# Tensor-GaLore: Memory-Efficient Training via Gradient Tensor Decomposition

**Robert Joseph George**[1] RGEORGE@CALTECH.EDU
**David Pitt**[1] DPITT@CALTECH.EDU
**Jiawei Zhao**[2] JIAWEI@CALTECH.EDU
**Jean Kossaifi**[3] JKOSSAIFI@NVIDIA.COM
**Cheng Luo**[1] CHENGLUO@CALTECH.EDU
**Yuandong Tian**[2] YUANDONG@META.COM
**Anima Anandkumar**[1] ANIMA@CALTECH.EDU
[1]*California Institute of Technology,* [2]*Meta FAIR,* [3]*NVIDIA AI*

## Abstract

We present **Tensor-GaLore**, a novel method for efficient training of neural networks with higher-order tensor weights. Many models, particularly those used in scientific computing and computer vision, employ tensor-parameterized layers to capture complex, high-dimensional relationships. However, these tensor structures lead to significant memory requirements during training. Our method addresses this memory challenge through low-rank subspace optimization using Tucker decomposition, overcoming limitations of previous approaches restricted to matrix-parameterized weights, including those operating on complex-valued data. We showcase its effectiveness on Fourier Neural Operators (FNOs), a class of models crucial for solving partial differential equations. Across various PDE tasks, we achieved performance gains ranging from 11% to 50% better generalization while reducing optimizer memory usage by up to 76%. These consistent improvements, coupled with substantial memory savings across AI for science, demonstrate Tensor-GaLore's potential.

## 1. Introduction

Foundation models have revolutionized AI, demonstrating unprecedented performance across diverse domains [1, 5]. However, their immense scale presents significant computational challenges, particularly in terms of memory requirements for training and deployment. While recent work has focused on parameter-efficient fine-tuning methods for two-dimensional weight structures, many models, especially those in scientific computing, are built on higher-dimensional *tensor* weight structures. Tensors, as multidimensional arrays, offer a natural framework for representing and manipulating complex, high-dimensional data structures with applications in computer vision, signal processing, and scientific computing. Current memory-efficient training methods often fail to address the tensor nature of these models, either ignoring higher-dimensional structures or attempting to reshape them into matrices, potentially losing important spatial and dimensional relationships.

In particular, scientific computing has seen a significant paradigm shift towards applying AI to classical problems. The neural operator, specifically the FNO [10], is one of the most promising new architectures in this domain. FNOs are a class of neural network architecture designed to learn mappings between function spaces to solve parametric partial differential equations (PDEs),

© R.J. George[1], D. Pitt[1], J. Zhao[2], J. Kossaifi[3], C. Luo[1], Y. Tian[2] & A. Anandkumar[1].

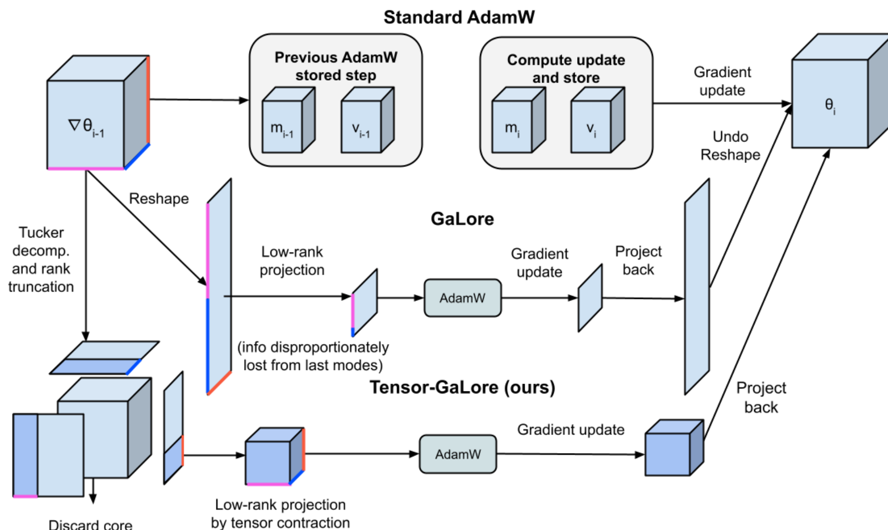


Figure 1: Comparison of our proposed Tensor-GaLore algorithm with standard AdamW and GaLore. GaLore applies matrix-based low-rank projection after reshaping tensors. Our Tensor-GaLore method leverages tensor decomposition to perform low-rank projection directly on tensor gradients, preserving multidimensional structure..

