Efficient training of low-rank neural networks

Francesco Tudisco[1]
Emanuele Zangrando[1], Steffen Schotthöfer[2], Jonas Kusch[3], Gianluca Ceruti[4]

RoMaDS Seminar @ University of Rome "Tor Vergata"
February 8, 2023

GSSI • Gran Sasso Science Institute • L'Aquila (Italy)

1 GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

2 KIT Karlsruher Institut für Technologie

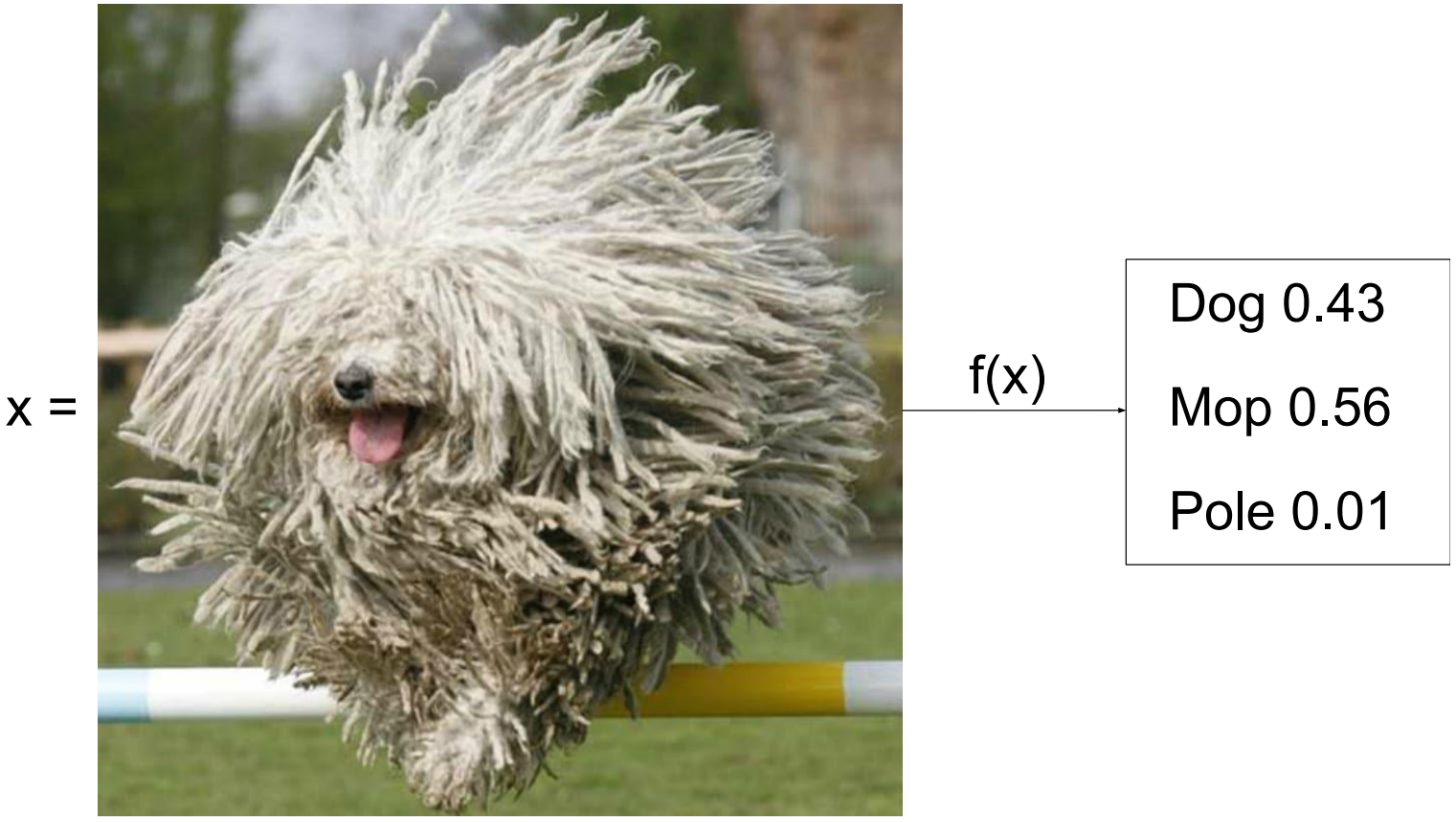
3 universität innsbruck

4 EPFL

Supervised learning

Input: Set of observations $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}^k$ for $i = 1, \ldots, N$

Goal: Infer the map $f : x \mapsto y$ in a **fast** and **reliable** way



Learning via Feed-forward NNs

In order to compute f we need an ansatz function space

Learning via Feed-forward NNs

In order to compute f we need an ansatz function space

FFNNs are a parametric family of functions that we can write as

$$f(x; \boldsymbol{W}) = \{\sigma_\ell \circ A_\ell \circ \cdots \circ \sigma_1 \circ A_1\}(x)$$

where

- $\boldsymbol{W} = (W_1, \ldots, W_\ell)$ are the parameters
- $A_i : \mathbb{R}^{N_1} \to \mathbb{R}^{N_{i+1}}$ are "simple mappings", for instance
  - Matrix multiplication $A_i(x) = W_i x$
  - Convolution $A_i(x) = W_i * x$
  - Dot product $A_i(x) = W_i x x^\top W_i^\top$
- σ_i are entrywise nonlinear *activation* functions

Memory and computation footprints

Each time we do inference, we need to evaluate f on a new point x.

