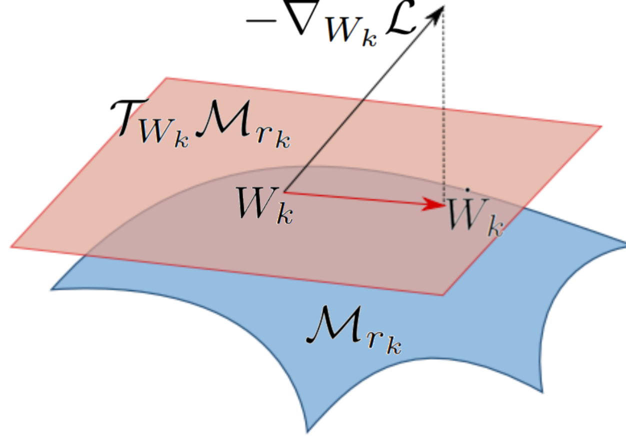

Figure 2.2: Graphical representation of Galerkin condition for projecting the vector field on the tangent space of low-rank matrices. Image taken from [27].

The main effort now is to understand how the elements of the tangent space $T_W\mathcal{M}$ are made, and to do this we need to use a bit of differential geometry.
Remembering that the tangent space of a product manifold is isomorphic to the product of the tangent spaces, let's consider the pushforward of ϕ to the tangent spaces in the point (U, S, V) :

$$\begin{aligned} d\phi(U,S,V)\colon T_U\mathcal{O}_{m,r}(\mathbb{R})\times T_S\mathrm{GL}(\mathbb{R}^r) \times T_V\mathcal{O}_{n\times r}(\mathbb{R}) &\to T_{\phi(U,S,V)}\mathcal{M}_r \\ (\delta U, \delta S, \delta V) \qquad &\longmapsto \qquad d\phi(U,S,V)(\delta U, \delta S, \delta V) \end{aligned}$$

In particular, since $GL(\mathbb{R}^r)$ is an open set, its tangent space at every point is isomorphic to $\mathbb{R}^{r\times r}$. An explicit description of $T_U\mathcal{O}_{m,r}(\mathbb{R})$ is instead given by remembering that the definition of $\mathcal{O}_{m,r}(\mathbb{R})$ is the counterimage of zero through the map $G(A) = A^\top A - I$ (that is smooth). The tangent space is defined as:

$$T_U\mathcal{O}_{m,r}(\mathbb{R}) = \{\dot{\gamma}(0)\,|\,\gamma \in \mathcal{C}^1((-1,1);\mathcal{O}_{m,r}(\mathbb{R})),\ \gamma(0) = U\} \tag{2.20}$$

Now, since $\mathcal{O}_{m,r}(\mathbb{R}) = G^{-1}(0)$, it is know (check example 2.1.2) that $T_U\mathcal{O}_{m,r}(\mathbb{R}) \cong Ker(dG(U))$, thus :

$$\begin{aligned} T_U\mathcal{O}_{m,r}(\mathbb{R}) =& \{\dot{\gamma} \in \mathbb{R}^{m\times r}\,|\,dG(U)(\dot{\gamma}) = U^\top\dot{\gamma} + \dot{\gamma}^\top U = 0\} = \\ =& \{\dot{\gamma} \in \mathbb{R}^{m\times r}\,|\,U^\top\dot{\gamma} \in SO(r)\} \end{aligned} \tag{2.21}$$

where $SO(r)$ is the space of $r \times r$ skew-symmetric matrices.

To obtain a closed form for $d\phi(U,S,V)(\delta U,\delta S,\delta V)$ we need to calculate the directional derivative of ϕ in the direction of the tangent vector:

$$\begin{aligned} d\phi(U,S,V)(\delta U,\delta S,\delta V) &= \frac{\partial\phi(U,S,V)}{\partial(\delta U,\delta S,\delta V)} = \\ &= \lim_{\epsilon\to 0}\frac{\phi(U+\epsilon\delta U, S+\epsilon\delta S, V+\epsilon\delta V)-\phi(U,S,V)}{\epsilon} = \\ &= \lim_{\epsilon\to 0}\frac{(U+\epsilon\delta U)(S+\epsilon\delta S)(V+\epsilon\delta V)^\top - USV^\top}{\epsilon} = \\ &= \lim_{\epsilon\to 0}\frac{\epsilon(\delta USV^\top + U\delta SV^\top + US(\delta V)^\top) + o_0(\epsilon)}{\epsilon} = \\ &= \delta USV^\top + U\delta SV^\top + US\delta V^\top \end{aligned} \tag{2.22}$$

In order to guarantee unicity of representation for tangent vectors of $T_{\phi(U,S,V)}\mathcal{M}$ we have to impose some additional conditions. As suggested in [15], we consider the extended tangent map:

$$\begin{aligned} \widetilde{d\phi}(U,S,V)\colon T_U\mathcal{O}_{m,r}(\mathbb{R})\times T_S\mathrm{GL}(\mathbb{R}^r)\times T_V\mathcal{O}_{n\times r}(\mathbb{R}) &\to T_{\phi(U,S,V)}\mathcal{M}_r\times SO(r)\times SO(r) \\ \Big(\delta U,\delta S,\delta V\Big) &\longmapsto \Big(d\phi|_{(U,S,V)}(\delta U,\delta S,\delta V), U^\top\delta U, V^\top\delta V\Big) \end{aligned}$$

This last linear map has a trivial kernel.It can be seen by using the fact that $U^\top\delta U = 0 = V^\top\delta V$:

$$\begin{aligned} d\phi|_{(U,S,V)}(\delta U,\delta S,\delta V) = 0 \implies & U^\top\delta USV^\top V + U^\top U\delta SV^\top V + U^\top US(\delta V)^\top V = \\ &= \delta S = 0 \end{aligned} \tag{2.23}$$

By using this last condition, we obtain that :

$$\begin{aligned} \delta USV^\top + US(\delta V)^\top = 0 &\implies U^\top\delta USV^\top + U^\top US(\delta V)^\top = S(\delta V)^\top = 0 \\ &\underset{S\in GL(R^r)}{\implies} \delta V = 0 \end{aligned} \tag{2.24}$$

In the same way we can obtain (multiplying by V on the right) that $\delta U = 0$, hence the extended tangent map has a trivial kernel. Moreover, by a dimensionality count we can observe that the dimension of the domain is :

$$\begin{aligned} & dim_{\mathbb{R}}\Big(T_U\mathcal{O}_{m,r}(\mathbb{R})\times T_S\mathrm{GL}(\mathbb{R}^r)\times T_V\mathcal{O}_{n\times r}(\mathbb{R})\Big) = dim_{\mathbb{R}}\Big(T_U\mathcal{O}_{m,r}(\mathbb{R})\Big) + \\ & + dim_{\mathbb{R}}\Big(T_S\mathrm{GL}(\mathbb{R}^r)\Big) + dim_{\mathbb{R}}\Big(T_V\mathcal{O}_{n\times r}(\mathbb{R})\Big) = (mr - \frac{r(r+1)}{2}) + r^2 + \\ & + (nr - \frac{r(r+1)}{2}) = r(m+n-1) \end{aligned} \tag{2.25}$$

In the same way the dimension of the codomain is :

$$\begin{aligned} &dim_{\mathbb{R}}\Big(T_{\phi(U,S,V)}\mathcal{M}_r \times SO(r) \times SO(r)\Big) = dim_{\mathbb{R}}\Big(T_{\phi(U,S,V)}\mathcal{M}_r\Big) + \\ &+ 2\, dim_{\mathbb{R}}\Big(SO(r)\Big) = (m+n-r)r + 2\frac{r(r-1)}{2} = r(m+n-1) \end{aligned} \tag{2.26}$$

Since domain and codomain have the same dimension, the extended linear map $\widetilde{d\phi}$ is an isomorphism of vector spaces. This implies in particular, that any matrix in $T_W\mathcal{M}$ is image of a particular $(\delta U, \delta S, \delta V)$ through $d\phi(U, S, V)$ and moreover that the representation is unique if we impose the Gauge conditions:

$$\begin{aligned} U^\top \delta U = 0 \\ V^\top \delta V = 0 \end{aligned} \tag{2.27}$$

In fact, from the relation $\delta W = \delta U S V^\top + U \delta S V^\top + U S \delta V^\top$, by multiplying on the left with $U^\top$, on the right by V and using Eq.(2.27) we arrive to:

$$\delta S = U^\top \delta W V \tag{2.28}$$

In a similar way, multiplying on the right by V and using Eq.(2.28) allows us to solve for δU explicitly, leading to :

$$\delta U = (I - UU^\top)\delta W V S^{-1} \tag{2.29}$$

Multiplying on the left by $U^\top$ and using Eq.(2.28) yields:

$$\delta V = (I - VV^\top)(\delta W)^\top U S^{-T} \tag{2.30}$$

By observing equations (2.29),(2.30) we notice that $UU^\top$ and $VV^\top$ are respectively the ortogonal projections onto the space spanned by the columns of U and V. So $(I - UU^\top) = (I - P_U) = P_U^\perp$ is the orthogonal projection onto $im(U)^\perp$ and the same holds for V.

Going back to Eq.(2.19) after this discussion, we can rewrite $\dot{W}$ using the chain rule and imposing orthogonality with all tangent vectors, which will lead to a system of ODEs for the factors U, S and V. In particular, it is sufficient to choose a basis of the tangent space.

Respectively, for $\delta W = U\delta S V^\top$, $\delta W = \delta U S V^\top$, $\delta W = U S \delta V^\top$, imposing orthogonality conditions for all tangent vectors (Eq.(2.19)), we obtain:

$$\begin{cases} \langle \dot{U}SV^\top + U\dot{S}V^\top + US\dot{V}^\top - F(W,t), U\delta S V^\top \rangle = 0 \\ \langle \dot{U}SV^\top + U\dot{S}V^\top + US\dot{V}^\top - F(W,t), \delta U S V^\top \rangle = 0 \\ \langle \dot{U}SV^\top + U\dot{S}V^\top + US\dot{V}^\top - F(W,t), US\delta V^\top \rangle = 0 \end{cases} \tag{2.31}$$

Using properties of Frobenius inner product (reported in section 0.1), we can rewrite Eq.(2.31) as :

$$\begin{cases} \langle U^\top \dot{U}SV^\top V + U^\top U\dot{S}V^\top V + U^\top US\dot{V}^\top V - U^\top F(W,t)V, \delta S \rangle = 0 \\ \langle \dot{U}SV^\top VS^\top + U\dot{S}V^\top VS^\top + US\dot{V}^\top VS^\top - F(W,t)VS^\top, \delta U \rangle = 0 \\ \langle S^\top U^\top \dot{U}SV^\top + S^\top U^\top U\dot{S}V^\top + S^\top U^\top US\dot{V}^\top - U^\top F(W,t), \delta V^\top \rangle = 0 \end{cases} \tag{2.32}$$

Now, using Gauge conditions (2.27) on Eq.(2.32) and the orthogonality of U and V we obtain:

$$\begin{cases} \langle \dot{S} - U^\top F(W,t)V, \delta S\rangle = 0 \\ \langle \dot{U}SS^\top + U\dot{S}S^\top - F(W,t)VS^\top, \delta U\rangle = 0 \\ \langle S^\top \dot{S}V^\top + S^\top S\dot{V}^\top - S^\top U^\top F(W,t), \delta V^\top\rangle = 0 \end{cases} \tag{2.33}$$

Since first equation of (2.33) has to hold for all δS, we obtain the first differential equation for S. Moreover, the third equation can be rewritten again by transposing both matrices in the inner product. Finally, by inserting the first equation of (2.33) in the other two, we obtain:

$$\begin{cases} \dot{S} = U^\top F(W,t)V \\ \langle \dot{U}SS^\top + UU^\top F(W,t)VS^\top - F(W,t)VS^\top, \delta U\rangle = 0 \\ \langle VV^\top F(W,t)^\top US + \dot{V}S^\top S - F(W,t)^\top US, \delta V\rangle = 0 \end{cases} \tag{2.34}$$

By the Gauge conditions (2.27), the range of δU and δV are orthogonal to the ranges of U and V respectively, thus it is possible to rewrite $\delta U = P_U^\perp \alpha$, $\delta V = P_V^\perp \beta$ for some matrices α,β. Using this reparametrization in Eq.(2.34) and using symmetry of the projection operator we get:

$$\begin{cases} \dot{S} = U^\top F(W,t)V \\ \langle P_U^\perp \dot{U}SS^\top - P_U^\perp F(W,t)VS^\top, \alpha\rangle = 0 \\ \langle P_V^\perp \dot{V}S^\top S - P_V^\perp F(W,t)^\top US, \beta\rangle = 0 \end{cases} \tag{2.35}$$

Recalling Eq.(2.35) has to hold for all α, β, and since $P_U^\perp \dot{U} = \dot{U}$ (same analogue for V), we finally get:

$$\begin{cases} \dot{S} = U^\top F(W,t)V \\ \dot{U} = P_U^\perp F(W,t)VS^{-1} \\ \dot{V} = P_V^\perp F(W,t)^\top US^{-\top} \end{cases} \tag{2.36}$$

Equations (2.36) represent the flow of the starting ODE on the manifold of rank r matrices, in terms of the factors U, S and V.
Using these three last equations we can also obtain a closed form for the projection onto the tangent space at a particular point $W = USV^\top$:

$$\begin{cases} \dot{W} = \dot{U}SV^\top + U\dot{S}V^\top + US\dot{V}^\top \underset{[(2.36)]}{=} FP_V + P_U F - P_U F P_V = P(W)F(W,t) \\ \text{with } P(W)M = M - P_U^\perp M P_V^\perp \end{cases} \tag{2.37}$$

The system of ODEs (2.36) can be solved numerically throught the use of any numerical integration tecnique. It is important to notice that in the context of neural networks, a numerical approximation of Eq.(2.36) **never requires the reconstruction of the full weight matrices**.
It is also important to notice that a straightforward numerical integration of Eq. (2.36) can lead to problems due to the inversion of S, that can be almost singular in the case of an overestimation of the rank. A suited

numerical integrator for this kind of problem was first proposed in [23], the projector-splitting integrator.

Due to its importance, we report here a result regarding the regularity of the projection operator proven in [15]. This results clarifies that the stiffness of Eq.(2.36) does not depend on the decomposition, it is intrinsic of the problem:

Lemma 2.7.1 (Lemma 4.2 [15]) *Let $W_1 \in \mathcal{M}_r$ be such that its smallest nonzero singular value $\sigma_r(W) \geq \rho > 0$ and let $W_2 \in \mathcal{M}_r$ with $\|W_1 - W_2\| \leq \frac{\rho}{8}$. Then, for all $B \in \mathbb{R}^{n\times m}$ the following bounds hold:*

$$\begin{aligned} &\|\big(P(W_2) - P(W_1)\big)B\| \leq 8\rho^{-1}\|W_2 - W_1\|\|B\|_2 \\ &\|P^{\perp}(W_2)(W_2 - W_1)\| \leq 4\rho^{-1}\|W_2 - W_1\|^2 \end{aligned} \tag{2.38}$$

In particular, remembering that $\|B\|_2 \leq \|B\|$, the first inequality of (2.38) gives us a bound on Lipschitz constant of P (locally on $\mathcal{M}_r \cap B(W_1, \frac{\rho}{8})$) for the Frobenius norm. In particular, it holds:

$$\|P\|_{\text{Lip}} \leq 8\rho^{-1} \tag{2.39}$$

This observation is clarifying also the fact that if the matrix $\sigma_r(W_1) \approx 0$, then the local Lipschitz constant of the projection can potentially be really big, and thus requiring a smaller time integration step.