a cornerstone of modern scientific computing. Unlike traditional neural networks, FNOs involve high-dimensional tensor operations. For our case, in an FNO the spectral convolution layer employs a weight tensor $\mathcal{W} \in \mathbb{C}^{N_1 \times N_2 \times N_3 \times N_4}$ to perform operations in the Fourier domain: $(\mathcal{K}v_l)(x) = \mathcal{F}^{-1}(R \cdot T_K \mathcal{F} v_l)(x)$, where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the Fourier transform and its inverse, $R$ is a learnable transformation, and $T_K$ truncates to the lowest $K$ Fourier modes. While these tensor operations are powerful for capturing complex, high-dimensional relationships in scientific data, they pose unique challenges related to memory consumption during training. The primary issue lies not in the activation memory induced by forward and backward passes, but in the memory overhead required for optimization. This is due to the need to store the Fourier coefficients and perform operations in the frequency domain [12]. This memory bottleneck is exacerbated by modern optimizers, which often store multiple tensors for each weight tensor to track gradients, momentum, and other quantities, as in the case of Adam. Consequently, the optimizer state comprises a significant portion of the memory overhead in training large-scale NOs.

To address the memory requirements of optimization, recent work has shown that these stored gradients often exhibit low-rank structures during training, suggesting that the most important gradient information can be preserved at a fraction of the memory cost. GaLore (Gradient Low-Rank Projection) [17] leveraged this insight to reduce memory usage in large language model training by projecting gradients onto a low-rank subspace and performing optimization on the resultant low-rank gradients. The key steps of GaLore are mentioned in Appendix C. However, GaLore's approach was limited to matrix operations which is not directly applicable to the tensor gradients

encountered in NOs. In particular, the limitations of matrix-based approaches like GaLore when applied to tensor operations (also in Appendix E) are twofold:

1. **Tensor Structure**: NOs often involve inherently tensor-structured gradients. Flattening these tensors into matrices loses the multi-dimensional relationships crucial for capturing complex physical phenomena.

2. **Dimension-specific Information**: Simply "rolling up" tensor dimensions into matrix form doesn't guarantee convergence and can lead to losing important dimension-specific information. For instance, in an FNO, different dimensions might correspond to spatial, temporal, or channel information, each requiring distinct treatment.

These challenges motivate the need for a tensor-specific approach to gradient projection and optimization. In this paper, we introduce Tensor-GaLore, a novel approach that extends the principles of gradient low-rank projection to tensor operations. Our contributions are summarized as follows:

1. We introduce **Tensor-GaLore, a novel method for efficiently training NOs through low-rank gradient projections**. To the best of our knowledge, Tensor-GaLore is the first work to explore low-rank subspace learning for gradients of higher-order tensors that seeks low-rank representation while offering a significant advancement in memory-efficient optimization and topologically preserving the structure.

2. We demonstrate the effectiveness of Tensor-GaLore on various PDE tasks, showing significant reductions in memory usage (upto 30%) while improving model performance (up to 50%).

## 2. Tensor GaLore

We mention the details of tucker decomposition in Appendix D. In particular, to extend GaLore to methods with learned tensor weights, we replace the matrix-based SVD with tensor decomposition methods. This extension, called Tensor-GaLore, allows us to handle multi-dimensional data and complex network architectures more efficiently. For a gradient tensor $\mathcal{G} \in \mathbb{C}^{I_1 \times I_2 \times \cdots \times I_N}$, the Tucker-based Tensor-GaLore as shown in Algorithm F performs the following steps:

1. Compute the Tucker decomposition of the gradient tensor:

$$\mathcal{G} \approx \mathcal{C} \times_1 U^{(1)} \times_2 U^{(2)} \cdots \times_N U^{(N)} = [\![\mathcal{C}; U^{(1)}, U^{(2)}, \ldots, U^{(N)}]\!] \tag{1}$$

   where $\mathcal{C} \in \mathbb{C}^{R_1 \times R_2 \times \cdots \times R_N}$ is the core tensor and $U^{(n)} \in \mathbb{C}^{I_n \times R_n}$ are factor matrices.