This requires:

- To evaluate ℓ mappings A_i $\quad \approx \sum_i N_i N_{i+1}$
- To evaluate ℓ activations $\quad \approx \sum_i N_i$
- To store ℓ weight matrices W_i $\quad \approx \sum_i N_i N_{i+1}$

Overall, for $N_i = N$,

$O(\ell N^2)$ operations $\qquad$ $O(\ell N^2)$ variables

In practice (examples of ℓN^2)

- AlexNet ..... $60+$ million parameters
- VGG16 ..... $135+$ million parameters
- DALL-E ..... $3.5+$ billion parameters
- M-BERT ..... $4+$ billion parameters

(Very) Prohibitive for online-learning and for limited-resource devices, such as smartphones, drones, satellites, etc.

Model compression, aka **pruning**

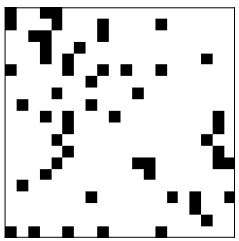
Constrain the parameter space.

Most common examples:

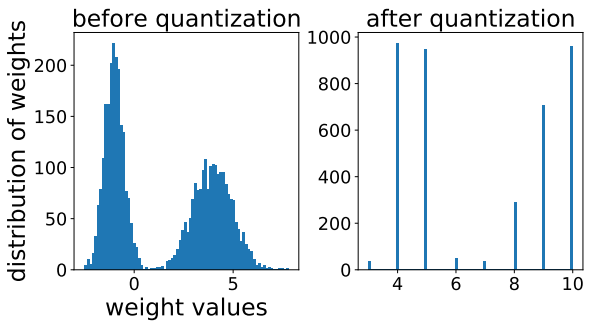
sparsity: $\mathrm{nnz}(W_i) = O(\sqrt{N_i N_{i+1}})$;

quantization: $(W_i)_{ij} \in \mathbb{Z}$;

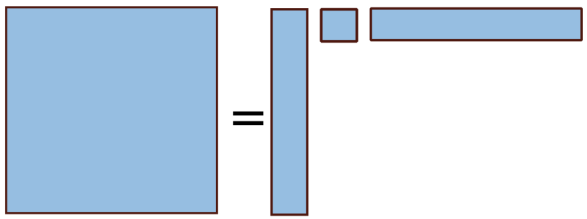
low-rank structure: $\mathrm{rank}(W_i) = r_i$.



Pruning methods



Weight quantization

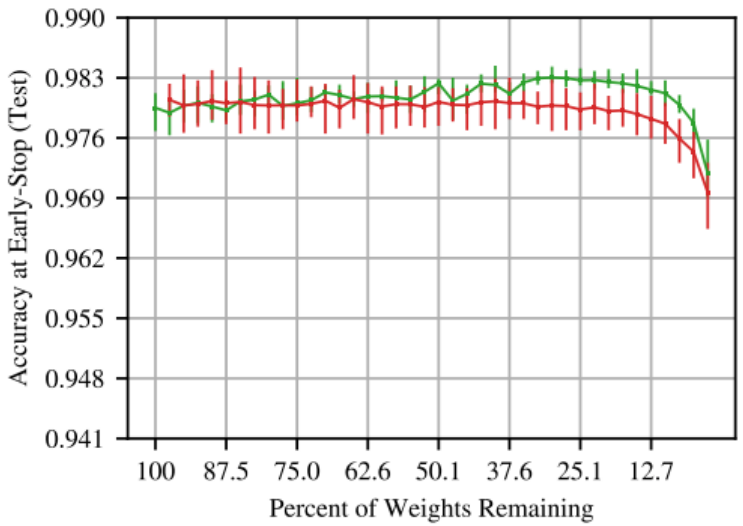


Low-rank layer factorization

Lottery ticket hypothesis

The lottery ticket hypothesis: finding sparse, trainable neural networks, J Frankle, M Carbin, ICLR 2019

A randomly-initialized dense NN contains a subnetwork that, when trained in isolation and for the same number of iterations, can match the accuracy of the original full net.



Full network vs pruned network as level of pruning increases on LeNet5

In practice

- Train. Train the full network

  ↓

- Prune. Prune the obtained optimal weights

  ↓

- Adjust. Fix the obtained constraints space and “fine tune”

Is this happening by chance?

Proving the lottery ticket hypothesis, E Malach, G Yehudai, S Shalev-Shwartz, O Shamir, ICML 2020

"A network of depth ℓ can be approximated arbitrarily well by pruning a network of depth 2ℓ"

Given $\varepsilon > 0$ and a NN f of depth ℓ and width N, there exists:

- a larger network g of depth 2ℓ and width $\mathrm{poly}(\ell, N, \varepsilon^{-1})$
- a subnetwork $\widetilde{f}$ with only $O(\ell N^2)$ parameters

such that $|\widetilde{f} - f| \leq \varepsilon$ with probability at least $(1 - \varepsilon)$

Problem(s): the result is not constructive and it gives no proof of the existence of winning tickets

How about the training phase?

Moreover,

This approach **completely ignores training costs**.

How about the training phase?

Moreover,

This approach **completely ignores training costs**.