2.7.2 Projector-splitting integrator

As mentioned at the end of last section, a straightforward integration of the system of ODEs derived in (2.36) may be problematic in the case in which the rank is overestimated, and thus the inversion of S may cause issues. The main idea behind the projector-splitting integrator is to use Lie-Trotter splitting on (2.37). The authors' proposal consists in using the following representation of the projection operator

$$P(W)M = MP_V - P_U MP_V + P_U M \tag{2.40}$$

and then sequentially performing a numerical integration of three differential equations:

1. As a first step, integrate between t_0 and t_1 the ODE

$$\begin{cases} \dot{W}_I = F(W,t)P_V \\ W_I(t_0) = W(t_0) \end{cases} \tag{2.41}$$

2. Then, integrate between t_0 and t_1

$$\begin{cases} \dot{W}_{II} = -P_U F(W,t)P_V \\ W_{II}(t_0) = W_I(t_1) \end{cases} \tag{2.42}$$

3. Lastly, integrate between t_0 and t_1

$$\begin{cases} \dot{W}_{III} = P_U F(W,t) \\ W_{III}(t_0) = W_{II}(t_1) \end{cases} \tag{2.43}$$

and use $W_{III}(t_1)$ as an approximation for $W(t_1)$.

In all of the three steps any numerical integrator can be used. It is important to notice also that the vector fields on the right hand side of all three differential equations are tangent to the manifold. In fact, by using Eq.(2.40) and the fact that projection are idempotent we get:

$$\begin{aligned} P(W)F(W,t)P_V &= F(W,t)P_V \\ P(W)P_U F(W,t)P_V &= P_U F(W,t)P_V \\ P(W)P_U F(W,t) &= P_U F(W,t) \end{aligned} \tag{2.44}$$

This ensures that the flow will stay in the correct manifold if the integration is exact. Remarkably, all of the three steps can be solved exactly in terms of the factorization(Lemma 3.1, [23]). This last lemma basically rewrites Eq.(2.37) as a system of differential equations in terms of the factorization $W(t) = U(t)S(t)V(t)^\top$, that can be numerically solved with any kind of integrator.

Algorithm 2.1 Projector-splitting integrator [23]

Input : Initial svd factorization $W(t_0) = U(t_0)S(t_0)V(t_0)^\top$ with $S(t_0) \sim r \times r$; $U(t_0) \sim n \times r$; $V(t_0) \sim n \times r$;
Define $K(t) = U(t)S(t)$, $L(t) = V(t)S(t)^\top$ and let $\eta = t_1 - t_0$

1 Integrate between t_0 and t_1 the differential equation:
$\dot{K}(t) = F(K(t)V(t)^\top, t)V(t)$ with $K(t_0) = U(t_0)S(t_0)$; `/* K-step */`
2 Factorize $K(t_1) = U_1\widehat{S}_1$ using either QR or SVD;

3 Integrate between t_0 and t_1 the differential equation:
$\dot{\widetilde{S}}(t) = -U_1^\top F(K(t)V(t)^\top, t)V(t)$ with $S(t_0) = \widehat{S}_1$; `/* S-step */`
4

5 Integrate between t_0 and t_1 the differential equation:
$\dot{L}(t) = F(K(t)V(t)^\top, t)^\top U_1$ with $L(t_0) = V(t_0)\widetilde{S}_0$; `/* L-step */`
6 Factorize $L(t_1) = V_1 S_1$ using either QR or SVD;

7 $W_1 = U_1 S_1 V_1^\top$ now approximates $W(t_1)$.

We point out that in Alg.2.1 as in all the next algorithms, we implicitly suppose that in any SVD the singular values are ordered in decreasing order.

2.7.3 Unconventional robust integrator

An alternative improvement of [23] is proposed in [5]. More than solving the instability problem given by the presence of S^{-1} in the system of ODEs, this approach is able to be partially parallelized and it avoids numerical integrations backward in time (that can be instable and it is not suitable for a training algorithms

since it changes the direction of the gradient flow).
As in algorithm 2.1, the notation $K := US$ and $L := VS^\top$ is used. Starting from Eq. (2.36) with a bit of calculation using Leibnitz rule, we can arrive to:

$$\begin{cases} \dot{K} = (\dot{US}) = F(KV^\top, t)V \\ \dot{L} = (\dot{VS^\top}) = F(UL^\top, t)^\top U \\ \dot{S} = U^\top F(USV^\top, t)V \end{cases} . \tag{2.45}$$

Application of the unconventional integrator to this system of differential equations leads to update rule presented in algorithm 2.2.

Algorithm 2.2 Unconventional robust integrator [5]

Input : Initial svd factorization $W(t_0) = U(t_0)S(t_0)V(t_0)^\top$ with $S(t_0) \sim r \times r$; $U(t_0) \sim n \times r$; $V(t_0) \sim n \times r$;
Define $K(t) = U(t)S(t)$, $L(t) = V(t)S(t)^\top$

8 Integrate from t_0 to t_1 the ode $\dot{K} = F(K(t)V(t_0)^\top, t)V(t_0)$, with $K(t_0) = U(t_0)S(t_0)$ `/* K-step */`
9 Factorize $K(t_1) = U_1 R_1$ using either QR or SVD and define $M := U_1^\top U_0$;

10 Integrate from t_0 to t_1 the ode $\dot{L} = F(UL(t)^\top, t)^\top U(t_0)$, with $L(t_0) = V(t_0)S(t_0)^\top$ `/* L-step */`
11 Factorize $L(t_1) = V_1 \widetilde{R}_1$ using either QR or SVD and define $N := V_1^\top V_0$;

12 Integrate from t_0 to t_1 the ode $\dot{S} = U_1^\top F(U_1 S(t) V_1^\top, t)^\top V_1$, with $S(t_0) = MS(t_0)N^\top$ `/* S-step */`
13 Set $S_1 := S(t_1)$;

14 $W_1 = U_1 S_1 V_1^\top$ now approximates $W(t_1)$.

In particular, in all the three steps of Alg. 2.2 any numerical integrator can be used. In the case of neural networks, we will use mostly forward Euler.
A remarkable property is that this integrator mantains the exactness properties of Alg.2.1, as shown in theorems 3,4 of [5]. Moreover, an advantage of this integrator with respect to the Projector-Splitting is that it exploits the ruled structure of the low-rank manifold, moving along flat subspaces during the K and L integration steps.
Despite the effectiveness of this integrator, it is often not easy in applications to choose the rank by hand. This is exceedingly more important in our application case, since choosing a good rank for each layer of a deep neural network can be prohibitive. Fortunately, a rank-adaptive version of the unconvetional integrator had been developed in [18].

2.7.4 Rank-adaptive unconventional integrator

Apart from stability problems solved by approaches algorithms 2.1,2.2, it is not easy in general to choose the rank of the state of the system a priori. A solution to this problem had been proposed in [18]. In particular, the latter determines the rank adaptively at discrete times during the numerical integration of the ODE.

The main idea behind the modification of the unconventional integrator provided in [18] is to integrate a basis augmented version of the differential equation for S and then perform a rank-adaption step.
The rank-adaptive system of ODEs can be sinthetically resumed by the following:

$$\begin{cases} \dot{K} = (\dot{US}) = F(USV^\top, t)V \\ \dot{L} = (V\dot{S}^\top) = F(USV^\top, t)^\top U \\ \dot{\widehat{S}} = \widehat{U}^\top F(\widehat{U}\widehat{S}\widehat{V}^\top, t)\widehat{V} \end{cases}, \tag{2.46}$$

where the hat is put on quantities that have been modified by the basis augmentation, so they make reference to a doubled rank, as explained in the original work. A detailed algorithmic description of this method is presented in algorithm 2.3.

Algorithm 2.3 rank-adaptive unconventional integrator [18]

Input : Initial svd factorization $W(t_0) = U(t_0)S(t_0)V(t_0)^\top$ with $S(t_0) \sim r \times r$; $U(t_0) \sim n \times r$; $V(t_0) \sim n \times r$;
Define $K(t) = U(t)S(t)$, $L(t) = V(t)S(t)^\top$;
rank tolerance ϑ; Quantities with the hat make reference to rank $2r$

15 Integrate from t_0 to t_1 the ode $\dot{K} = F(K(t)V(t_0)^\top, t)V(t_0)$, with $K(t_0) = U(t_0)S(t_0)$ /* K-step */

16 Factorize $(K(t_1), U_0) = \widehat{U}R \sim n \times 2r$ using either QR or SVD and define $\widehat{M} := \widehat{U}^\top U_0 \sim 2r \times r$;

17 Integrate from t_0 to t_1 the ode $\dot{L} = F(UL(t)^\top, t)^\top U(t_0)$, with $L(t_0) = V(t_0)S(t_0)^\top$ /* L-step */

18 Factorize $(L(t_1), V_0) = \widehat{V}\widetilde{R} \sim m \times 2r$ using either QR or SVD and define $\widehat{N} := \widehat{V}^\top V_0 \sim 2r \times r$;

19 Integrate from t_0 to t_1 the ode $\dot{\widehat{S}} = \widehat{U}^\top F(\widehat{U}\widehat{S}(t)\widehat{V}^\top, t)^\top \widehat{V}$, with $\widehat{S}(t_0) = \widehat{M}S(t_0)\widehat{N}^\top$ /* S-step */

20

21 Compute the SVD $\widehat{S}(t_1) = \widehat{P}\widehat{\Sigma}\widehat{Q}^\top$; /* truncation step */

22 Choose the minimal $r_1 \leq 2r$ such that:

$$\Big(\sum_{j=r_1}^{2r} \Sigma_{jj}^2\Big)^{\frac{1}{2}} \leq \vartheta \tag{2.47}$$

Let $S_1 := \sum_{j=1}^{r_1} \Sigma_{j,j} e_j e_j^\top$ and let P_1,Q_1 be the matrices containing the first r_1 columns of $\widehat{P}$, $\widehat{Q}$ respectively;
Set $U_1 := \widehat{U}P_1$ and $V_1 := \widehat{V}Q_1$

23 $W_1 = U_1 S_1 V_1^\top$ now approximates $W(t_1)$.

This version of the integrator is the most suitable for neural network training, since the ranks are chosen automatically and there is only one hyperparameter that controls the amount of cutting.
In the next section we will formalize the application of these integrators in the context of neural network training.

3

The lottery ticket hypothesis and dynamical low-rank training

During my thesis' work I implemented a version of the rank adaptive robust integrator in the context of neural networks. The work produced is publicly available [27] along with the repository of the implementation at https://github.com/compile-gssi-lab/DLRT/tree/efficient_gradient.

3.0.1 Pruning and lottery tickets

Model compression had been a very active area of research in the last several years. These techniques range from pruning, weight quantization, layer factorizations and low-rank compression.
Especially in the recent years, the development of pruning tecniques has been leading this area. The main rationale is that the model under consideration is probably overfitting "by construction", and thus it is safe to "prune" nodes or connections in the neural network without loosing much performance. In order to prune in a methodic way, different tecniques have been proposed. Some of them use measures of importance on the nodes (both of zero and first order) [3], other use randomization [22], other approaches focus on structured pruning [21, 36, 39]. The resulting neural network is often of much smaller size, allowing to reduce both memory consumption and inference time complexity.
Despite the popularity and the effectiveness of these approaches, there are still issues they leave unsolved:

1. They're sometimes based on heuristics;
2. Training has often to be done on the full architecture before pruning, not reducing training costs;
3. Even if the pruning is done during train time, graphical processing units are not able to exploit arbitrarily sparse structures efficiently [25];
4. Fine tuning is often required after pruning;

5. In general, even if the pruned neural network is able to produce comparable metrics with the full version, there is no guarantee it will be able to be trained from scratch and reach the same performance.

An attempt to find a solution to the last problem was first proposed in [9], introducing the "Lottery ticket hypothesis": fully-connected, randomly initialized networks contain subnetworks that when trained in isolation are able to achieve approximately the same performance of the original model. The method proposed in [9] aims to find winning lottery tickets through iterative pruning during the train phase and reinitialization to a masked version of starting weights. Despite its effectiveness to find winning tickets, this approach leaves some of the above-cited problems unsolved. In particular, the cost reduction during training is not the main concern of this algorithm. Moreover, arbitrarily sparse structures cannot be exploited efficiently by graphical processing units.
Other recently explored possibilities are low-rank compression strategies, in which the main idea is to factorize layers' weights into an SVD like decomposition as in [11, 13, 38]. Also these kind of approaches raise different problems:

1. In some cases they require the reconstruction of the full weight matrices during training (like [11, 21, 36, 39]), leading to no space complexity improving during that phase;
2. some other methods (like [38, 39]) are pruning singular values after a first training, leading no space complexity reduction during training;
3. Some approaches require the use of alternating optimization or fine tuning in order to reach the full network performance (like [11]).
4. In layer decomposition approaches like [13] there may be stability problems concerning the optimization;

In DLRT [27] we propose a method able to tackle some of the issues presented for both pruning and low-rank compression methods. In particular, our proposal is able to reduce space and time complexity both during training and inference phases.
The next sections are organized as follows: first, the method is described formally with all the mathematical details. After that, an experiment section follows. Lastly, implementation details with a cost analysis is presented.

3.0.2 Overview of DLRT method [27]

As already introduced in the last chapter, the starting idea is to reinterpret the optimization of a neural network as a gradient flow problem. In particular, it can be interpreted as a system of matrix (or tensor in the case of convolution) ordinary differential equations:

$$\dot{W}_{ij}^{(\alpha)} = -\frac{\partial J}{\partial W_{ij}^{(\alpha)}} \tag{3.1}$$

We decided to apply [18] in a layerwise manner. More precisely, we grouped the system of ODEs layerwise and used algorithm 2.3 on each of them. This boils down to projecting each layer's vector field onto the tangent space of the low-rank manifold, leading to the following ODE:

$$\dot{W}^{(\alpha)} = -P_{r^{(\alpha)}}(W^{\alpha})\nabla_{W^{(\alpha)}}J, \quad \forall\alpha = 1,\dots,L \tag{3.2}$$

where $P_{r^{(\alpha)}}(W^{\alpha})$ is the orthogonal projection operator onto $T_{W^{(\alpha)}}\mathcal{M}_{r_\alpha}$ (and the index α indicates the layer of the neural network).

Each of these matrix ODEs is exactly in the scenario in which algorithm 2.3 can be applied. By using the adaptive version of the integrator we can reparametrize each layer α using the variables $K^{(\alpha)}$, $L^{(\alpha)}$ and $\widehat{S}^{(\alpha)}$ to obtain from Eq.(3.2) the following system of ODEs:

$$\begin{cases} \dot{K}^{(\alpha)} = -\nabla_{W^{(\alpha)}}J(\{K^{(\beta)}V^{(\beta)\top}\}_{\beta=1,..,L})V^{(\alpha)} \\ \dot{L}^{(\alpha)} = -\nabla_{W^{(\alpha)}}J(\{U^{(\beta)}L^{(\beta)\top}\}_{\beta=1,..,L})^{T}U^{(\alpha)} \\ \dot{\widehat{S}}^{(\alpha)} = -\widehat{U}^{(\alpha)\top}\nabla_{W^{(\alpha)}}J(\{\widehat{U}^{(\beta)}\widehat{S}^{(\beta)}\widehat{V}^{(\beta)\top}\}_{\beta=1,..,L})\widehat{V}^{(\alpha)} \end{cases} . \tag{3.3}$$

This first group of differential equation can be already exploited as a first implementation. From now on, we will refer to it as "**efficient forward**". The reason for this nomenclature is that the implementation based on equation 3.3 requires only one forward and one backpropagation for every optimization step, as opposed to the one we will present next.