2. Project the gradient tensor onto the low-rank subspace and update the optimizer states and model parameters using the projected gradient $\mathcal{G}_{\text{proj}}$.

$$\mathcal{G}_{\text{proj}} = [\![G_{\text{core}} U^{(1)}, U^{(2)}, \ldots, U^{(N)}]\!] \tag{2}$$

3. Project the gradient back when updating.

$$\mathcal{G}_{\text{core}} = [\![G_{\text{proj}} {U^{(1)}}^T, {U^{(2)}}^T, \ldots, {U^{(N)}}^T]\!] \tag{3}$$

### 2.1. Implicit Regularization

Tucker decomposition also allows for a separate rank along each mode of the tensor, meaning that all key information can be preserved explicitly. Additionally, the factors learned in iterative Tucker decomposition can be initialized to non-random factors, meaning that as learning progresses, results from a previous decomposition can be used to 'warm-restart' the decomposition, leading to convergence in fewer iterations. The low-rank tensor approximation acts as an implicit regularizer, helping to prevent overfitting and promoting smoother optimization trajectories and hence why we observe much better convergence (generalization) in our experiments. In particular we consistently observed that a rank of around 25% of the full rank provided optimal performance across various tasks. This sweet spot suggests that Tensor-GaLore is acting as an implicit regularizer, preventing overfitting by constraining the model to learn more robust, low-rank representations of the underlying physics. This aligns with findings from [14], which demonstrated that tensor factorization naturally tends towards low-rank solutions.

## 3. Experimental Setup

We conduct a comprehensive evaluation of GaLore and Tensor-GaLore on a diverse set of benchmark datasets for NOs. We select 3 datasets representing a range of PDEs with varying complexity and dimensionality. In particular they are the Burger's, Darcy and ElectroMagnetic Wave propagation dataset. Their details are in Appendix G.

### 3.1. Model Architecture and Training

We implement Tensor-GaLore with the FNO architecture. Models are trained using the Adam optimizer with a learning rate of $10^{-3}$ and weight decay of $10^{-4}$. Other hyperparameters, such as batch size and number of epochs, are detailed in the Appendix for each dataset and model configuration K. For Tensor-GaLore, we investigate the impact of varying the rank of the decomposition's. We explore ranks ranging from 1% to 100% of the full rank, allowing us to assess the trade-off between model compression and performance. Similarly, for the GaLore implementation, we conduct experiments with different ranks. Additionally, for tensor inputs, we explore various ways of reshaping the tensor to a matrix before applying GaLore. Specifically, we examine each possible "rollout" dimension, where we flatten all dimensions except one into a single dimension. This allows us to compare the effectiveness of different tensor-to-matrix projections.

## 4. Results

Our experiments demonstrate the effectiveness of Tensor-GaLore across various datasets, showing significant improvements in both performance and memory efficiency as shown in Table 1. For the Burgers' equation, our method consistently outperformed the baseline FNO, with performance improving as rank increased. On the Darcy flow problem, Tensor-GaLore achieved up to a 50% gain in test loss at rank 0.25, while reducing optimizer memory by 76% as shown in Appendix I. Electromagnetic wave propagation simulations saw up to 11% gains. The results show that Tensor-GaLore can significantly reduce the memory footprint of the optimizer states while improving model performance in many cases.

Table 1: Evaluating Tensor-GaLore across various tasks.

| Model | Rank Ratio | Memory (GB) | Train (Loss ($\times 10^{-2}$)) | Test $H_1$ (Loss ($\times 10^{-2}$)) | Test $L_2$ (Loss ($\times 10^{-2}$)) | Gain (%) |
|---|---|---|---|---|---|---|
| **Darcy** | | | | | | |
| Baseline | 1.0 | 8.88 | 0.7151 | 1.6230 | 0.2050 | / |
| GaLore (d=2) | 0.25 | 7.34 | 0.4200 | 1.3210 | 0.1680 | 19 |
| Tensor-GaLore | 0.25 | 7.32 | **0.2930** | **0.8680** | **0.1050** | **48.8** |
| **ElectroMagnetic** | | | | | | |
| Baseline | 1.0 | 4.83 | 2.973 | 0.1902 | 0.2000 | / |
| GaLore (d=2) | 0.25 | 4.83 | 2.392 | 0.1802 | 0.1900 | 5 |
| Tensor-GaLore | 0.25 | 4.63 | **2.132** | **0.1681** | **0.1782** | **11** |
| **Burgers (1e-4)** | | | | | | |
| Baseline | 1.0 | 3.94 | 0.2052 | 0.0050 | 0.0026 | / |
| GaLore (d=2) | 0.5 | 3.88 | 0.5053 | 0.0100 | 0.0062 | -250 |
| Tensor-GaLore | 0.5 | 3.87 | **0.0860** | **0.0041** | **0.0025** | **+5** |