Learning boils down to the following optimization problem,

$$\min_{\boldsymbol{W}} \ \mathrm{loss}(\boldsymbol{W};\, x, y) := \mathrm{fit}(f(x; \boldsymbol{W}), y) + \mathrm{reg}(\boldsymbol{W})$$

which we typically solve via stochastic first-order methods.

For example, for SGD

$$\boldsymbol{W}^{(n+1)} = \boldsymbol{W}^{(n)} - \lambda_n \nabla_{\boldsymbol{W}} \mathrm{loss}(\boldsymbol{W}^{(n)};\, x_{\mathrm{batch}(n)}, y_{\mathrm{batch}(n)})$$

Training cost

Computing $\boldsymbol{W}^{(n+1)}$ from $\boldsymbol{W}^{(n)}$ essentially requires

1. At least one evaluation of $f(x_{\text{batch}(n)}; \boldsymbol{W}^{(n)})$
2. At least one evalutation of $\nabla_{\boldsymbol{W}} \text{loss}(\boldsymbol{W}^{(n)}; x_{\text{batch}(n)}, y_{\text{batch}(n)})$
3. Storing at least the current variable $\boldsymbol{W}^{(n)} = (\boldsymbol{W}_1^{(n)}, \dots, \boldsymbol{W}_\ell^{(n)})$

Overall, for $N_i = N$ (ignoring nonlinearities and batch sizes):

- 1 and 2 cost $O(2\ell N^2)$ operations,
- 3 requires to store $O(\ell N^2)$ parameters,

per iteration!

Train and (then) compress

- Train.
- ↓
- Prune.
- ↓
- Adjust.

requires

- full training cost
- finding the right constraints space
- cost of pruning (e.g. projection) and fine-tuning

Train and (then) compress

- Train.
- ↓
- Prune.
- ↓
- Adjust.

requires

- full training cost
- finding the right constraints space
- cost of pruning (e.g. projection) and fine-tuning

We design an approach that overcomes all these issues by using low-rank constraints

Low-rank setting

In our approach, we compress by reducing the rank.

Consider the following "low-rank parameter space"

$$\mathcal{M} = \mathcal{M}_{r_1} \times \cdots \times \mathcal{M}_{r_\ell} \qquad \mathcal{M}_{r_i} = \{W_i : \operatorname{rank}(W_i) = r_i\}$$

with $r_i \ll \min\{N_i, N_{i+1}\}$, and the associated constrained training

$$\min_{\boldsymbol{W} \in \mathcal{M}} \operatorname{loss}(\boldsymbol{W}; x, y)$$

Complexity comparison

Operations within $\mathcal{M}$ are potentially much cheaper due to the compositional structure of NNs.

For example, if we parametrize $\mathcal{M}_{r_i}$ as

$$\mathcal{M}_{r_i} = \{U_i S_i V_i^\top : U_i \sim N_{i+1} \times r_i,\, S_i \sim r_i \times r_i,\, V_i \sim r_i \times N_i\},$$

we have

$$\begin{array}{llll} W_i \in \mathcal{M}_{r_i} & \sigma_i(W_i x) & \text{costs} & O(r_i(N_i + N_{i+1} + r_i)) \\ W_i \text{ generic} & \sigma_i(W_i x) & \text{costs} & O(N_i N_{i+1}) \end{array}$$

(a similar comparison holds for the memory storage)

Previous approaches: penalized loss and projected gradient

Impose the constraint $\boldsymbol{W} \in \mathcal{M}$ by

Previous approaches: penalized loss and projected gradient

Impose the constraint $\boldsymbol{W} \in \mathcal{M}$ by

- projecting the iterations at each step onto $\mathcal{M}$

  H Yang et al, CVPR 2020

$$P_{\mathcal{M}}(\boldsymbol{W}) = \operatorname{argmin}_{A \in \mathcal{M}} \|A - \boldsymbol{W}\|_F^2$$

Previous approaches: penalized loss and projected gradient

Impose the constraint $\boldsymbol{W} \in \mathcal{M}$ by

- projecting the iterations at each step onto $\mathcal{M}$

  H Yang et al, CVPR 2020

  $$P_{\mathcal{M}}(\boldsymbol{W}) = \operatorname{argmin}_{A \in \mathcal{M}} \|A - \boldsymbol{W}\|_F^2$$

- add a penalty term $C(\boldsymbol{W})$ to the loss, for example

  Y Idelbayev, MA Carreira-Perpiñán, CVPR, 2020

  $$C(\boldsymbol{W}) = \sum_{i=1}^{L} \alpha_i \operatorname{rank}(W_i)$$

Previous approaches: penalized loss and projected gradient

Impose the constraint $\boldsymbol{W} \in \mathcal{M}$ by

- projecting the iterations at each step onto $\mathcal{M}$

  H Yang et al, CVPR 2020

  $$P_{\mathcal{M}}(\boldsymbol{W}) = \operatorname{argmin}_{A \in \mathcal{M}} \|A - \boldsymbol{W}\|_F^2$$

- add a penalty term $C(\boldsymbol{W})$ to the loss, for example

  Y Idelbayev, MA Carreira-Perpiñán, CVPR, 2020

  $$C(\boldsymbol{W}) = \sum_{i=1}^{L} \alpha_i \operatorname{rank}(W_i)$$

- Require computing the full-rank flow during training
- Require at least one SVD at each step
- Require to choose the ranks a-priori

Our approach

We propose a strategy based on Dynamic Low-Rank Approximation from model order reduction of matrix differential equations.