It is important to remark that the right hand side of Eq.(3.3) are the gradients with respect to K, L and $\widehat{S}$, in fact:

$$\begin{aligned} \frac{\partial J}{\partial K_{ij}^{(\alpha)}} &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial W_{km}^{(\beta)}}{\partial K_{ij}^{(\alpha)}} = \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial}{\partial K_{ij}^{(\alpha)}}(K_{kl}^{(\beta)}V_{ml}^{(\beta)}) \\ &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\delta_{ik}\delta_{jl}\delta_{\alpha\beta}V_{ml}^{(\beta)} = \frac{\partial J}{\partial W_{im}^{(\alpha)}}V_{mj}^{(\alpha)} = \left[\nabla_W J V^{(\alpha)}\right]_{ij} \end{aligned} \tag{3.4}$$

where we used Einstein summation convention. Similarly, we obtain:

$$\begin{aligned} \frac{\partial J}{\partial L_{ij}^{(\alpha)}} &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial W_{km}^{(\beta)}}{\partial L_{ij}^{(\alpha)}} = \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial}{\partial L_{ij}^{(\alpha)}}(U_{kl}^{(\beta)}L_{ml}^{(\beta)}) \\ &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\delta_{im}\delta_{jl}\delta_{\alpha\beta}U_{kl}^{(\beta)} = \frac{\partial J}{\partial W_{ki}^{(\alpha)}}U_{kj}^{(\alpha)} = \left[\nabla_W J^{\top} U^{(\alpha)}\right]_{ij} \end{aligned} \tag{3.5}$$

Lastly, for $\widehat{S}$ we have:

$$\begin{aligned} \frac{\partial J}{\partial \widehat{S}_{ij}^{(\alpha)}} &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial W_{km}^{(\beta)}}{\partial \widehat{S}_{ij}^{(\alpha)}} = \frac{\partial J}{\partial W_{km}^{(\beta)}}\frac{\partial}{\partial \widehat{S}_{ij}^{(\alpha)}}(\widehat{U}_{kl}^{(\beta)}\widehat{S}_{lt}^{(\beta)}\widehat{V}_{mt}^{(\beta)}) \\ &= \frac{\partial J}{\partial W_{km}^{(\beta)}}\delta_{il}\delta_{jt}\delta_{\alpha\beta}\widehat{U}_{kl}^{(\beta)}\widehat{V}_{mt}^{(\beta)} = \frac{\partial J}{\partial W_{km}^{(\alpha)}}\widehat{U}_{ki}^{(\alpha)}\widehat{V}_{mj}^{(\alpha)} = \left[\widehat{U}^{(\alpha)\top}\nabla_W J\widehat{V}^{(\alpha)}\right]_{ij} \end{aligned} \tag{3.6}$$

Knowing this, Eq.(3.3) can be rewritten as:

$$
\begin{cases}
\dot{K}^{(\alpha)} = -\nabla_{K^{(\alpha)}} J(\{K^{(\beta)} V^{(\beta)\top}\}_{\beta=1,..,L}) \\
\dot{L}^{(\alpha)} = -\nabla_{L^{(\alpha)}} J(\{U^{(\beta)} L^{(\beta)\top}\}_{\beta=1,..,L}) \\
\dot{\widehat{S}}^{(\alpha)} = -\nabla_{\widehat{S}^{(\alpha)}} J(\{\widehat{U}^{(\beta)} \widehat{S}^{(\beta)} \widehat{V}^{(\beta)\top}\}_{\beta=1,..,L})
\end{cases} . \tag{3.7}
$$

This last observation can be exploited in order to tape also the gradients in an efficient way, so the implementation that will follow from Eq.(3.7) will be called "**efficient gradient**". The reason of the name is that this second approach is also space efficient in handling the gradients, since they are also represented using a low-rank structure.

More details about this efficient taping are presented in the next section. A careful cost analysis of the two different versions is presented in section 5.1.

3.1 Efficient gradient taping

We can exploit equations (3.4),(3.5),(3.6) in the implementation in order to avoid the calculation of the full gradient with respect to the weights. We exploited Pytorch's automatic differentiation to tape the gradients with respect to K,L and $\widehat{S}$ without never constructing the full gradients with respect to W (visual representation in Fig.3.1,3.2). This potential computationally saving approach comes together with the need of three forward propagations at each optimization step (and one backpropagation, as shown in Eq.(6.2)). Suppose we have a neural network

$$
\begin{cases}
f(x; W) = z^{(L)}(x; W) \\
x^{(\alpha+1)} = T^{(\alpha+1)}(z^{(\alpha)}; W^{(\alpha+1)}) \\
z^{(\alpha)} = \sigma^{(\alpha)}\left(x^{(\alpha)}\right) \\
z^{(0)} = x, z^{(L)} = y
\end{cases} \tag{3.8}
$$

where $\sigma^{(\alpha)}$ are nonlinear entrywise activation functions and $T^{(\alpha)}$ are parametric (with matrix parameters) linear operators acting on the embeddings $z^{(\alpha)}$. It is worth to notice that both fully connected and convolutional neural networks are included in this representation. Suppose moreover we have a dataset (X, Y) along with a loss function $J = J(W; X, Y)$.

Suppose again we are using the same parametrization as in Eq.(2.46). We can efficiently tape the gradients needed to integrate the system (3.3) by calculating the gradients with respect to $K_{ij}^{(\alpha)}$, $L_{ij}^{(\alpha)}$ and $S_{ij}^{(\alpha)}$. In particular:

$$
\frac{\partial J}{\partial K_{ij}^{(\alpha)}} = \frac{\partial J}{\partial z^{(\alpha)}} \frac{\partial z^{(\alpha)}}{\partial x^{(\alpha)}} \frac{\partial x^{(\alpha)}}{\partial K_{ij}^{(\alpha)}} \tag{3.9}
$$

In order for this phase to be efficient we exploited the backpropagation algorithm already implemented in Pytorch. In this way we were able to recover the partial gradients $\frac{\partial J}{\partial x^{(\alpha)}}$ and so to compute $\frac{\partial J}{\partial K_{ij}^{(\alpha)}}$ efficiently. The only distinction that has to be done is depending on the functional form of the linear operator $T^{(\alpha)}$.

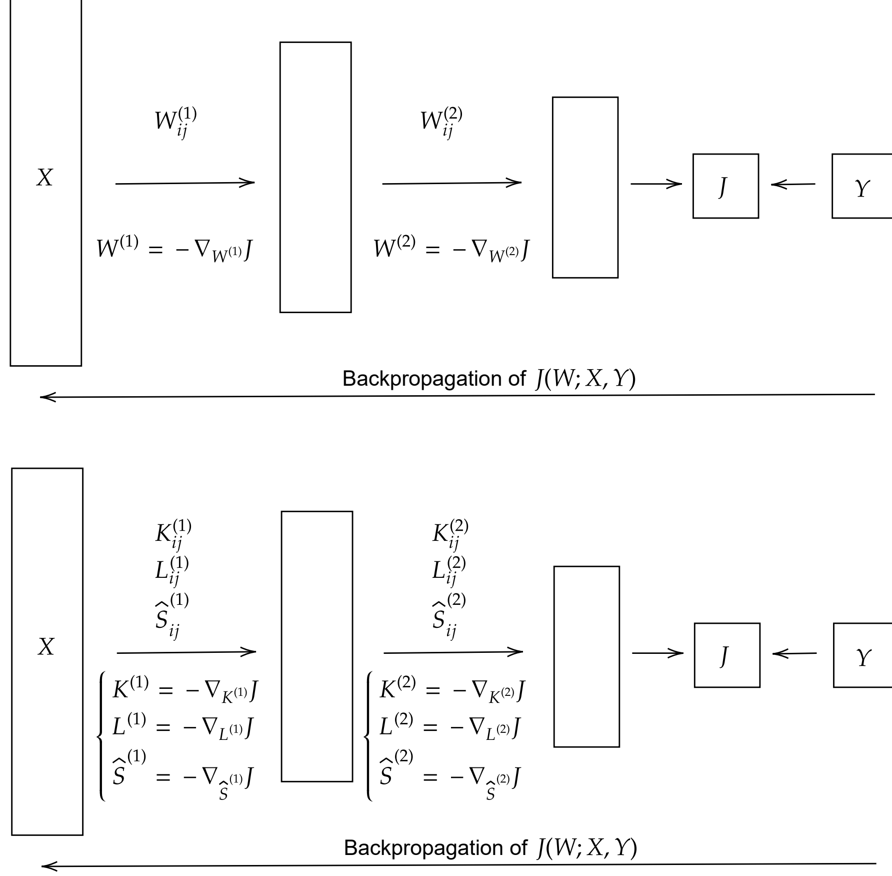

Figure 3.1: Full and low-rank representation of neural networks with the associated gradient flow problems.

In the next section we will present the main kind of linear transformations used in neural networks.

Fully-connected layers

This is the most general case, and any other affine transformation can be (with some constraints) reinterpreted in this setting. In particular, this is the general case in which:

$$x^{(\alpha)} = T^{(\alpha)}(z^{(\alpha-1)}; W^{(\alpha)}) = W^{(\alpha)} z^{(\alpha-1)} + b^{(\alpha)} \tag{3.10}$$

For the three different forward phases, this affine transformation is parametrized as follows:

$$\begin{aligned} x^{(\alpha)} &= K^{(\alpha)} V^{(\alpha)\top} z^{(\alpha-1)} + b^{(\alpha)} && \textbf{K-step} \\ x^{(\alpha)} &= U^{(\alpha)} L^{(\alpha)\top} z^{(\alpha-1)} + b^{(\alpha)} && \textbf{L-step}\,. \\ x^{(\alpha)} &= \widehat{U}^{(\alpha)} \widehat{S}^{(\alpha)} \widehat{V}^{(\alpha)\top} z^{(\alpha-1)} + b^{(\alpha)} && \textbf{S-step} \end{aligned} \tag{3.11}$$

With the respective partial derivatives for the chain rule:

$$
\begin{aligned}
&\frac{\partial x_\ell^{(\alpha)}}{\partial K_{ij}^{(\alpha)}} = \delta_{i\ell}\Big[V^{(\alpha)\top} z^{(\alpha-1)}\Big]_j && \textbf{K-step} \\
&\frac{\partial x_\ell^{(\alpha)}}{\partial L_{ij}^{(\alpha)}} = U_{\ell j}^{(\alpha)} z_i^{(\alpha-1)} && \textbf{L-step} \\
&\frac{\partial x_\ell^{(\alpha)}}{\partial \widehat{S}_{ij}^{(\alpha)}} = \widehat{U}_{\ell i}^{(\alpha)}\Big[\widehat{V}^{(\alpha)\top} z^{(\alpha-1)}\Big]_j && \textbf{S-step}
\end{aligned}
\tag{3.12}
$$

2D Convolution layers

Despite the fact that convolution with a fixed kernel can be interpreted as a linear transformation, its matrix representation would be much bigger in size, and moreover it would be constrained to be a Toeplitz matrix. Since this matrix structure is not preserved by the unconventional integrators, other solutions need to be used.
A simple solution is to flatten the convolution kernel on some dimensions and to use a low-rank decomposition of the obtained matrix. This is the approach we used in all the experiments.

A convolution kernel can be represented as a four-mode tensor $W \in \mathbb{R}^{F\times C\times J\times K}$ consisting 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,

$$
(Z * W)(n, f, u, v) = \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} W(f, c, j, k) Z(n, c, u-j, v-k)\,. \tag{3.13}
$$

As already mentioned, the goal is to flatten the kernel in a meaningful way and treat it as it were a fully connected layer. The reshape of W comes with shapes $W^{\text{resh}} \in \mathbb{R}^{F\times CJK}$. This reshaping is also considered in [11].

With this kind of flattening, the convolution can be seen as the contraction between an three-mode tensor Z^{unf} of patches and the reshaped kernel matrix W^{resh} (Pytorch's fold-unfold function was used in the implementation). We constructed the unfold by stacking the vectorized version of sliding patterns of the kernel on the original input, obtaining in this way a tensor $Z^{\text{unf}} \in \mathbb{R}^{N\times CJK\times L}$, where L denotes the dimension of flatten version of the output of the 2-D convolution. With this notation, equation (3.13) can be rewritten as a tensor mode product:

$$
\begin{aligned}
x^{(\alpha)} =& (Z * W^{(\alpha)})(n, f, u, v) = \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} W^{(\alpha)\text{resh}}(f, (c, j, k)) Z^{\text{unf}}(n, (c, j, k), (u, v)) \\
=& \sum_{p=1}^{r} U^{(\alpha)}(f, p) \sum_{q=1}^{r} S^{(\alpha)}(p, q) \sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{c=1}^{C} V^{(\alpha)}((c, j, k), q) Z^{\text{unf}}(n, (c, j, k), (u, v))
\end{aligned}
\tag{3.14}
$$

Last step is to rewrite the three different step for this last operation. Rewriting Eq.(3.14) using the three parametrizations, we get:

$$
\begin{aligned}
x^{(\alpha)} &= \sum_{q=1}^{r_\alpha} K^{(\alpha)}(f,q) \sum_{j,k,c} V^{(\alpha)}((c,j,k),q) Z^{\text{unf}}(n,(c,j,k),(u,v)) && \textbf{K-step} \\
x^{(\alpha)} &= \sum_{p=1}^{r_\alpha} U^{(\alpha)}(f,q) \sum_{j,k,c} L^{(\alpha)}((c,j,k),p) Z^{\text{unf}}(n,(c,j,k),(u,v)) && \textbf{L-step} \\
x^{(\alpha)} &= \sum_{p=1}^{2r_\alpha} \widehat{U}^{(\alpha)}(f,q) \sum_{q=1}^{2r_\alpha} \widehat{S}^{(\alpha)}(p,q) \sum_{j,k,c} \widehat{V}^{(\alpha)}((c,j,k),p) Z^{\text{unf}}(n,(c,j,k),(u,v)) && \textbf{S-step}
\end{aligned} \tag{3.15}
$$

or more compactly, using tensor mode products (definition in the notation section 0.1):

$$
\begin{aligned}
x^{(\alpha)} &= [Z^{\text{unf}} \times_2 V^{(\alpha)\top}] \times_2 K^{(\alpha)} && \textbf{K-step} \\
x^{(\alpha)} &= [Z^{\text{unf}} \times_2 L^{(\alpha)\top}] \times_2 U^{(\alpha)} && \textbf{L-step} \\
x^{(\alpha)} &= \Big[[Z^{\text{unf}} \times_2 \widehat{V}^{(\alpha)\top}] \times_2 \widehat{S}^{(\alpha)}\Big] \times_2 \widehat{U}^{(\alpha)} && \textbf{S-step}
\end{aligned} \tag{3.16}
$$

As pointed out in the section, in order to be able to backpropagate through this layer we need to compute the derivatives of each one of the steps in Eq. (3.15) with respect to their parameters. This calculation gives:

$$
\begin{aligned}
\frac{\partial x^{(\alpha)}(n,f,u,v)}{\partial K^{(\alpha)}(\mu,\nu)} &= \delta(f,\mu)\Big[Z^{\text{unf}} \times_2 V^{(\alpha)\top}\Big](n,\nu,(u,v)) && \textbf{K-step} \\
\frac{\partial x^{(\alpha)}(n,f,u,v)}{\partial L^{(\alpha)}(\mu,\nu)} &= U^{(\alpha)}(f,\nu) Z^{\text{unf}}(n,\mu,(u,v)) && \textbf{L-step} \\
\frac{\partial x^{(\alpha)}(n,f,u,v)}{\partial \widehat{S}^{(\alpha)}(\mu,\nu)} &= \widehat{U}^{(\alpha)}(f,\mu)\Big[Z^{\text{unf}} \times_2 V^{(\alpha)\top}\Big](n,\nu,(u,v)) && \textbf{S-step}
\end{aligned} \tag{3.17}
$$

3.2 Training procedure description

In this section we present in detail the adaptation of the unconventional integration for the training of neural networks. A detailed description of the algorithm is contained in Alg.3.1.
As already mentioned in this chapter, it is possible to train a neural network using dynamical low-rank in different ways. It is in fact possible to train some layers using the rank-adaptive unconventional integrator, others with the fixed rank unconventional integrator and others without any kind of compression in their full rank representation (using any desired optimizer). In particular the algorithmic description 3.1 is nothing more than iterating through the layers and for each one either use one between rank-adaptive,full rank unconventional integrators or (if the layer is not to be compressed) a standard optimizer. It is worth to notice that the numerical integration steps in Alg.3.1 are left as generic as possible, since in theory any numerical integrator can be used. Although this pecularity opens to a lot of possibilities coming from numerical analysis, the choice has to be restricted to methods leading to computationally feasible trainings. In particular, in all the experiment of chapter 5 we used forward Euler integration.