## 5. Applications

Tensor-GaLore has potential applications across various domains where tensor-based models are prevalent. In the field of large language models (LLMs), it could enable training of tensor-based architectures that capture higher-order relationships in language data, offering improved memory efficiency and implicit regularization while preserving the natural tensor structure. In vision, Convolutional Neural Networks (CNNs) also heavily utilize higher-order tensor weights. CNN convolution layers include 4-dimensional tensor weights. As discussed previously, these weight gradients and optimizer states have high memory requirements, making memory consumption a significant bottleneck in training deep CNNs [16].

## 6. Conclusion

Our experiments with Tensor-GaLore reveal significant insights into its performance and applications. The method consistently improves convergence across datasets, creating a more stable optimization landscape that facilitates faster convergence to better solutions, as evidenced in the Darcy flow, Burgers and EM experiments as well as reduced memory usage by a huge margin especially the optimizer state. However, challenges remain, including the overhead of tensor decomposition and optimal rank selection. Future work should focus on automating rank selection, expanding applications to a broader range of scientific computing tasks, and exploring potential in other domains as well as testing on larger scale PDE Datasets like the Navier Stokes equations with high Reynolds numbers. We expect to see much better performance gains there. Tensor-GaLore opens up new avenues for building and scaling foundational models in scientific computing, potentially leading to more accurate and computationally efficient models for critical applications like climate prediction and fluid dynamics.

## References


[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] Robert Joseph George, Jiawei Zhao, Jean Kossaifi, Zongyi Li, and Anima Anandkumar. Incremental spatial and spectral learning of neural operators for solving large-scale pdes, 2024. URL https://arxiv.org/abs/2211.15188.

[3] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2022.

[4] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *International Conference on Learning Representations*, 2016.

[5] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[6] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. doi: 10.1137/07070111X. URL https://doi.org/10.1137/07070111X.

[7] Jean Kossaifi, Nikola Kovachki, Kamyar Azizzadenesheli, and Anima Anandkumar. Multi-grid tensorized fourier neural operator for high-resolution pdes. *arXiv preprint arXiv:2403.00071*, 2024.

[8] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

[9] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *International Conference on Learning Representations*, 2015.

[10] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[11] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2021.

[12] Levi Lingsch, Mike Y. Michelis, Emmanuel de Bezenac, Sirani M. Perera, Robert K. Katzschmann, and Siddhartha Mishra. Beyond regular grids: Fourier-based neural operators on arbitrary domains, 2024. URL https://arxiv.org/abs/2305.19663.

[13] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. *arXiv preprint arXiv:1509.06569*, 2015.

[14] Noam Razin, Asaf Maman, and Nadav Cohen. Implicit regularization in hierarchical tensor factorization and deep convolutional neural networks, 2022. URL https://arxiv.org/abs/2201.11729.

[15] Chongjie Si, Xuehui Wang, Xue Yang, Zhengqin Xu, Qingyun Li, Jifeng Dai, Yu Qiao, Xiaokang Yang, and Wei Shen. Flora: Low-rank core space for n-dimension, 2024. URL https://arxiv.org/abs/2405.14739.

[16] Muhammad Yaqub, Jinchao Feng, M. Sultan Zia, Kaleem Arshid, Kebin Jia, Zaka Ur Rehman, and Atif Mehmood. State-of-the-art cnn optimizer for brain tumor segmentation in magnetic resonance images. *Brain Sciences*, 10(7), 2020. ISSN 2076-3425. doi: 10.3390/brainsci10070427. URL https://www.mdpi.com/2076-3425/10/7/427.

[17] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

## Appendix A. FNO Memory Usage

As illustrated in Figure 2, the memory consumption for activations (shown in dark green) remains relatively constant and low across different numbers of frequency modes in FNOs. However, the memory usage for individual components, including gradients and optimizer states (shown in yellow), grows significantly as the number of modes increases.

## Appendix B. Related Work

Our work, Tensor-GaLore, introduces a novel approach to efficiently training neural operators by decomposing gradients. While significant work has been done in related areas, the specific approach of gradient decomposition in tensors has not been explored before.