Main properties:

- Assuming $W_i^{(n)} = U_i S_i V_i^\top$, computes $W_i^{(n+1)}$ with the same structure using only the factors
- Adaptively adjusts the rank r_i of S_i

Gradient flow formulation

We can rephrase the training problem as the search of equilibrium points for the matrix ODE

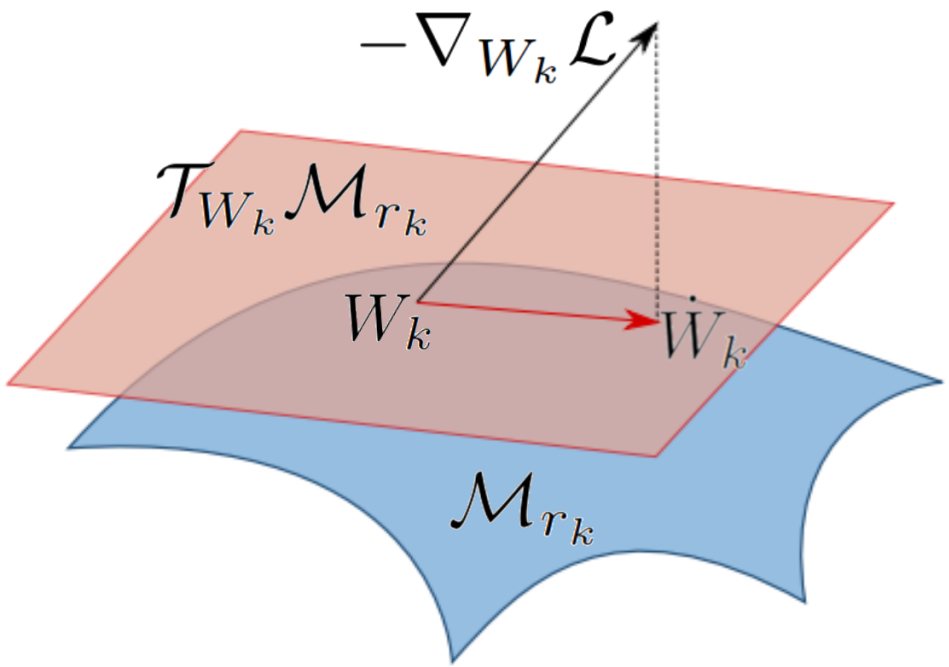
$$\frac{d}{dt}\boldsymbol{W}(t) = -\nabla_{\boldsymbol{W}} \operatorname{loss}(\boldsymbol{W}(t);\, x, y)$$

With this notation, for example, GD $=$ Explicit Euler

Project the whole vector field

Project the whole vector field $F := -\nabla_{\boldsymbol{W}} \mathrm{loss}$ onto the tangent space $T_{\boldsymbol{W}(t)}\mathcal{M}$ of $\mathcal{M}$ at the current point $\boldsymbol{W}(t)$

$$\frac{d}{dt}\boldsymbol{W}(t) = -P_{T_{\boldsymbol{W}(t)}\mathcal{M}} \nabla_{\boldsymbol{W}} \mathrm{loss}(\boldsymbol{W}(t); x, y)$$



Impose Galerkin and Gauge conditions

For each $W_i \in \{W_1, \ldots, W_\ell\}$, assume $W_i(t) = U_i(t)S_i(t)V_i(t)^\top$, where $U_i(t), V_i(t)$ are tall and skinny matrices and $S_i(t)$ is a small square invertible matrix (not necessarily diagonal!).

Imposing Galerkin and Gauge conditions on the factorization, we can rewrite the projected ODE as a system of ODEs for each of the factors $U_i(t)$, $V_i(t)$, $S_i(t)$.

Galerkin:

$\langle \dot{W}_i(t) + \nabla_{W_i} \mathrm{loss}(W_i(t)), \delta W_i(t) \rangle = 0$ for all $\delta W_i(t) \in T_{W_i(t)}\mathcal{M}_i$

Gauge:

$U_i(t)^\top \delta U_i(t) = V_i(t)^\top \delta V_i(t) = 0$ for all $i = 1, \ldots, \ell$

System of low-rank ODEs

For each $W \in \{W_1, \ldots, W_\ell\}$,

$$\begin{cases} \dot{S} = -U^\top \nabla_W \mathrm{loss}(W) V \\ \dot{U} = -(I - UU^\top) \nabla_W \mathrm{loss}(W) V S^{-1} \\ \dot{V} = -(I - VV^\top) \nabla_W \mathrm{loss}(W)^\top U S^{-\top} \end{cases}$$

System of low-rank ODEs

For each $W \in \{W_1, \ldots, W_\ell\}$,

$$\begin{cases} \dot{S} = -U^\top \nabla_W \mathrm{loss}(W) V \\ \dot{U} = -(I - UU^\top) \nabla_W \mathrm{loss}(W) V S^{-1} \\ \dot{V} = -(I - VV^\top) \nabla_W \mathrm{loss}(W)^\top U S^{-\top} \end{cases}$$

These ODEs reflect the local curvature of the low-rank manifold, which is proportional to the inverse of the smallest singular value of S, and thus are unstable when the singular values of S are small.