Algorithm 3.1 Dynamic Low Rank Training Scheme (DLRT) [27]

Input : Initial low-rank factors $S_0^{(\alpha)} \sim r_0^{(\alpha)} \times r_0^{(\alpha)}; U_0^{(\alpha)} \sim n_\alpha \times r_0^{(\alpha)}; V_0^{(\alpha)} \sim n_{\alpha-1} \times r_0^{(\alpha)}$ for $\alpha = 1, \ldots, L$;
`iter`: maximal number of descent iterations per epoch;
`adaptive`: Boolean flag that decides whether or not to dynamically update the ranks;
$\mathcal{C}$: list of layers to be compressed;
ϑ: absolute singular value threshold for adaptive procedure;
τ: relative singular value threshold for adaptive procedure (either this or ϑ is required)

24 **for** *each* `epoch` **do**
25 **for** $t = 0$ *to* $t =$ `iter` **do**
26 **for** *each layer* α **do**
27 **if** $\alpha \in \mathcal{C}$ **then**
28 $K_t^{(\alpha)} \leftarrow U_t^{(\alpha)} S_t^{(\alpha)}$ `/* K-step */`
29 $K_{t+1}^{(\alpha)} \leftarrow \{\dot{K}(t) = -\nabla_K \mathcal{L}(K(t)(V^{(\alpha)}(t))^\top z^{(\alpha-1)} + b^{(\alpha)}(t)), K(0) = K_t^{(\alpha)}\}$
30 $L_t^{(\alpha)} \leftarrow V_t^{(\alpha)} (S_t^{(\alpha)})^\top$ `/* L-step */`
31 $L_{t+1}^{(\alpha)} \leftarrow$ one-step-integrate$\{\dot{L}(t) = -\nabla_L \mathcal{L}(U^{(\alpha)}(t) L(t)^\top z^{(\alpha-1)} + b^{(\alpha)}(t)), L(0) = L_t^{(\alpha)}\}$
32 **if** `adaptive` **then** `/* Basis augmentation step */`
33 $K_{t+1}^{(\alpha)} \leftarrow [K_{t+1}^{(\alpha)} \mid U_t^{(\alpha)}]$ $L_{t+1}^{(\alpha)} \leftarrow [L_{t+1}^{(\alpha)} \mid V_t^{(\alpha)}]$
34 $U_{t+1}^{(\alpha)} \leftarrow$ orthonormal basis for the range of $K_{t+1}^{(\alpha)}$ `/* S-step */`
35 $M^{(\alpha)} \leftarrow (U_{t+1}^{(\alpha)})^\top U_t^{(\alpha)}$
$V_{t+1}^{(\alpha)} \leftarrow$ orthonormal basis for the range of $L_{t+1}^{(\alpha)}$
$N^{(\alpha)} \leftarrow (V_{t+1}^{(\alpha)})^\top V_t^{(\alpha)}$
$\widetilde{S}_t^{(\alpha)} \leftarrow M^{(\alpha)} S_t^{(\alpha)} N^{(\alpha)\top}$
$S_{t+1}^{(\alpha)} \leftarrow \{\dot{S}(t) = -\nabla_S \mathcal{L}(U_{t+1}^{(\alpha)} S(t) (V_{t+1}^{(\alpha)})^\top z^{(\alpha-1)} + b^{(\alpha)}(t)), S(0) = \widetilde{S}_t^{(\alpha)}\}$
36 **if** `adaptive` **then** `/* Rank compression step */`
37 $P, \Sigma, Q \leftarrow \mathrm{SVD}(S_{t+1}^{(\alpha)})$
if `absolute threshold` **then**
38 $r_{t+1}^{(\alpha)} \leftarrow \min\{0 \leq r \leq 2r_t^{(\alpha)} \mid \sum_{s=r+1}^{2r_t^{(\alpha)}} \Sigma_{jj}^2 \leq \vartheta^2\}$
39 **else if** `relative threshold` **then**
40 $r_{t+1}^{(\alpha)} \leftarrow \min\{0 \leq r \leq 2r_{(\alpha)}(t) \mid \sum_{s=r+1}^{2r_t^{(\alpha)}} \Sigma_{jj}^2 \leq \tau^2 \sum_{s=1}^{2r_t^{(\alpha)}} \Sigma_{jj}^2\}$
41 $S_{t+1}^{(\alpha)} \leftarrow$ truncate Σ using the singular value threshold ϑ
$U_{t+1}^{(\alpha)} \leftarrow U_{t+1}^{(\alpha)} \widetilde{P}$ where $\widetilde{P} =$ [first $r_{t+1}^{(\alpha)}$ columns of P]
$V_{t+1}^{(\alpha)} \leftarrow V_{t+1}^{(\alpha)} \widetilde{Q}$ where $\widetilde{Q} =$ [first $r_{t+1}^{(\alpha)}$ columns of Q]
42 **else if** $\alpha \notin \mathcal{C}$ **then**
43 $W_{t+1}^{(\alpha)} \leftarrow \{\dot{W}(t) = -\nabla_W \mathcal{L}(W_t^{(\alpha)} z^{(\alpha-1)} + b(t)), W(0) = W_t^{(\alpha)}\}$
`/* Bias update step */`
44 $b_{t+1}^{(\alpha)} \leftarrow \{\dot{b}(t) = -\nabla_b \mathcal{L}(U_{t+1}^{(\alpha)} S_{t+1}^{(\alpha)} (V_{t+1}^{(\alpha)})^\top z^{(\alpha-1)} + b(t)), b(0) = b_t^{(\alpha)}\}$

3.3 Low-rank lottery tickets

Once the training procedure described in Alg.3.1 is completed, by using the U, S, V representation for each layer it is possible to fix U and V and fine-tune only S using any optimizer.
Fixing U and V for each layer can be interpreted as the pruning analogue of fixing the sparse structure and train on the remaining weights. As in the original paper [9], we would expect an improvement of performance by analogy. Computationally, this procedure allows to reduce even more the memory complexity needed in DLRT, since only the gradients of S needs to be accumulated (and they have dimension $r \times r$, where r is the rank at the end of DLRT training). Moreover, fine-tuning requires only one forward and not three anymore. Although this approach was not used in all experiments, its effectiveness is shown in the experiment in section 5.6.

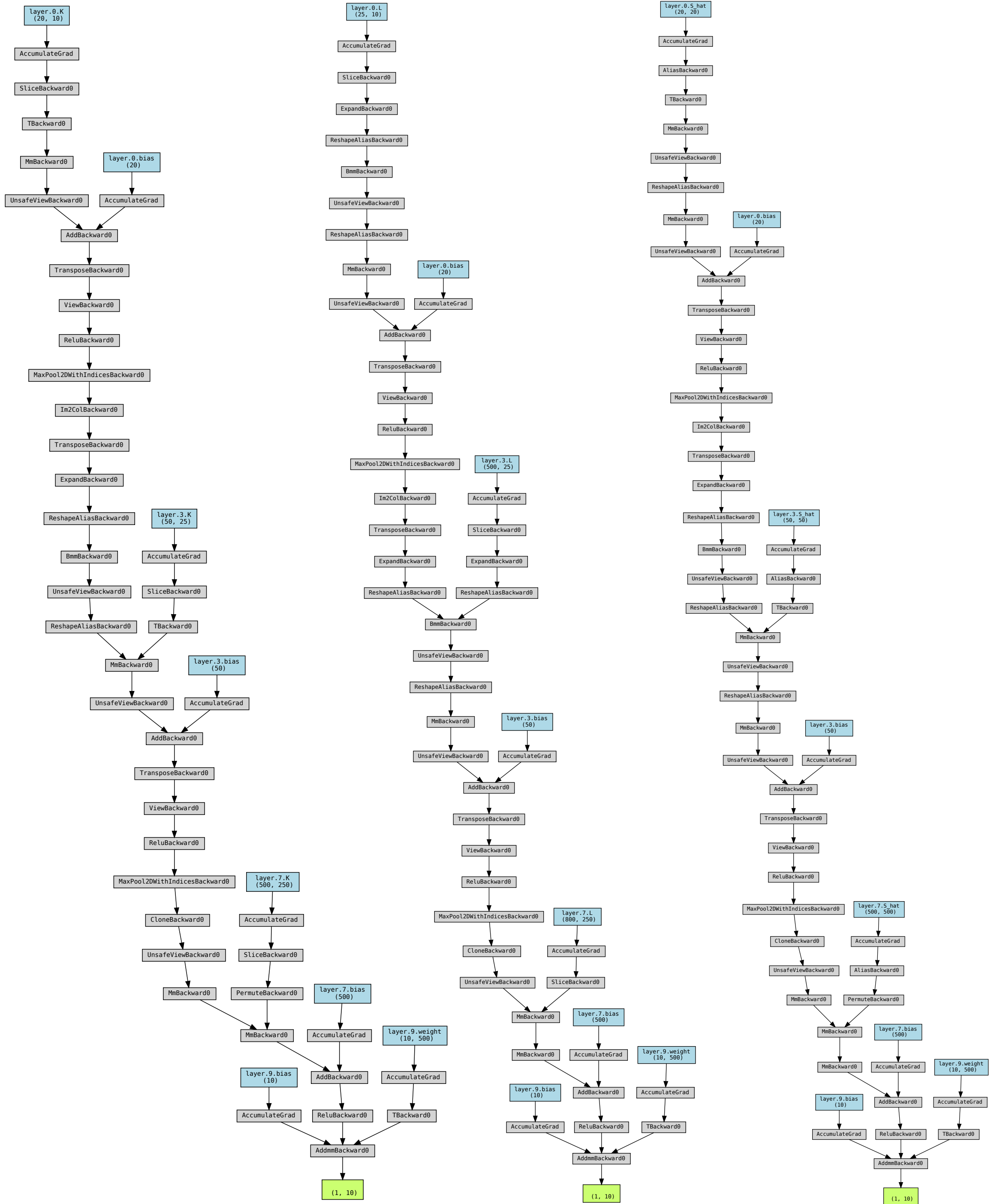

Figure 3.2: Lenet5 representations of the K,L and S steps for the gradient efficient implementation of Alg.3.1.

4
Comparison with other methods

In this section we give a synthetic overview on the other methods proposed in the literature with which we are comparing DLRT with in the experiments in chapter 5.

4.1 Structured sparse learning

Structured sparse learning is the compression technique presented in [36]. As the name suggests, the proposal of this paper is focusing on a structured sparsification of convolutional kernels using group Lasso (example 2.5.3) regularization. Group Lasso penalization can be applied to convolutional kernels in different ways, depending how the variables inside the tensor are grouped. In [36], three main groupings are presented. Following the notation used in section 3.1, the three grouping penalties are summarized as follows:

- Channelwise and filterwise grouping: in this first one the penalty is applied to the convolution kernel both filterwise and channelwise. More precisely, the regularization terms added to the total loss are:

$$\lambda_f \sum_{\alpha=1}^{L} \sum_{f_\alpha=1}^{F_\alpha} \|W^{(\alpha)}_{f_\alpha,:,:,:}\|_F + \lambda_c \sum_{\alpha=1}^{L} \sum_{c_\alpha=1}^{C_\alpha} \|W^{(\alpha)}_{:,c_\alpha,:,:}\|_F \tag{4.1}$$

  where the first term refers to the filterwise pruning, and the second to the channelwise one. With this grouping, entire filters or channels of the convolution kernel are set to zero by the Lasso.

- Shapewise: this second approach prunes local "stripes" of the convolutional filter. The regularization is:

$$\lambda_s \sum_{\alpha=1}^{L} \sum_{f_\alpha=1}^{C_\alpha} \sum_{j_\alpha=1}^{J_\alpha} \sum_{k_\alpha=1}^{K_\alpha} \|W^{(\alpha)}_{:,c_\alpha,j_\alpha,k_\alpha}\|_F \tag{4.2}$$

- Depthwise: this last approach selects and prunes entire kernels in the neural network architecture.

The penalty is given by:

$$\lambda_d \sum_{\alpha=1}^{L} \|W^{(\alpha)}\|_F \tag{4.3}$$

4.2 Rethinking smaller-norm less-informative assumption

This second approach is presented in [39].
The main point of the author's work is to question the common assumption that a smaller magnitude of a weight means a less informative feature. This assumption originally came as an induction from simpler models' observations, like linear regression. The authors also point out some problems in using penalization on deep architectures to choose the important predictors (e.g., Lasso or Ridge), mainly attributable to the need of feature normalization to not bias the regularization.
The proposal in this work is to enforce sparsity on the scale parameter of batch normalization layers by using a variation of iterative shrinkage thresholding proposed in [2].
Following notation from section 3.1, suppose we have a batch normalization following the convolutional layer number α. Suppose Z is the input of the convolution, a tensor of shapes $N \times C \times J \times K$, where N is the batch size, C is the number of channels and $J \times K$ are the spatial dimensions. Batch normalization after this convolutional layer is defined as:

$$BN(W^{(\alpha)} * Z)_{ncjk} = \gamma_c^{(\alpha)} \frac{(W^{(\alpha)} * Z)_{ncjk} - \mathbb{E}[(W^{(\alpha)} * Z)_{n,:,:,:}]}{\sqrt{var((W^{(\alpha)} * Z)_{n,:,:,:}) + \epsilon}} + \beta_c^{(\alpha)} \tag{4.4}$$

The action of a batch normalization layer is to rescale the statistic of equation (4.4) in order to be optimal for the next layer activation, by controlling mean and variance of the batch through the learnable parameters γ and β. By imposing a sparse structure on each $\gamma^{(\alpha)}$, some output channels are forced to be constantly equal to the corresponding entry of $\beta^{(\alpha)}$ (Eq.(4.4) with $\gamma_c^{(\alpha)} = 0$ is constantly equal to $\beta_c^{(\alpha)}$). This observation is important, since in the case of no padding the output of a batch normalization layer will have some constant channels, so of no practical usage. The authors numerically showed that it is safe to prune the corresponding channels in the convolution kernel with approximately no functional changes from the original network.
This approach allows versatility, allowing to choose among different kinds of structured pruning, having also the advantage of using a relatively easy regularization. However, as discussed in section 3.0.1, this method does not reduce training costs.

4.3 Pruning via GAL

This approach is proposed in [21].
The proposal of the authors is to use generative adversarial learning to identify which structures in the neural network can be pruned. This approach considers three neural networks: a baseline model $f_B(x; W_B)$, a generator $f_G(x; m, W_G)$ and a discriminator $f_D(y; W_D)$. Following the notation of the original paper, W

are the neural network's weights are m is a soft mask after each structure (as explained in [21], different kinds of masking can be adopted,e.g, for entire blocks of the network, channels,...). The baseline model is a non-pruned version of the model we want to compress, already trained on the dataset of interest. The discriminator acts on the last layer embedding features of the generator and the baseline, with the objective of recognizing if it comes from one or the other.
This approach boils down to the minimization of a loss function composed of different terms, namely:

$$\begin{aligned}
&J(W_D, W_G, m) = \max_{W_D} J_{\text{adv}}(W_G, m, W_D) + J_{\text{data}}(W_G, m) + J_{\text{reg}}(W_G, m, W_D) \\
&J_{\text{adv}}(W_G, m, W_D) = \mathop{\mathbb{E}}_{f_B(x)\sim p_B(x)}\Big[\log(f_D(f_B(x); W_D))\Big] + \mathop{\mathbb{E}}_{x,z}\Big[\log(1 - f_D(f_G(x, z; W_G); W_D))\Big] \\
&J_{\text{data}}(W_G, m) = \frac{1}{N}\sum_{i=1}^{N}\|f_G(x; W_G) - f_B(x)\|_2^2 \\
&J_{\text{reg}}(W_G, m, W_D) = \lambda_G\|W_G\|_F^2 + \lambda_D\|m\|_1 + \lambda_D R(W_D)
\end{aligned} \tag{4.5}$$

where $p_B(x)$ is the output distribution of the baseline network for an input distributed as p_{data}. The second expectation is taken with respect to x, z, with $z \sim p_z(z)$ (distribution of the noisy input of the generator) and $f_G(x, z) \sim p_G(x, z)$. For more details about generative adversarial learning, we refer to [10].
The first term of Eq.(4.5) is the adversarial loss, that is trying to enforce the generator's output to produce outputs whose distribution is similar to the one of the baseline and at the same time is trying to enforce the discriminator to recognize from which neural network the output is coming from (generator or baseline). Intuitively, this loss is trying to enforce the masked generator to "mimic" the behaviour of the baseline network. The second term is a regression loss (typically mean squared error), encouraging the masked generator outputs to be as near as possible to the ones of the baseline for the same input. The third part is an additive regularization term (the sum of three functions, each one depending on only one group of variables), for which they proposed an L^2 regularization on the weights of the generator, a L^1 (cause it enforces sparsity) on the mask m and another loss on the weights of the discriminator to prevent it from dominating the training. In particular, L^1 regularization on the mask m is the one that is truly sparsifying the network.
Despite the versatility of this approach, some of the issues presented in section 3.0.1 are present. In particular, this method can be described as a sort of compressed knowledge distillation, since it requires an already trained non-compressed model. Thus, the compressed neural network training is not meant to give space advantages during the training phase, but it is only useful for a compression a posteriori. The versatility of masking m allows to sparsify the underlying network in many ways, but it also introduces a choice to be done a priori before training.

4.4 Low-rank compression of neural networks

This approach is proposed in [11].
In this work, the authors propose a method to include the automatic choice of the ranks for each layer in the

optimization problem. In particular, the optimization problem they proposed to solve is:

$$\underset{r_\alpha, rank(W^{(\alpha)})\leq R_\alpha}{\arg\min} \quad J(W^{(1)},\dots,W^{(L)}) + \lambda C(r_1,\dots,r_L) \tag{4.6}$$

Given the combinatorial nature of this optimization problem, they propose a functional form for the regularization C and an optimization algorithm that makes it computationally approachable. The constraint on the rank can be handled with a layer decomposition $W^{(\alpha)} = U^{(\alpha)}V^{(\alpha)\top}$ with $U^{(\alpha)} \in \mathbb{R}^{n_{\alpha+1}\times r_\alpha}$ and $V^{(\alpha)} \in \mathbb{R}^{n_\alpha \times r_\alpha}$. Moreover, they assume that C is a separable function of the ranks, more precisely a positive linear combination of them:

$$\begin{aligned} C(r_1,\dots,r_L) &= C(r_1) + \cdots + C(r_L) \\ C(r_\alpha) &= (n_\alpha + n_{\alpha+1})r_\alpha \end{aligned} \tag{4.7}$$

The regularization they proposed (Eq.(4.7)) can be interpreted as a "memory regularization", since $C(r_\alpha)$ represented the number of weights that has to be saved for layer α. This functional form for C allows the optimization problem to be numerically solved with linear cost in the number of layers (instead of exponential), and moreover it is trying to enforce a lower rank on the layers that are heavier from a memory complexity point of view.
As the authors suggest, problem (4.6) can be rewritten using auxiliary variables $\Theta^{(\alpha)}$ as follows:

$$\begin{cases} \underset{r_\alpha, W^{(\alpha)}, \Theta^{(\alpha)}}{\arg\min} \quad J(W^{(1)},\dots,W^{(L)}) + \lambda C(r_1,\dots,r_L) \\ rank(\Theta^{(\alpha)}) \leq R_\alpha \\ W^{(\alpha)} = \Theta^{(\alpha)} \end{cases} \tag{4.8}$$

This reformulation takes the form of "model compression as constrained optimization", presented in [4]. In the latter, the authors suggest to combine a penalty method with alternating optimization. The penalty leads to the following loss function:

$$\begin{cases} J(W,r,\Theta) = J(W) + \lambda C(r) + \frac{\mu}{2}\sum_{j=1}^{L}\|W^{(\alpha)} - \Theta^{(\alpha)}\|_F^2 \\ rank(W^{(\alpha)}) \leq R_\alpha \end{cases} \tag{4.9}$$

The alternating optimization approach divides the problem in two steps: the learning and the compression step. In the learning step, the objective is to minimize the expression in Eq.(4.9) as a function of only the weights $W^{(\alpha)}$ with all the other variables fixed, leading to:

$$\underset{W^{(\alpha)}}{\arg\min}\, J(W) + \frac{\mu}{2}\sum_{j=1}^{L}\|W^{(\alpha)} - \Theta^{(\alpha)}\|_F^2 \tag{4.10}$$

Moreover, thanks to the separability of C, the compression step can be rewritten layerwise as follows:

$$\begin{cases} \arg\min_{r_\alpha, \Theta^{(\alpha)}} \lambda C(r_\alpha) + \frac{\mu}{2} \|W^{(\alpha)} - \Theta^{(\alpha)}\|_F^2 \\ rank(\Theta^{(\alpha)}) \leq R_\alpha \end{cases} \tag{4.11}$$

Since the norm used in the compression step is the Frobenius one, Eq.(4.11) can be interpreted as a standard penalized low-rank approximation problem, that can be solved exactly through a singular value decomposition of $\Theta^{(\alpha)}$, using Eckhart-Young theorem.
This approach again does not solve one of the problems presented in section 3.0.1: it does not reduce training memory costs.

4.5 Singular Vector Orthogonality Regularization and Singular Value Sparsification

This approach is proposed in [38].
Contrary to all other works we presented, in this paper the main focus is to introduce a low-rank training algorithm that is able to be performed directly on the factors, without the need of the full-rank representation of the weights. The authors proposal consists on two main points: first, the neural network with layer decomposition is trained with full-rank, then a singular value pruning on the resulting network is performed (together with a fine tuning, if necessary). To maintain the SVD-like structure during the training procedure, the authors propose a regularization approach to preserve the orthogonality constraint on the columns of the factors U and V. The training boils to down to the following loss:

$$\begin{aligned} J(\{U^\alpha, V^{(\alpha)}, S^{(\alpha)}\}_\alpha) = & J_{\text{data}}(U^{(\alpha)}\sqrt{S^{(\alpha)}}, V^\alpha \sqrt{S^{(\alpha)\top}}) + \\ & + \lambda_o \sum_{\alpha=1}^{L} \frac{1}{r_\alpha} \Big(\|U^{(\alpha)\top} U^{(\alpha)}\|_F^2 + \|V^{(\alpha)\top} V^{(\alpha)} + \|_F^2 \Big) + \lambda_s \sum_{\alpha=1}^{L} \ell_s(S^{(\alpha)}) \end{aligned} \tag{4.12}$$

where the first term is the data loss written in terms of the layer factorization, the second one is to enforce orthogonality on the columns of the factors U and V of each layer, and the last one is meant to enforce sparsity in the singular value diagonal matrix.
The authors argue about the choice of the sparsity enforcing loss, since L^1 regularization has the property of not being invariant to a scale transformation (since it is a norm), thus every singular value is shrinked towards zero in the same manner, regardless of its magnitude (Example 2.5.3). This property makes this kind of regularization less appealing to be used together with pruning, since the final magnitude of the singular values after training cannot be used as a robust indicator of their importance. The authors propose to substitute L^1 regularization with Hoyer's regularization, that is scale invariant. This regularization has the following expression:

$$\ell_s(S^{(\alpha)}) = \frac{\|diag(S^{(\alpha)})\|_1}{\|diag(S^{(\alpha)})\|_2} = \frac{\sum_i |S_{ii}^{(\alpha)}|}{\sqrt{\sum_i S_{ii}^{(\alpha)2}}} \tag{4.13}$$

Hoyer's regularization Eq.(4.13) allows to maintain all the optimization friendly properties of L^1 norm while having the advantage of being scale invariant. This allows to maintain the majority of the importance in the singular values with the bigger magnitude, and so to prune more safely the lowest magnitude singular values. After the first training, a pruning of the singular values is performed by using an energy threshold. A fine tuning can be also performed if the performance drops significantly.

5
Cost analysis and experiments

5.1 Cost analysis

In this section, a theoretical cost analysis is presented. It is important to remark the fact that even though all forward matrix multiplications are associative, the cost is not. In particular, by being careful with the implementation, it is possible to reduce inference cost.
The analysis is organized as follows: first, we present space and time complexity of the efficient gradient taping procedure. This discussion can be compared with the efficient forward phase implementation.

5.1.1 Efficient gradient taping

The advantage of this first implementation is its ability to save space even for the gradients during the train phase, leading to a potentially more appealing algorithm for memory limited devices. This advantage comes together with the disadvantage of having to perform three forwards instead of one before backpropagating.
In the following calculations, we will assume that we have a fully connected architecture with L layers and that the dimension of layer α is n_α, with rank r_α. Moreover, we assume that the input has shape $n_1 \times b$, where b is the batch size. We will not consider bias and nonlinear activation costs (in any case the cost of these operations is the same of the one in full forward).

Space complexity during training

In this section we present a space complexity analysis of [27]. As described in section 3.2, dynamical low-rank training requires the construction of additional matrices during the training phase.
Layer α has stored inside: $K^{(\alpha)} \sim n_{\alpha+1} \times r_\alpha$ (gradient required), $L^{(\alpha)} \sim n_\alpha \times r_\alpha$ (gradient required), $\widehat{U}^{(\alpha)} \sim n_{\alpha+1} \times 2r_\alpha, \widehat{V}^{(\alpha)} \sim n_\alpha \times 2r_\alpha, \widehat{S}^{(\alpha)} \sim 2r_\alpha \times 2r_\alpha$ (diagonal,gradient required), $\widehat{M}^{(\alpha)} \sim 2r_\alpha \times r_\alpha$ and $\widehat{N}^{(\alpha)} \sim 2r_\alpha \times r_\alpha$. Thus, the overall memory complexity of one step in the training phase of our

approach compared with the standard weight representation (without counting the gradients) is:

$$\begin{aligned} C_{\text{memory, no grad}} &= \sum_{\alpha=1}^{L} O\Big(r_\alpha(3n_{\alpha+1} + 3n_\alpha + 4r_\alpha + 2)\Big) \\ C_{\text{memory full, no grad}} &= \sum_{\alpha=1}^{L} O\Big(n_\alpha n_{\alpha+1}\Big) \end{aligned} \tag{5.1}$$

Taking also the space consumption of the gradients, the last comparison becomes:

$$\begin{aligned} C_{\text{memory, with grad}} &= \sum_{\alpha=1}^{L} O\Big(4r_\alpha(n_{\alpha+1} + n_\alpha + r_\alpha + 1)\Big) \\ C_{\text{memory full,with grad}} &= \sum_{\alpha=1}^{L} O\Big(2n_\alpha n_{\alpha+1}\Big) \end{aligned} \tag{5.2}$$

As a measure of compression to compare the two approaches, we decided to use the compression ratio. It is defined as the fraction of parameters DLRT is able to save compared to the vanilla approach. More formally, it is defined as:

$$CR = 1 - \frac{C_{\text{memory}}}{C_{\text{memory full}}} \tag{5.3}$$

This measure can be used both taking and not taking into account the gradients. In particular, since gradients occupy a non-negligible space in the memory during training, we can compute the compression ratio taking this into account. Moreover, to be able to find a closed form upper bound for the maximal rank in order for our approach to be advantageous, we consider compression ratio for just one layer:

$$CR_{\alpha,\text{ with grads}} = 1 - \frac{4r_\alpha(n_{\alpha+1} + n_\alpha + r_\alpha + 1)}{2n_\alpha n_{\alpha+1}} \tag{5.4}$$

Having a particular network, one can calculate the maximum needed rank of a layer to achieve a compression ratio bigger than a fixed threshold by solving $CR_{\alpha,\text{ with grads}} \geq \beta$ for r_α. In particular, a theoretical space gain layerwise counting also the gradients is obtained for all r_α that satisfy the inequality $CR_{\alpha,\text{ with grads}} \geq 0$, and this happens if:

$$0 \leq r_\alpha \leq \frac{\sqrt{n_\alpha^2 + (1 + n_{\alpha+1})^2 + 2n_\alpha(2n_{\alpha+1} + 1)} - (n_\alpha + n_{\alpha+1} + 1)}{2} \leq \frac{\sqrt{2n_\alpha n_{\alpha+1}}}{2} \tag{5.5}$$

Time complexity during training

(K-forward) In this first forward, each low-rank layer is represented using $K^{(\alpha)}$ and $V^{(\alpha)}$. For a single layer, the costs are $O(bn_\alpha r_\alpha)$ for the multiplication between $z^{(\alpha)}$ and $V^{(\alpha)\top}$. This new matrix has shape $r_\alpha \times b$, so its multiplication with $K^{(\alpha)}$ has cost $O(bn_{\alpha+1}r_\alpha)$. Summing these costs in all layers, we obtain an overall cost of $\sum_{\alpha=1}^{L} O\big(br_\alpha(n_\alpha + n_{\alpha+1})\big)$;

(L-forward) This cost analysis is identical to the one just presented, since L has the same shape of V and K has the same shape of U, leading to the same overall cost;

(S-forward) In this forward phase each layer is represented using variables $\widehat{U}, \widehat{S}, \widehat{V}$. The cost of the multiplication between $\widehat{V}^{(\alpha)\top}$ and $z^{(\alpha)}$ has a cost $O(2br_\alpha n_\alpha)$. The result of this multiplication has shape $2r_\alpha \times b$, so its multiplication with $\widehat{S}$ (that is diagonal) has a cost of $O(2br_\alpha)$. The result now has again shape $2r_\alpha \times b$, and its multiplication with $\widehat{U}$ has cost $O(2bn_{\alpha+1}r_\alpha)$. The overall cost of this last forward is thus $\sum_{\alpha=1}^{L} O\big(2br_\alpha(n_\alpha + n_{\alpha+1} + 1)\big)$;