Moreover, they still require the full gradient ∇_W, which is computationally inefficient

Breaking "the curse of the curvature": KLS parametrization

The simple change of variable

$$K(t) = U(t)S(t) \qquad L(t) = V(t)S(t)^\top$$

leads to

$$\begin{cases} \dot{K} = -\nabla_W \text{loss}(KV^\top)V \\ \dot{L} = -\nabla_W \text{loss}(UL^\top)^\top U \\ \dot{S} = -U^\top \nabla_W \text{loss}(USV^\top)V \end{cases}$$

which requires no matrix inversions and ....

Breaking "the curse of the curvature": KLS parametrization

The simple change of variable

$$K(t) = U(t)S(t) \qquad L(t) = V(t)S(t)^\top$$

leads to

$$\begin{cases} \dot{K} = -\nabla_W \text{loss}(KV^\top)V = -\nabla_K \text{loss}(KV^\top) \\ \dot{L} = -\nabla_W \text{loss}(UL^\top)^\top U = -\nabla_L \text{loss}(UL) \\ \dot{S} = -U^\top \nabla_W \text{loss}(USV^\top)V = -\nabla_S \text{loss}(USV^\top) \end{cases}$$

which requires no matrix inversions and .... does not need to compute the full gradient allowing us to operate only with the small matrices K, L, S still having access to U, S, V

Rank-adaptive DLRT Algorithm

Step1 Update the current $K^{(n)} = U^{(n)} S^{(n)}$ and $L^{(n)} = V^{(n)} S^{(n)\top}$ by integrating from $t = t_0$ to $t = t_1$ (in parallel) the ODEs

$$\begin{cases} \dot{K} = -\nabla_K \text{loss}(KV^\top); & K(t_0) = K^{(n)} \\ \dot{L} = -\nabla_L \text{loss}(UL^\top)^\top U; & L(t_0) = L^{(n)} \end{cases}$$

Rank-adaptive DLRT Algorithm

Step1 Update the current $K^{(n)} = U^{(n)}S^{(n)}$ and $L^{(n)} = V^{(n)}S^{(n)\top}$ by integrating from $t = t_0$ to $t = t_1$ (in parallel) the ODEs

$$\begin{cases} \dot{K} = -\nabla_K \text{loss}(KV^\top); & K(t_0) = K^{(n)} \\ \dot{L} = -\nabla_L \text{loss}(UL^\top)^\top U; & L(t_0) = L^{(n)} \end{cases}$$

Step2 Form augmented basis $\widetilde{U} \leftrightarrow [K(t_1)|U^{(n)}]$ and $\widetilde{V} \leftrightarrow [L(t_1)|V^{(n)}]$

Rank-adaptive DLRT Algorithm

Step1 Update the current $K^{(n)} = U^{(n)}S^{(n)}$ and $L^{(n)} = V^{(n)}S^{(n)\top}$ by integrating from $t = t_0$ to $t = t_1$ (in parallel) the ODEs

$$\begin{cases} \dot{K} = -\nabla_K \text{loss}(KV^\top); & K(t_0) = K^{(n)} \\ \dot{L} = -\nabla_L \text{loss}(UL^\top)^\top U; & L(t_0) = L^{(n)} \end{cases}$$

Step2 Form augmented basis $\widetilde{U} \leftrightarrow [K(t_1)|U^{(n)}]$ and $\widetilde{V} \leftrightarrow [L(t_1)|V^{(n)}]$

Step3 Lift the current $S^{(n)}$ to $\widetilde{S} = \widetilde{U}^\top U^{(n)} S^{(n)} V^{(n)\top} \widetilde{V}$

Rank-adaptive DLRT Algorithm

Step1 Update the current $K^{(n)} = U^{(n)}S^{(n)}$ and $L^{(n)} = V^{(n)}S^{(n)\top}$ by integrating from $t = t_0$ to $t = t_1$ (in parallel) the ODEs

$$\begin{cases} \dot{K} = -\nabla_K \text{loss}(KV^\top); & K(t_0) = K^{(n)} \\ \dot{L} = -\nabla_L \text{loss}(UL^\top)^\top U; & L(t_0) = L^{(n)} \end{cases}$$

Step2 Form augmented basis $\widetilde{U} \leftrightarrow [K(t_1)|U^{(n)}]$ and $\widetilde{V} \leftrightarrow [L(t_1)|V^{(n)}]$

Step3 Lift the current $S^{(n)}$ to $\widetilde{S} = \widetilde{U}^\top U^{(n)} S^{(n)} V^{(n)\top} \widetilde{V}$

Step4 Update the current $\widetilde{S}$ by integrating from t_0 to t_1 the ODE

$$\dot{S} = -\nabla_S \text{loss}(\widetilde{U}S\widetilde{V}^\top); \quad S(t_0) = \widetilde{S}$$

Step5 Form $U^{(n+1)}, V^{(n+1)}, S^{(n+1)}$ by truncating the singular values of $S(t_1)$ using a threshold ϑ =compression rate, and using the corresponding sing vecs

Advantages of the gradient flow formulation

Our approach can be interpreted as a form of Riemaniann Optimization (RO) scheme with a special retractor. Using the continuous formulation as a number of advantages