The total cost of one of our forward phases during training (the sum of the costs of K,L and S steps) compared to the vanilla approach (indicated as full) is:

$$
\begin{aligned}
C_{\text{time}} &= \sum_{\alpha=1}^{L} O\Big(2br_\alpha(2n_\alpha + 2n_{\alpha+1} + 1)\Big) \lesssim \sum_{\alpha=1}^{L} O\Big(4br_\alpha(n_\alpha + n_{\alpha+1} + 1)\Big) \\
C_{\text{full,time}} &= \sum_{\alpha=1}^{L} O\Big(bn_\alpha n_{\alpha+1}\Big)
\end{aligned}
\tag{5.6}
$$

Since all operations are performed in reverse during backpropagation, its cost analysis is similar to this one.
In this case, a theoretical layerwise time advantage is attained if:

$$
0 \le r_\alpha \le \frac{n_\alpha n_{\alpha+1}}{4(n_\alpha + n_{\alpha+1} + 1)} \tag{5.7}
$$

Space complexity during inference

During inference, it is sufficient to keep in memory $K^{(\alpha)}$ and $V^{(\alpha)}$ for each layer. This leads to a memory consumption comparison given by:

$$
\begin{aligned}
C_{\text{memory}} &= \sum_{\alpha=1}^{L} O\Big(r_\alpha(n_{\alpha+1} + n_\alpha)\Big) \\
C_{\text{memory full}} &= \sum_{\alpha=1}^{L} O\Big(n_\alpha n_{\alpha+1}\Big)
\end{aligned}
\tag{5.8}
$$

The layer-wise compression ratio in this case is positive if and only if:

$$
0 \le r_\alpha \le \frac{n_\alpha n_{\alpha+1}}{n_\alpha + n_{\alpha+1}} \tag{5.9}
$$

Time complexity during inference

As we said earlier, during inference it is sufficient to consider $K^{(\alpha)}$ and $V^{(\alpha)}$. In particular, following section 5.1.1, we obtain a comparison given by:

$$
\begin{aligned}
C_{\text{time}} &= \sum_{\alpha=1}^{L} O\Big(br_\alpha(n_{\alpha+1}+n_\alpha)\Big) \\
C_{\text{time full}} &= \sum_{\alpha=1}^{L} O\Big(bn_\alpha n_{\alpha+1}\Big)
\end{aligned} \tag{5.10}
$$

leading to a layer compression ratio equal to the one in Eq.(5.9).

5.1.2 Efficient forward phase

In this section we present a complexity analysis of Alg.3.1 in the case where the full gradients with respect to W are taped. In contrast to the gradient-efficient approach, this requires only one forward phase for each optimization step (compared to three), but the space complexity is greater because of the representation of the full derivatives, leading to a trade-off in memory and time complexity. In this implementation, the representation of each layer is different: compared to $K^{(\alpha)} \sim n_{\alpha+1} \times r_\alpha, L^{(\alpha)} \sim n_\alpha \times r_\alpha$ and $\widehat{S}^{(\alpha)} \sim 2r_\alpha \times 2r_\alpha$ we represent each layer using $U^{(\alpha)} \sim n_{\alpha+1} \times r_\alpha, S^{(\alpha)} \sim r_\alpha \times r_\alpha$ and $V^{(\alpha)} \sim n_\alpha \times r_\alpha$.

Space complexity during training

For each layer, we have to keep in memory $U^{(\alpha)}, S^{(\alpha)}$ and $V^{(\alpha)}$, so following section 5.1.1 we obtain the comparison:

$$
\begin{aligned}
C_{\text{space,no grads}} &= \sum_{\alpha=1}^{L} O\Big(r_\alpha(n_{\alpha+1}+n_\alpha+1)\Big) \\
C_{\text{space full,no grads}} &= \sum_{\alpha=1}^{L} O\Big(bn_\alpha n_{\alpha+1}\Big)
\end{aligned} \tag{5.11}
$$

By taking into account also the gradients, we obtain instead:

$$
\begin{aligned}
C_{\text{space,no grads}} &= \sum_{\alpha=1}^{L} O\Big(n_{\alpha+1}n_\alpha + r_\alpha(n_{\alpha+1}+n_\alpha+1)\Big) \\
C_{\text{time full,no grads}} &= \sum_{\alpha=1}^{L} O\Big(2n_\alpha n_{\alpha+1}\Big)
\end{aligned} \tag{5.12}
$$

It is worth to notice that if we care about asymptotic results, this second approach has no memory advantage since the dominating term is $n_\alpha n_{\alpha+1}$ in each layer.

Time complexity during training

The forward for each layer consists in sequentially multiplying the batch input with the three matrices. The computational cost comparison with the vanilla approach is the following:

$$
\begin{aligned}
C_{\text{time}} &= \sum_{\alpha=1}^{L} O\Big(br_\alpha(n_{\alpha+1} + n_\alpha + 1)\Big) \\
C_{\text{time full}} &= \sum_{\alpha=1}^{L} O\Big(bn_\alpha n_{\alpha+1}\Big)
\end{aligned}
\tag{5.13}
$$

Space complexity during inference

During inference, the only approach we can use to reduce space complexity is to multiply $U^{(\alpha)}$ and $S^{(\alpha)}$, leading to a cost of:

$$
\begin{aligned}
C_{\text{space}} &= \sum_{\alpha=1}^{L} O\Big(r_\alpha(n_{\alpha+1} + n_\alpha)\Big) \\
C_{\text{space full}} &= \sum_{\alpha=1}^{L} O\Big(n_\alpha n_{\alpha+1}\Big)
\end{aligned}
\tag{5.14}
$$

Time complexity during inference

As we said earlier, the only advantage in the inference time that we can obtain here is to multiply again $U^{(\alpha)}$ and $S^{(\alpha)}$. The comparison with vanilla forward is:

$$
\begin{aligned}
C_{\text{time}} &= \sum_{\alpha=1}^{L} O\Big(br_\alpha(n_{\alpha+1} + n_\alpha)\Big) \\
C_{\text{time full}} &= \sum_{\alpha=1}^{L} O\Big(bn_\alpha n_{\alpha+1}\Big)
\end{aligned}
\tag{5.15}
$$

5.2 Compression effects

In this section, we present some experiments concerning the effects of compression on timing and accuracy. All these experiments are performed using the gradient efficient implementation algorithm 3.1.

5.2.1 Timings

In this first experiment, we compared a layerwise compression ratio with the CPU execution time of different operations. In particular, we considered a pass on the full MNIST dataset [7] of the following operations:

1. "forward train": a forward pass during training phase of DLRT on the full MNIST (so three forwards for K,L and S representations):
2. "backward train": a forward and backward pass during training phase of DLRT (so three forwards and one backward);
3. "forward test": a forwards pass during inference phase of DLRT (so only on one representation, we used the K, V one).

This experiment has been conducted on a fully connected architecture represented in table 5.1 with the fixed rank version of algorithm 3.1. A grid of compression ratios between zero and one (with a step of 0.05) had been created, and for each one of these the first three layers were compressed by that amount (e.g., with compression ratio zero the ranks are full and with one they are set to a minimum of 2). For each compressed network, the timings had been recorded 5 times to obtain a measure of the standard deviation. Moreover, we took 5 measurements also for each one of the baselines. The results of this experiment are reported in Fig.5.1. These results are meant to show the relation between memory and time complexity of different operations, compared with a standard full rank Pytorch's implementation baseline. It is worth to notice that inference time of our approach is advantageous (on average) over a 20% layerwise compression rate, but as expected the difference is already not significant since the beginning. However, inference time of our approach starts to become significantly smaller than the baseline approach after a 45% layer-wise compression ratio.
For what concerns backpropagation time, the difference in execution time starts to be less significant around 75% layerwise compression ratio, and our approach starts to become more significantly advantageous after 85%. Comparing these results with figure 5.2, we observe that even though the timing is not advantageous before a 75% layer-wise compression ratio, an overall advantage in memory complexity during training is already achieved after 45%.
Finally, our forward during training phase starts becoming advantageous in terms of time after 80% layer-wise compression ratio. This is expected as in the backward measure, since the difference in inference time is not decreasing fast enough to compensate the fact that our forward phase consists in three steps. However, as shown in Fig.5.2, memory advantage of our approach is already detectable after a 55% without counting the gradients in the compression ratio (45% if we also count the gradients in the memory complexity).
As we will see in sections 5.3,5.2.2, even though in this experiment the time advantage starts to be visible after large layer-wise compression ratios, the performance of the neural network drops very slowly as a function of compression.
Moreover, we would expect better results regards timings on larger networks and datasets, since the over-parametrization may be more pronounced than in this example. However, to not bias results we decided to test only this architecture, since Pytorch's convolution is using compiled routines.

Layer type	activation	output shape	# parameters
Linear	Relu	500	392500
Linear	Relu	300	150300
Linear	Relu	100	30100
Linear	Softmax	10	1010
Total parameters			573910

Table 5.1: Fully connected architecture was used for the timing experiment. The input is a flattened image of Mnist, of size 784.

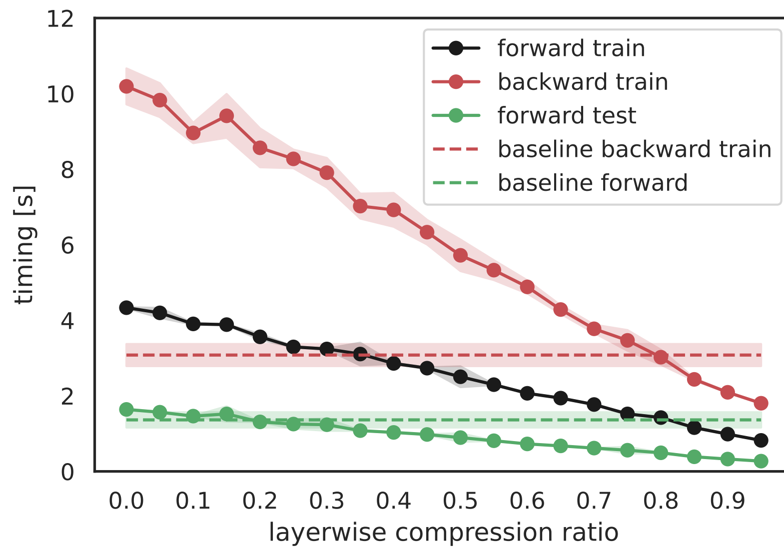

Figure 5.1: Cpu timings of different operations on full Mnist as a function of the layerwise compression ratio. The coloured area around each line indicates one standard deviation interval.

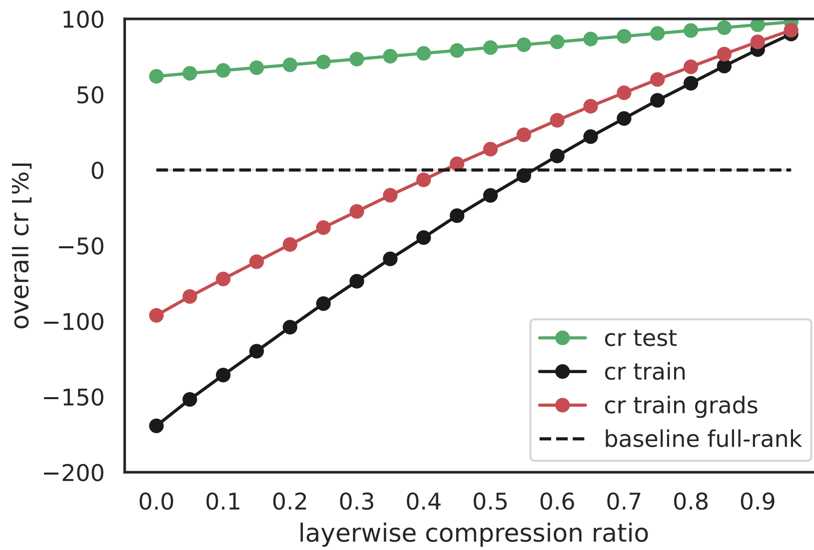

Figure 5.2: Layerwise compression ratio against overall compression ratio during different phases, to compare with Fig.5.1

5.2.2 Compression ratio and accuracy

In this second experiment, we experimentally investigated quantitatively how quickly a neural network's performance deteriorates depending on the compression rate. In particular, for this test we used Lenet5 evaluated on the MNIST dataset and as in section 5.2.1, we used the fixed rank version of Alg.3.1 in order to control the amount of compression precisely.
We created a grid of layer-wise compression ratios and for each one of these we trained five times a compressed Lenet5 on MNIST using early stopping with validation loss. Results are reported in Fig.5.3 along with the baseline (Lenet5 trained with stochastic gradient descent, early stopped).
As we can see, the accuracy drops really slow as a function of the layer-wise compression ratio until around 90% of the weights of each layer is thrown away. In particular, for 90% layer-wise compression the accuracy drop with respect to the baseline is of 2%, with a memory saving of 88% during training.
This result can be also compared with the ones presented in section 5.2.1: even though the time advantage during training is noticeable for big compression rates, it is also true that it seems that big compression rates do not influence the resulting accuracy by a big amount.

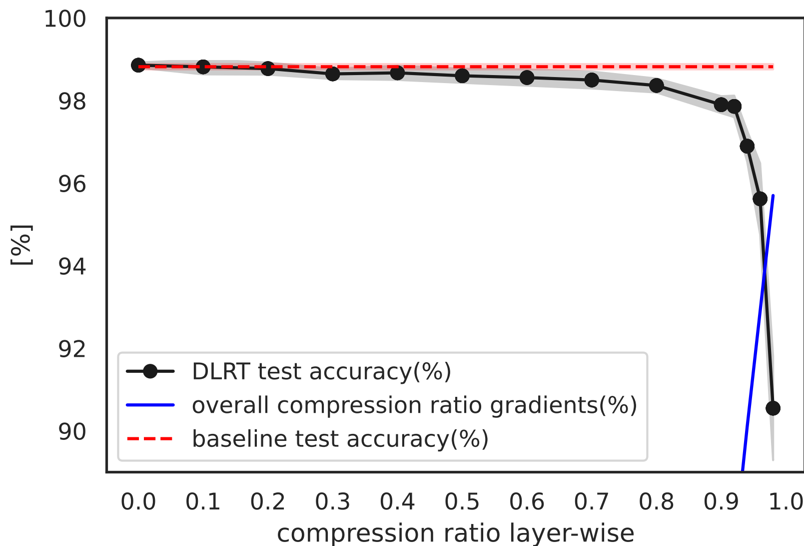

Figure 5.3: layerwise compression ratio against test accuracy after training Lenet5 on MNIST. The blue line represents the overall memory compression (counting the gradients).

5.3 Lenet on MNIST

In this experiment, we tested the adaptive version of DLRT on Lenet5 (architecture represented in Fig.3.2) using again MNIST dataset. In particular, we created a grid of relative thresholds τ (following notation of Alg.3.1) and for each one of this we performed five independent runs. In each one of the runs, the starting weight is reinitialized as random normal and the dataset is re-splitted to perform a sort of Monte Carlo cross

validation. We compared the results both with a baseline (full rank trained with standard Pytorch stochastic gradient descent, same dataset preparation) and with other compression approaches discussed in section section 3.0.1. Detailed numerical results are reported in Tab.5.2, along with some visual representations of the results reported in Fig.5.4.
In Fig.5.4 is reported the average accuracy as a function of the relative threshold, together with test and train compression ratios. As expected, the best compression is achieved with the highest $\tau = 0.45$, and it amounts to using 98.4% less weights than the original model, with a space saving during training of 95.4% on average. Furthermore, the accuracy drop with respect to the baseline with this compression rate is 4.7% on average. Without looking for the biggest compression, a good compromise between performance drop and compression is attained already with $\tau = 0.2$, which gives a 0.7% accuracy drop with respect to the baseline with a compression ratio of 96.9% during inference (83.4% memory saving during training), comparable also with all other approaches in Tab.5.2.

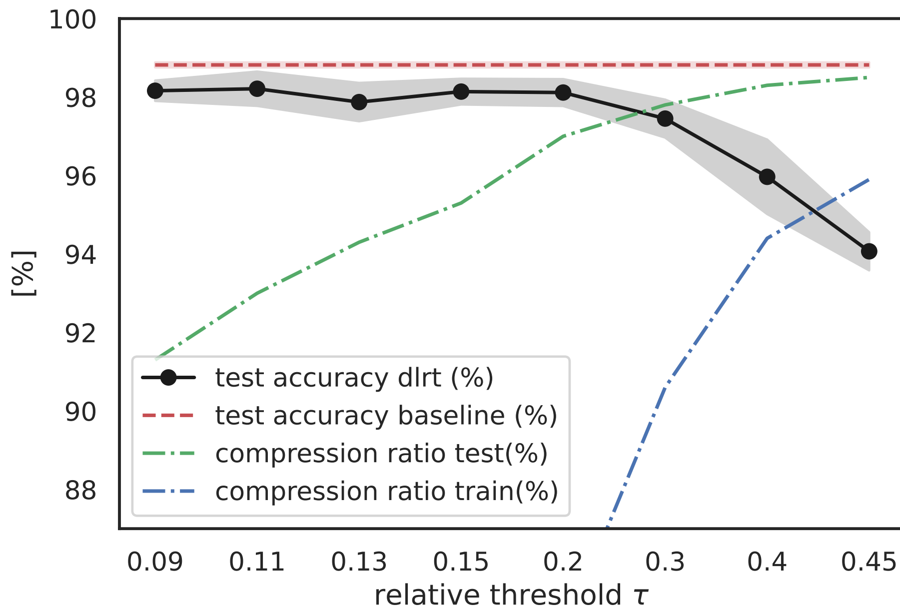

Figure 5.4: Relative singular value threshold τ of the adaptive version of Alg.3.1 against test compression ratio, train compression ratio (counting the gradients) and test accuracy (with relative full rank vanilla training baseline). Detailed numerical results are presented in Tab.5.2.

method	NN metrics: mean test acc.	NN metrics: ranks	Inference: params	Inference: c.r.	Train: params	Train: c.r.
LeNet5	$\mathbf{98.8\% \pm 0.06}$	$[20, 50, 500, 10]$	430500	0%	861000	0%
DLRT $\tau = 0.09$	$98.2\% \pm 0.26$	$[10, 23, 62, 10]$	37445	$90.9\% \pm 0.3$	532176	$35.5\% \pm 1.8$
DLRT $\tau = 0.11$	$98.2\% \pm 0.44$	$[10, 20, 48, 10]$	30278	$93.1\% \pm 0.45$	412898	$53.3\% \pm 3.5$
DLRT $\tau = 0.13$	$97.9\% \pm 0.49$	$[9, 16, 37, 10]$	24542	$94.3\% \pm 0.17$	316997	$63.2\% \pm 1.1$
DLRT $\tau = 0.15$	$98.1\% \pm 0.33$	$[9, 16, 28, 10]$	20033	$95.4\% \pm 0.23$	251477	$71.4\% \pm 1.83$
DLRT $\tau = 0.2$	$98.1\% \pm 0.34$	$[8, 8, 15, 10]$	13091	$96.9\% \pm 0.16$	135536	$83.4\% \pm 1.21$
DLRT $\tau = 0.3$	$97.5\% \pm 0.48$	$[4, 6, 8, 10]$	9398	$97.9\% \pm 0.08$	80792	$91.2\% \pm 0.59$
DLRT $\tau = 0.4$	$96.0\% \pm 0.94$	$[2, 4, 4, 10]$	7250	$98.3\% \pm 0.06$	47882	$94.4\% \pm 0.3$
DLRT $\tau = 0.45$	$94.1\% \pm 0.49$	$[2, 2, 3, 10]$	6647	$\mathbf{98.4\% \pm 0.07}$	35654	$\mathbf{95.4\% \pm 0.4}$
SSL [36] (ft)	99.18%		110000	74.4%		< 0%
NISP [39] (ft)	99.0%		100000	76.5%		< 0%
GAL [21]	98.97%		30000	93.0%		< 0%
LRNN [11]	98.67%	$[3, 3, 9, 9]$	18075	95.8%		< 0%
SVD prune [38]	94.0%	$[2, 5, 89, 10]$	123646	71.2%		< 0%

Table 5.2: Results of the training of LeNet5 on MNIST dataset. "Params" represent the number of parameters we have to save (in train and inference phases) using DLRT Alg.3.1. The compression ratio (c.r.) is the percentage of parameter reduction with respect to the full model ($< 0\%$ indicates that the ratio is negative), as explained in section 5.1.1 (for the train compression ratio we took the gradients into account). "ft" indicates that the model has been fine tuned. "LeNet5" line indicates the baseline, trained using stochastic gradient descent.

5.4 Cifar10 and Cifar100

In this experiment, we tested Alg.3.1 on AlexNet [17] and VGG16 [29], adapted to both Cifar10 and Cifar100 datasets [16]. We compared our results both with methods proposed in the recent literature and with a full rank baseline. All runs had been performed using the same hyperparameters, in particular we used a batch size of 64, learning rate of 0.05, momentum 0.1 and a threshold $\tau = 0.08$. For this experiment, we decided to test a partial compression: more precisely, all convolutional layers are trained in their full rank representation, and only the final linear layers of each network are trained with DLRT.
Results are presented in Tabels 5.3, 5.4.
In the Cifar10 table 5.3 we can notice that every model attained a worse accuracy with respect to the full-rank baseline. It is also to remark that our approach is comparable to the baselines, despite only the final fully connected layers had been compressed.
The performance deteriorates by a maximum of 3.88% with respect to the baseline, but together with an 85.1% space compression during inference and 79.2% during training (on Cifar100). The maximum compression ratio we obtained is 86.3% during inference (84.2% during training) for AlexNet on Cifar10, with a loss in test accuracy of 1.79% with respect to the full-rank baseline.
For Cifar100, VGG16 reported an improvement in test accuracy of 1.52% with respect to the baseline, but with a smaller compression ratio (51% during inference and 72% during training).

method	network	difference with baseline	c.r. [eval]	c.r. [train]
DLRT $\tau = 0.1$	VGG16	-1.89%	56%	77.5%
GAL [21]	VGG16	-1.87%	77%	$< 0\%$
LRNN [11]	VGG16	-1.9%	60%	$< 0\%$
DLRT $\tau = 0.1$	AlexNet	-1.79%	86.3%	84.2%
NISP [39] (ft)	AlexNet	-1.06%	$n.a.$	$< 0\%$

Table 5.3: 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. The "difference with baseline" column indicates the difference in final test accuracy between each method and the baseline (full rank,trained with stochastic gradient descent with the same hyperparameters).

method	difference with baseline	c.r. [eval]	c.r. [train]
VGG16	1.52%	51%	72%
AlexNet	-3.88%	85.1%	79.2%

Table 5.4: Compression results of on Cifar100. DLRT with $\tau = 0.08$ is used. The "difference with baseline" column indicates the difference in final test accuracy between each method and the baseline (full rank,trained with stochastic gradient descent with the same hyperparameters).

5.5 Robustness to small singular values

As already mentioned in section 2.7.1, a straighforward numerical integration of the gradient flow equations of the factors in the case of an U, S, V factorization would lead to stability problems due to the inversion of S. Unfortunately, this issue is not to be imputed to the particular parametrization, but it is intrinsic due to curvature of the manifold of low-rank matrices [15, 32] (explained in lemma 2.7.1). As pointed out in the same section, the unconventional integrator (and its rank-adaptive counterpart) are able to exploit the ruled structure of the constraint manifold and to move along flat subspaces during the K and L integration steps, "avoiding to see the curvature". This robustness to small singular values is not shared with other numerical integrators in general.
In this experiment, we show numerically the difference of training by using DLRT (the fixed rank version to have a fair comparison) and a truncated $UV^\top$ layer decomposition trained with stochastic gradient descent in the scenario in which the starting singular value distribution has a fast decaying tail.

This type of approach is used for example in [13, 35]. Results of the experiment are presented in Figures 5.5,5.6.
We initialized Lenet5 weights as normally distributed, we computed a singular value decomposition layer-wise, and we scaled singular values by making them decay with powers of two, ten and with no decay as a baseline. In the case of layer factorization, we multiplied together U and S in order to have a $UV^\top$ decomposition for each layer.
After this initialization, we trained ten times these neural networks on MNIST for ten epochs (each time with a different weight initialization and a different split of the dataset), to compare DLRT with a vanilla layer factorization approach (same hyperparameters, batch size of 128 and learning rate of 0.01,ranks fixed

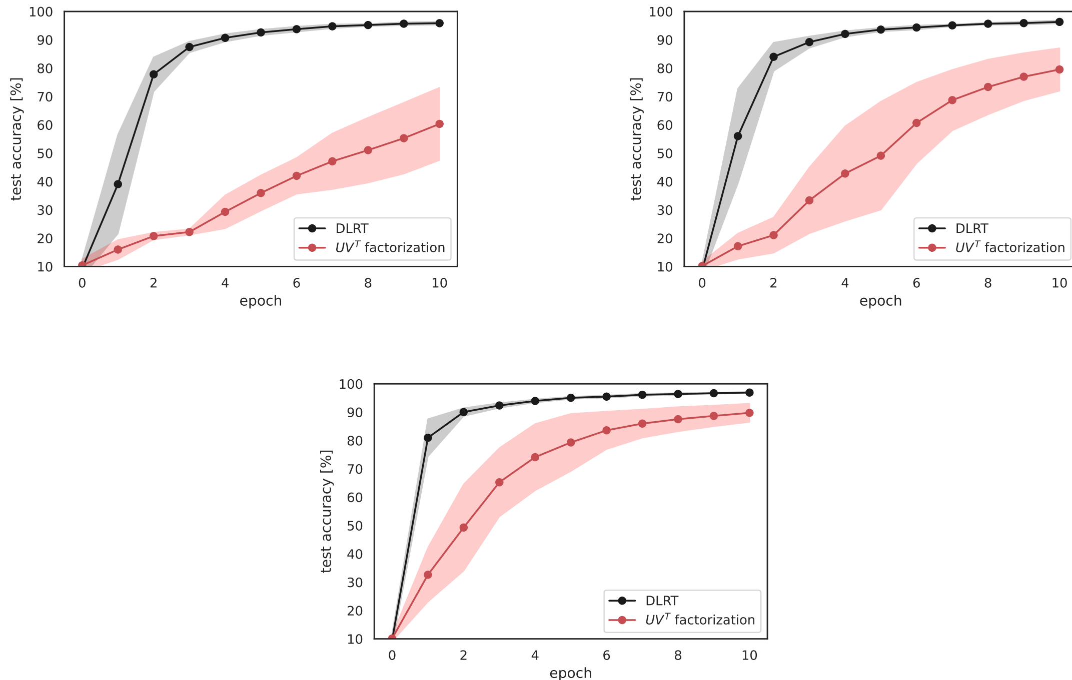

Figure 5.5: Accuracy over epochs of DLRT and and layer factorization [13] on Lenet5 architecture trained on Mnist. Decay with powers of 10 (top left), with powers of 2 (top right), and no decay (bottom).

to $[20, 20, 20, 10]$).
As shown the results, at parity of integration step, if the starting point is around a high curvature region in the manifold, the convergence to equilibrium of a non tailored numerical integration is slower (Fig.5.6). In terms of the final performance of the model, in Fig.5.5 the same effect is also visible in the test accuracy. It is also worth to notice that the faster the decay of the singular values, the slower the convergence of stochastic gradient descent on the layer factorization (red lines). This effect is not so visible by using the unconventional integrator (black lines).

5.6 Fine tuning effectiveness

In this experiment, we tested the effectiveness of the fine tuning strategy proposed in section 5.6. For this purpose, we decided to test different networks on Cifar10 in an overcompression setting. More precisely, we trained different models by overcompressing on purpose, to see how effective is fine tuning for improving the performance. For each architecture, we trained for 20 epochs (with a batch size of 64 and a learning rate of 0.05) and then we fine tuned for other 20. Results are reported in Tab.5.5.
The results show that the fine tuning procedure introduced in section 5.6 is able to significantly improve the

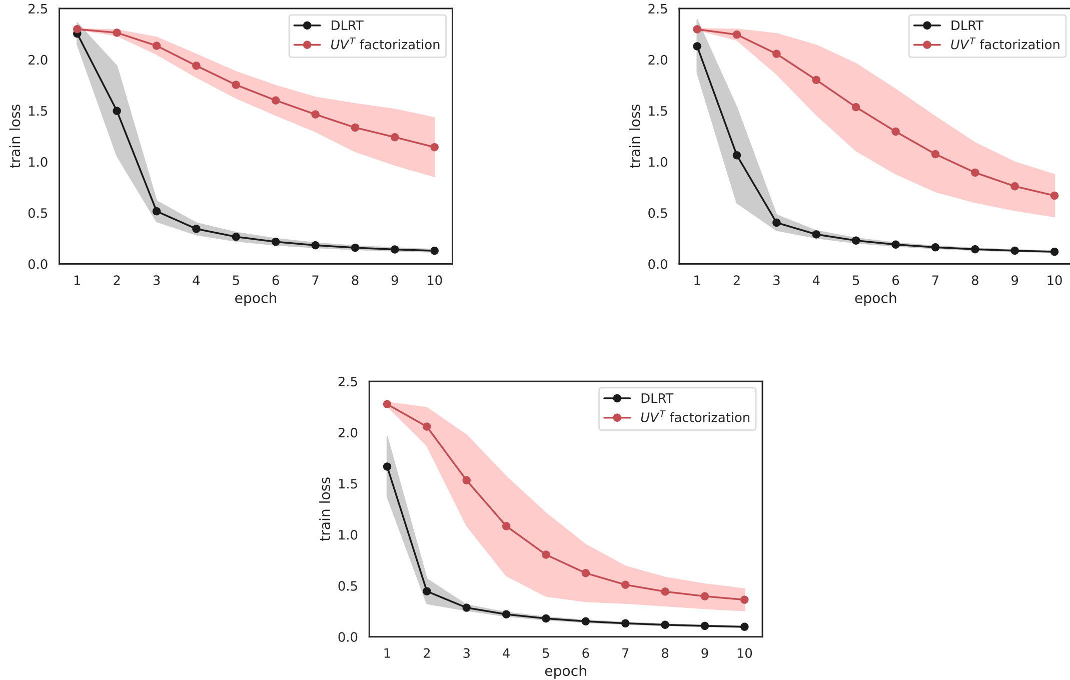

Figure 5.6: Training loss over epochs of DLRT and and layer factorization [13] on Lenet5 architecture trained on Mnist. Decay with powers of 10 (top left), with powers of 2 (top right), and no decay (bottom).

performance even when the compressed neural network is underparametrized, with a maximum improvement in the final test accuracy of 28.11% for VGG16 and a minimum of 14.63% for Lenet.