1. The resulting scheme is computationally cheap and well-conditioned. It allows us to use any numerical integrator
2. It allows us to perform a rank adjustment step in a simple way, maintaining descent guarantees (while changing the rank of the manifold is a well-known tough challenge in RO)
3. It allows us to prove guarantees of approximation of the full network *(proof of low-rank lottery ticker hypothesis)*

Main theorem (informal)

$W^{(n)} = U^{(n)} S^{(n)} V^{(n),\top}$ computed low-rank flow; $W(t)$ exact flow

Suppose that

1. $\nabla_{\boldsymbol{W}} \mathrm{loss}$ is locally bounded and locally Lipschitz continuous
2. $\boldsymbol{W}(t_0)$ and $\nabla_{\boldsymbol{W}} \mathrm{loss}(\boldsymbol{W}(t_0))$ are "ε-close" to $\mathcal{M}$
3. the integration step $h = t_1 - t_0$ is "small enough"

Main theorem (informal)

$W^{(n)} = U^{(n)} S^{(n)} V^{(n),\top}$ computed low-rank flow; $W(t)$ exact flow

Suppose that

1. $\nabla_{\boldsymbol{W}} \mathrm{loss}$ is locally bounded and locally Lipschitz continuous
2. $\boldsymbol{W}(t_0)$ and $\nabla_{\boldsymbol{W}} \mathrm{loss}(\boldsymbol{W}(t_0))$ are "ε-close" to $\mathcal{M}$
3. the integration step $h = t_1 - t_0$ is "small enough"

then

- $\|\boldsymbol{W}^{(n)} - \boldsymbol{W}(nh)\| \leq c_1 \varepsilon + c_2 h + c_3 \vartheta / h$

where c_i are positive constants that do not depend on the singular values of $\boldsymbol{W}^{(n)}$ nor $\boldsymbol{W}(nh)$

Main theorem (informal)

$W^{(n)} = U^{(n)} S^{(n)} V^{(n),\top}$ computed low-rank flow; $W(t)$ exact flow

Suppose that

1. $\nabla_{\boldsymbol{W}} \mathrm{loss}$ is locally bounded and locally Lipschitz continuous
2. $\boldsymbol{W}(t_0)$ and $\nabla_{\boldsymbol{W}} \mathrm{loss}(\boldsymbol{W}(t_0))$ are "ε-close" to $\mathcal{M}$
3. the integration step $h = t_1 - t_0$ is "small enough"

then

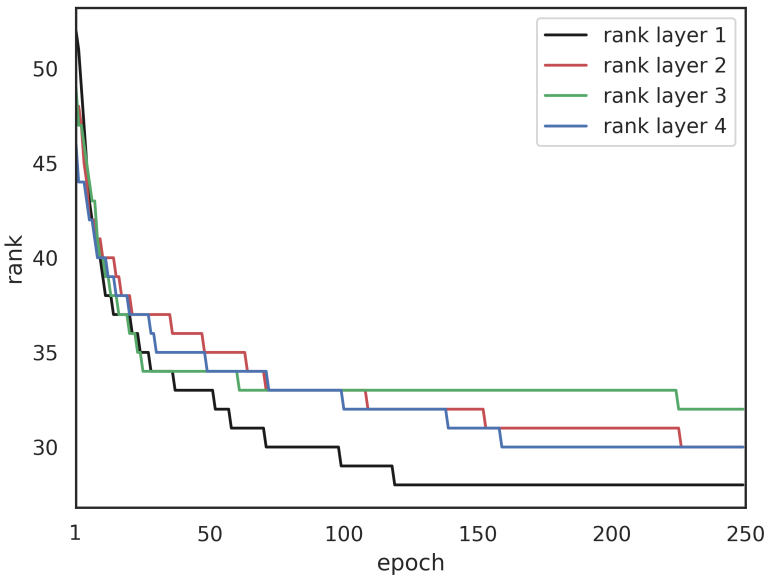
- $\|\boldsymbol{W}^{(n)} - \boldsymbol{W}(nh)\| \leq c_1\varepsilon + c_2 h + c_3\vartheta/h$
- $\mathrm{loss}(\boldsymbol{W}^{(n+1)}; x, y) \leq \mathrm{loss}(\boldsymbol{W}^{(n)}; x, y) - c_4 h + c_5\vartheta$

where c_i are positive constants that do not depend on the singular values of $\boldsymbol{W}^{(n)}$ nor $\boldsymbol{W}(nh)$

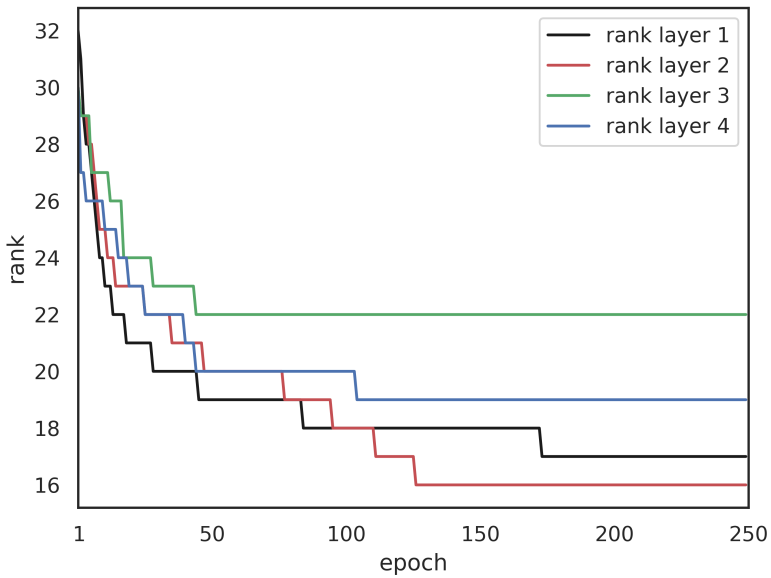
Fully-connected on MNIST: Rank evolution

5-layer NN, $N_i = 500$ for $i = 1, 2, 3, 4$;

Rank evolution for different compression rates



$\vartheta = 0.09$

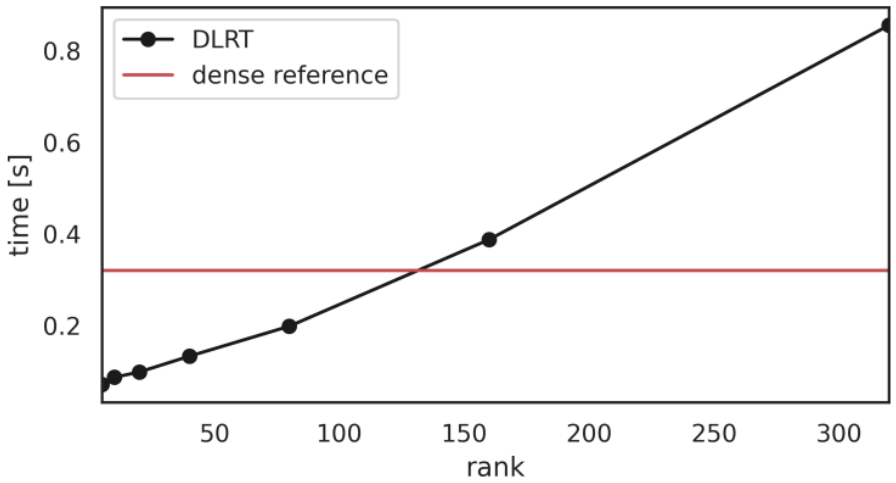


$\vartheta = 0.13$

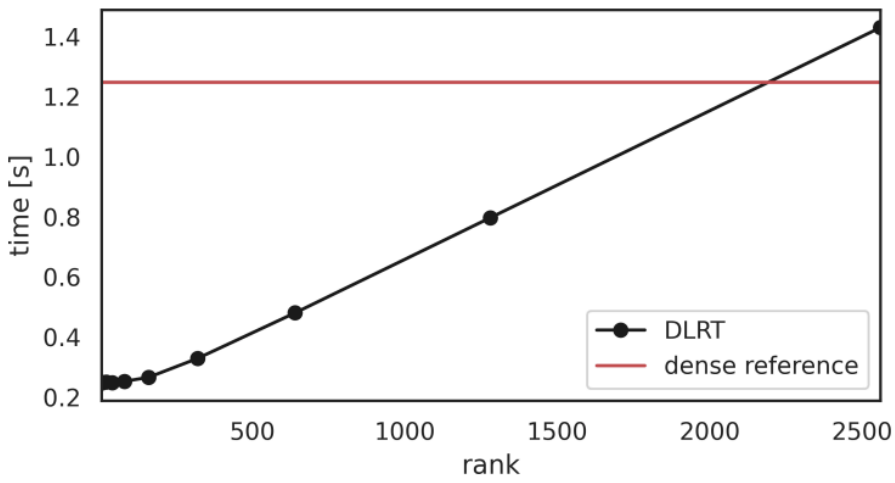
Compression: keep only $\left(\vartheta \sum_j \text{singval}_j(W_i)\right)$ sing values

Fully-connected on MNIST: Time and accuracy

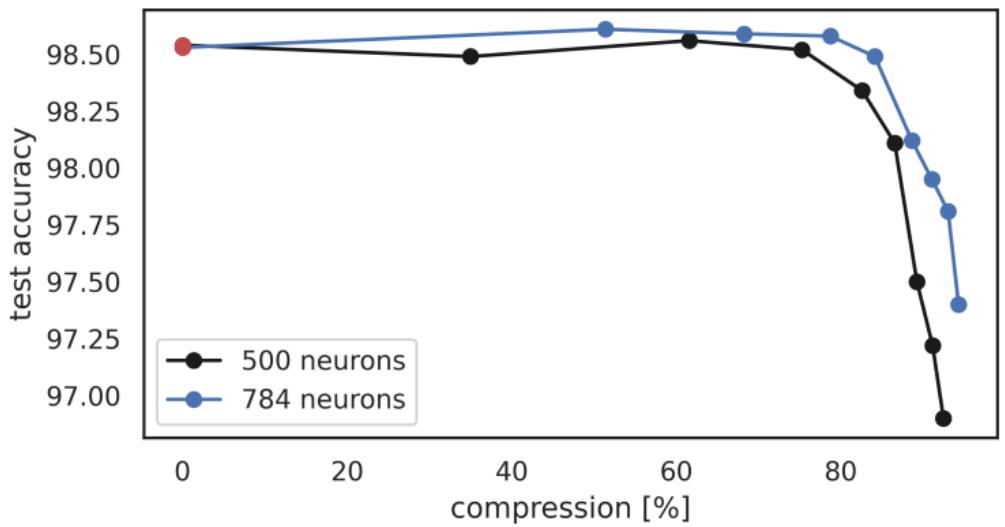
5-layer NN, $N_i = 5k$ for $i = 1, 2, 3, 4$; Average over $1k$ runs