Model	Test accuracy	After fine tuning	improvement	c.r. [eval]
VGG	$39.54\% \pm 6.33$	$67.65\% \pm 1.42$	$28.11\% \pm 6.21$	$97.58\% \pm 0.08$
AlexNet	$42.89\% \pm 3.64$	$58.61\% \pm 1.47$	$15.72\% \pm 3.04$	$97.12\% \pm 0.044$
Lenet	$44.04\% \pm 5.24$	$58.67\% \pm 1.83$	$14.63\% \pm 4.46$	$94.84\% \pm 0.11$
Resnet20	$45.84\% \pm 7.61$	$64.24\% \pm 0.61$	$18.40\% \pm 8.02$	$92.88\% \pm 0.16$

Table 5.5: Effects of fine tuning on underparametrized networks. In the table we report the final test accuracy without fine tuning (first column), and after fine tuning (second column). In the third column, we report the mean improvement after fine tuning. In the last column we report the overall test compression ratio achieved during the training. In each column a standard deviation across five independent runs is reported.

6
Implementation

This chapter contains some details about the actual Pytorch [25] implementation of Alg.3.1.

6.1 Low-rank Module

As mentioned in section 3.0.2, we implemented custom versions of linear and convolutional layers to be trained using Alg.3.1. To construct a neural network trainable using our implementation it is necessary that each neural network module contains:

- our custom layers;
- a **layer** attribute, an iterable object containing the layers of the neural network;
- a **populate_gradients** method (provided in our implementation, it doesn't change among different architectures);
- a **update_step** methods, that updates each low-rank layer variable representation.

6.1.1 Populate_gradients method

To handle the fact that DLRT requires three forward phases at each optimization step, each custom layer of the neural network has a **step** attribute that can be K, L or S. This attribute can be updated during the training with the **update_step** method, and it changes the layer representation to the chosen group of variables (e.g, if the K step is chosen, the forward will use the K, V representation for that layer). To tape all the gradients needed for the optimization step, it is therefore necessary to perform three forwards for each input batch (one for each **step** value). As described in Alg.3.1, the gradients with respect to the variable $\widehat{S}$ have to be recorded after having updated K and L, thus requiring the use of a closure function to be passed

to the optimizer. This kind of trick is standard in Pytorch, since it allows to perform backpropagation during the optimization step.
In our implementation each neural network has to possess a **populate_gradients** method, that takes as input a batch of data, a loss function and the step variable for which we want to record the gradients. This method updates the step variable of each low-rank layer to the desired one, computes the loss, and backpropagates. There is a simple trick that allows one to perform only one backpropagation for the K and L steps together instead of two. This boils down to considering the sum of the losses for the three phases:

$$J = J_K + J_L \tag{6.1}$$

where J_K is the loss calculated using the K, V representation for each layer (the same for L). Since no variable is shared among these representations, calculating the derivative of the sum with respect to a parameter reduces to calculating the derivative of the corresponding individual loss. More precisely:

$$\begin{cases} \nabla_K J = \nabla_K J_K \\ \nabla_L J = \nabla_L J_L \end{cases} \tag{6.2}$$

This observation shows that it is possible to compute the derivatives of the total loss J, and thus it is possible to call Pytorch's method backward only once.

Algorithm 6.1 Pseudo code for gradient taping

Input: f: torch.nn.Module object with custom low-rank layers and needed methods;
(x, y): batch of data;
ℓ: Pytorch's loss function;
opt : Custom dynamical low-rank optimizer object;

```
45 loss = f.populate_gradients(x,y,ℓ);                      /* tape gradients for K and L */
46 def closure():                                   /* closure function to tape S gradients. */
47     loss_S = f.populate_gradients(x,y,ℓ,step = S);
       return loss_S;
   opt.step(closure = closure)                                        /* optimization step */
```

6.2 Optimizer class

The most convenient way to implement DLRT in Pytorch was to build a custom optimizer class. In the implementation, our optimizer takes as input a torch.nn.Module instance (implemented with our custom low-rank layers, as explained in 6.1), the threshold τ for the ranks, and a Pytorch optimizer with its additional arguments. In particular, since forward Euler was used in the unconventional integrators, in all experiments we used stochastic gradient descent. The choice of using an already implemented optimizer to perform the integrator step has advantages, since it allows one to easily modify the training procedure by adding techniques such as momentum, learning rate schedulers, or regularization. In the next section, we will give

a brief explanation of the main methods of this class.

6.2.1 Preprocess steps

These methods (**K_preprocess_step,L_preprocess_step** and **S_preprocess_step**) correspond to the update of K,L and S before each integration step (as indicated in Alg.3.1). As described in the pseudocode of the training algorithm, each one of these methods has to be called before its relative integration step.

6.2.2 Integration steps

As described in the original paper, the integration steps for K and L are completely parallelizable. For this reason, after having recorded the gradients for each K and L variable, a unique optimizer step is needed to integrate both differential equations (by calling **K_and_L_integration_step** inside the optimizer step method).
For what concerns the integration step for S, it is performed after recording the gradients through the closure function and excluding all the other ones.

6.2.3 Postprocess steps

Lastly, for each representation there is a corresponding **postprocess** method. This corresponds to the updating of U, V, M and N after the K and L integration steps. The S postprocess step corresponds instead to the update after integration and to the eventual rank adaption in the corresponding layers.

6.2.4 Step method

Step is the main method of the custom optimizer. As input, it accepts a closure function to record the gradients of S after the integration of the other variables.
When called, it automatically performs the integration step for K and L in each layer, then it tapes the gradients with respect to S to perform the last integration step. In practice, the step method executes all operations described in alg.3.1.

6.2.5 Fine tuning

In order to fine tune an already trained model, the optimizer contains a method **activate_S_fine_tuning**. This method deactivates the gradients of all variables except S in each layer, and it updates the default step to the S one. Moreover, it fixes the rank of each adaptive layer to the current one. After calling this method, the neural network can be trained using standard Pytorch's syntax.

7
Conclusions and future improvements

In this thesis, we presented DLRT, an end-to-end approach to compress neural networks that exploits theory of ordinary differential equations together with recent improvements in dynamical low-rank theory. This founding theory serves as a theoretical justification for our approach, allowing to exploit the remarkable properties of the unconventional integrators, as reported in the main papers [5, 18]. Moreover, rank adaptivity allows to leave each layer choose its rank dynamically during the training, without the need of an a priori choice about it. The advantage of our proposal lies in the memory complexity during the training: unlike other methods with which we compared (briefly presented in chapter 4), beyond the memory advantage during the test phase, DLRT is also able to give a memory reduction during the training phase. This peculiarity can be exploited in situations in which it is not an option to do a full-memory training, for example if there is the need to train on a limited memory device.
In the experiment section, we quantitatively investigated some advantages of this approach. In the first experiment section 5.2.1 we compared the timing of our approach with a full-rank baseline (with standard Pytorch implementation) on some critical operations performed both during the training and inference using a neural network. The results show that our implementation can be advantageous in time during the training phase given that the ranks are sufficiently small. Nevertheless, in this experiment we also showed that even in the regime in which our approach is not advantageous in terms of timing, DLRT is already saving more than 70% of the required memory for training with respect to the baseline. Concerning inference, DLRT shows both memory and time advantage already in the small compression regime.
The first experiment may raise some questions, since the compression ratio needed to achieve a time advantage in terms of training step was high. The second section of the experiment 5.2.2 is meant to investigate that: even if the amount of compression needed to achieve this time advantage is high, the performance of the neural network is almost not decreasing until a compression ratio of 80%. Going further, the accuracy dropped on average of a 4% with approximately a 96% compression ratio. This shows that it is possible to reach the regime in which our approach, more than being memory advantageous during training, is also time advantageous, and all this with a relatively low performance degradation.

In the third and fourth experiments sections 5.3,5.4 we tested DLRT on some benchmark datasets (MNIST, Cifar10 and Cifar100) and neural networks (Lenet5, VGG16, AlexNet and Resnet20). We compared our results with both a full rank baseline and with some other literature (briefly presented in chapter 4). Results showed that our approach is comparable to our baseline in terms of performance, and it is even comparable with the other methods for what concerns the compression ratio during inference time.
In the last two experiments sections 5.5,5.6 we showed respectively the robustness of our approach with respect to small singular values and the effectiveness of the fine tuning we proposed.
As shown in section 5.2.1, DLRT has also some limitations: a memory and time complexity reduction during the training phase is obtained only if the ranks are sufficiently small. If this problem is not concerning for medium-sized neural networks with medium complexity datasets, this advantage may be not so pronounced for bigger neural networks fitted on bigger datasets, where the overparameterization may be not so pronounced, leading to final higher ranks.
Future work may be focused on testing DLRT on other benchmark datasets, like Imagenet. More than this, it would be interesting to implement and test DLRT in different ways for convolutional layers, and maybe to extend it for other layers like three-dimensional convolution.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/file/62dad6e273d32235ae02b7d321578ee8-Paper.pdf.

[2] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm with application to wavelet-based image deblurring. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 693–696, 2009. doi: 10.1109/ICASSP.2009.4959678.

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020. URL https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[4] Miguel Á. Carreira-Perpiñán. Model compression as constrained optimization, with application to neural nets. part I: general framework, 2017. URL https://arxiv.org/abs/1707.01209.

[5] Gianluca Ceruti and Christian Lubich. An unconventional robust integrator for dynamical low-rank approximation. *BIT Numerical Mathematics*, 05 2021. doi: 10.1007/s10543-021-00873-0.

[6] Frédéric Chazal and Bertrand Michel. An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists. *Frontiers in Artificial Intelligence*, September 2021. doi: 10.3389/frai.2021.667963. URL https://hal.inria.fr/hal-01614384.

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

[8] Vincent Fortuin, Adrià Garriga-Alonso, Sebastian W. Ober, Florian Wenzel, Gunnar Ratsch, Richard E Turner, Mark van der Wilk, and Laurence Aitchison. Bayesian neural network priors revisited. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=xkjqJYqRJy.

[9] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJl-b3RcF7.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. Low-rank compression of neural nets: Learning the rank of each layer. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8046–8056, 2020. doi: 10.1109/CVPR42600.2020.00807.

[12] Alan Izenman. Introduction to manifold learning. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4, 09 2012. doi: 10.1002/wics.1222.

[13] Mikhail Khodak, Neil A. Tenenholtz, Lester Mackey, and Nicolò Fusi. Initialization and regularization of factorized neural layers. In *ICLR*, 2021. URL https://openreview.net/forum?id=KTlJT1nof6d.

[14] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11): 3964–3979, nov 2021. doi: 10.1109/tpami.2020.2992934. URL https://doi.org/10.1109%2Ftpami.2020.2992934.

[15] Othmar Koch and Christian Lubich. Dynamical low-rank approximation. *SIAM Journal on Matrix Analysis and Applications*, 29(2):434–454, 2007. doi: 10.1137/050639703. URL https://doi.org/10.1137/050639703.

[16] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[18] Jonas Kusch, Gianluca Ceruti, and Christian Lubich. A rank-adaptive robust integrator for dynamical low-rank approximation. *BIT. Numerical Mathematics*, 2022. doi: 10.48550/ARXIV.2104.05247.

[19] John M. Lee. Introduction to smooth manifolds. Springer New York, NY, 2002. doi: https://doi.org/10.1007/978-1-4419-9982-5.

[20] Na Lei, Dongsheng An, Yang Guo, Kehua Su, Shixia Liu, Zhongxuan Luo, Shing-Tung Yau, and Xianfeng Gu. A geometric understanding of deep learning. *Engineering*, 6(3):361–374, 2020. ISSN 2095-8099. doi: https://doi.org/10.1016/j.eng.2019.09.010. URL https://www.sciencedirect.com/science/article/pii/S2095809919302279.

[21] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[22] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Li Shen, Decebal Constantin Mocanu, Zhangyang Wang, and Mykola Pechenizkiy. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=VBZJ_3tz-t.

[23] Christian Lubich and Ivan Oseledets. A projector-splitting integrator for dynamical low-rank approximation. *BIT*, 54, 01 2013. doi: 10.1007/s10543-013-0454-0.

[24] Frank Nielsen. The many faces of information geometry. *Notices of the American Mathematical Society*, 69:36–45, 01 2022. doi: 10.1090/noti2403.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[26] Antonio Romano and Mario Mango Furnari. *Introduction to Differentiable Manifolds*, pages 39–82. Springer International Publishing, Cham, 2019. ISBN 978-3-030-27237-1. doi: 10.1007/978-3-030-27237-1_2. URL https://doi.org/10.1007/978-3-030-27237-1_2.

[27] Steffen Schotthöfer, Emanuele Zangrando, Jonas Kusch, Gianluca Ceruti, and Francesco Tudisco. Low-rank lottery tickets: finding efficient low-rank neural networks via matrix differential equations, 2022. URL https://arxiv.org/abs/2205.13571.

[28] Damien Scieur, Vincent Roulet, Francis Bach, and Alexandre d'Aspremont. Integration methods and optimization algorithms. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://proceedings.neurips.cc/paper/2017/file/bf62768ca46b6c3b5bea9515d1a1fc45-Paper.pdf.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2015.html#SimonyanZ14a.

[30] Samuel L. Smith, Erich Elsen, and Soham De. On the generalization benefit of noise in stochastic gradient descent. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20, 2020.

[31] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.

[32] André Uschmajew and Bart Vandereycken. *Geometric Methods on Low-Rank Matrix and Tensor Manifolds*, pages 261–313. Springer International Publishing, Cham, 2020.

[33] Martin J. Wainwright. *High-Dimensional Statistics: A Non-Asymptotic Viewpoint*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2019. doi: 10.1017/9781108627771.

[34] Hanna Walach. Time integration for the dynamical low-rank approximation of matrices and tensors. In *7 Mathematisch-Naturwissenschaftliche Fakultät*, 2019. doi: DOI:10.15496/PUBLIKATION-31613.

[35] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. Pufferfish: Communication-efficient models at no extra cost. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 365–386, 2021. URL https://proceedings.mlsys.org/paper/2021/file/84d9ee44e457ddef7f2c4f25dc8fa865-Paper.pdf.

[36] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2082–2090, 2016. ISBN 9781510838819.

[37] Peter M. Williams. Bayesian regularization and pruning using a laplace prior. *Neural Computation*, 7(1):117–143, 1995. doi: 10.1162/neco.1995.7.1.117.

[38] Huanrui Yang, Minxue Tang, Wei Wen, Feng Yan, Daniel Hu, Ang Li, Hai Li, and Yiran 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 (CVPR) Workshops*, June 2020.

[39] Jianbo Ye, Xin Lu, Zhe Lin, and James Z. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HJ94fqApW.

[40] Han Zhao, Yao-Hung Hubert Tsai, Russ R Salakhutdinov, and Geoffrey J Gordon. Learning neural networks with adaptive regularization. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/file/2281f5c898351dbc6dace2ba201e7948-Paper.pdf.

[41] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven Chu Hong Hoi, and Weinan E. Towards theoretically understanding why sgd generalizes better than adam in deep learning. In *Advances in Neural Information Processing Systems*, volume 33, pages 21285–21296, 2020. URL https://proceedings.neurips.cc/paper/2020/file/f3f27a324736617f20abbf2ffd806f6d-Paper.pdf.

Acknowledgments

I would like to thank everyone that supported me in the process of writing this thesis. In particular, I would like to express my gratitude to my supervisor Prof. Francesco Rinaldi, my co-supervisor Prof. Francesco Tudisco and to Dr. Gianluca Ceruti, whose suggestions had been crucial for the writing.
I would also like to thank all people I had the occasion to collaborate with during my internship, without whom this stimulating experience would not have been possible.