(a) Training timings



(b) Prediction timings



Convolutional NN (LeNet5) on MNIST

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

Imagenet1K

Results of adaptive DLRT on Imagenet1K:
$\approx$1 million 256×256 images; $\approx$1000 classes

Model	test acc. [%]	c.r. eval [%]	c.r. train [%]
ResNet-50	-0.56	54.1	14.2
VGG16	-2.19	86	78.4

Test accuracy difference to full-rank baseline and compression rate for ResNet-50 and VGG16 with $\vartheta = 0.1$.

DLRT vs vanilla low-rank

Suppose I do not use the KLS strategy and perform a "vanilla" fixed-rank approach by setting $W = UV^\top$ and minimizing

$$\min_{U,V} \text{loss}(UV^\top; x, y)$$

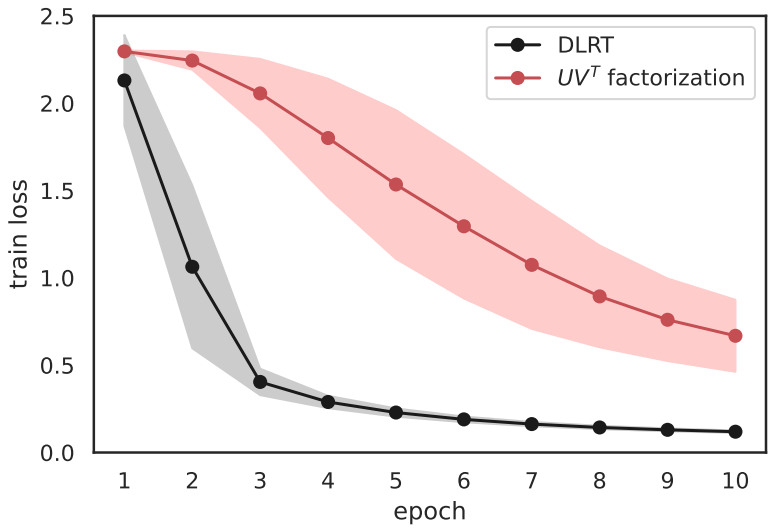
with alternate gradient descent over U and V

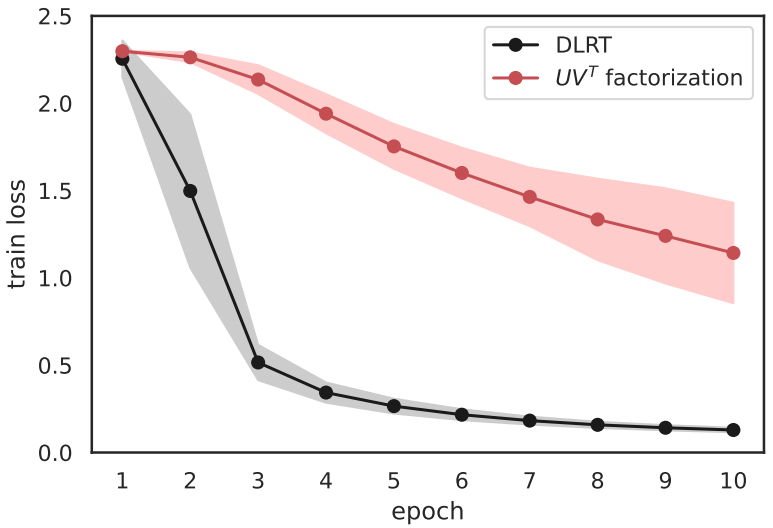
DLRT vs vanilla low-rank

Suppose I do not use the KLS strategy and perform a "vanilla" fixed-rank approach by setting $W = UV^\top$ and minimizing

$$\min_{U,V} \text{loss}(UV^\top; x, y)$$

with alternate gradient descent over U and V





Conclusion

- Reducing memory storage requirements as well as computational cost of deep learning pipelines is very important both in the **inference** and in the **training phase**
- Using low-rank matrix manifold (and in general, structured matrices) is a very attractive strategy
- The proposed Dynamic Low-Rank Training Algorithm reduces training and inference costs and memory footprint, maintaining high accuracy and fast loss descent

S. Schotthöfer, E. Zangrando, J. Kusch, G. Ceruti, F.T. *Low-rank lottery tickets: finding efficient low-rank neural networks via matrix differential equations*, NeurIPS(2022).

Conclusion

- Reducing memory storage requirements as well as computational cost of deep learning pipelines is very important both in the **inference** and in the **training phase**
- Using low-rank matrix manifold (and in general, structured matrices) is a very attractive strategy
- The proposed Dynamic Low-Rank Training Algorithm reduces training and inference costs and memory footprint, maintaining high accuracy and fast loss descent

S. Schotthöfer, E. Zangrando, J. Kusch, G. Ceruti, F.T. *Low-rank lottery tickets: finding efficient low-rank neural networks via matrix differential equations*, NeurIPS(2022).

Thank you!