**Tensor Methods in Deep Learning:** Tensor decomposition has been widely used to compress and improve deep networks, particularly in vision tasks [4, 9, 13]. These methods typically focus on decomposing the weight tensors of the network to reduce parameters and computational complexity. However, they do not address the decomposition of gradients during training.

**Neural Operators:** Recent advancements in learning-based approaches for solving PDEs have led to the development of neural operators [8, 10]. Fourier Neural Operators (FNOs) in particular have shown remarkable success in various scientific computing tasks [11]. While these methods have made significant strides in learning solution operators for PDEs, they have not explored gradient decomposition as a means of improving memory efficiency.

**Efficient Training Techniques:** Various approaches have been proposed to reduce the memory footprint of large-scale models. In the classical case, when model weights are stored as matrices,

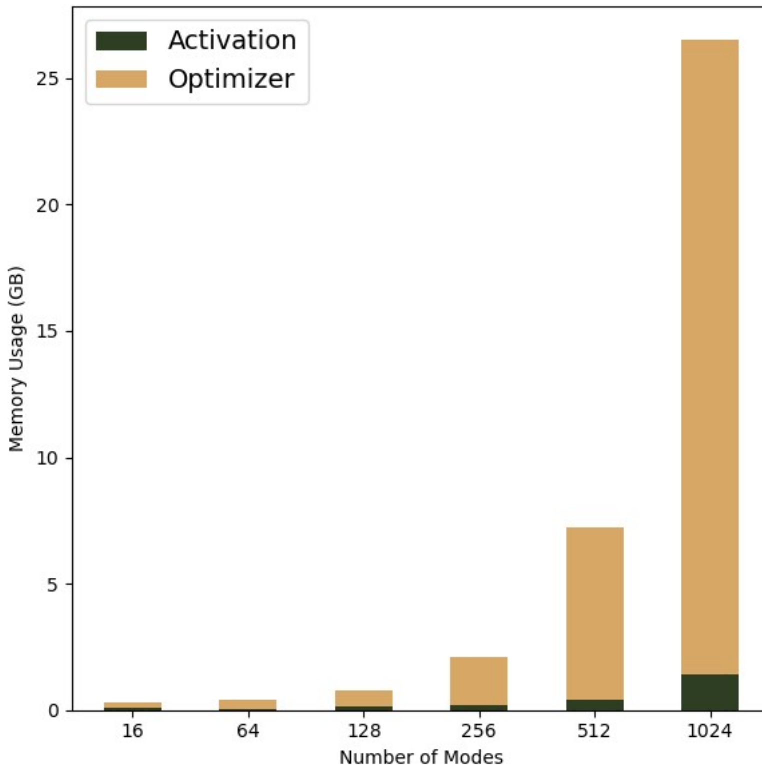


Figure 2: Memory usage in FNO

several techniques have demonstrated success. LoRA [3] adds a fine-tuning weight matrix created via a low-rank decomposition to an original pre-trained, frozen weight matrix. To drive further memory savings, GaLore [17] leverages the insight that a significant portion of memory usage resides in the optmizer state. GaLore projects weight matrices to a low-rank subspace and optimizes weights directly in that subspace, which brings further memory savings in the optimizer state.

In the higher-order case, FLoRA [15] extends the idea of low-rank adaptation to higher-dimensional parameter spaces using a Tucker tensor decomposition, which has the demonstrated benefit of applying a low-rank decomposition to each dimension of a higher-order space.

In the context of neural operators, which include higher-order tensorized weights, previous works have demonstrated the possibility of model compression via tensor factorization and low-rank weight approximations. Kossaifi et al. [7] introduced the Multi-Grid Tensorized Fourier Neural Operator (MG-TFNO), which combines tensor decomposition with a multi-grid domain decomposition approach. However, these methods focus on model compression rather than gradient decomposition during training. In order to balance low-rank memory optimization with model performance at higher ranks, the Incremental Fourier Neural Operator [2] incrementally scales both the size and rank of FNO weights during training in order to boost performance.

## Appendix C. Background

### C.1. Neural Operator

A neural operator $\mathcal{G}_\theta : \mathcal{A} \times \theta \to \mathcal{U}$ combines linear integral operators $\mathcal{K}$ with pointwise non-linear activations $\sigma$ to approximate non-linear operators, mapping initial conditions $a \in \mathcal{A}$ to solutions $u \in \mathcal{U}$. It is defined as $\mathcal{G}_\theta := \mathcal{Q} \circ (W_L + \mathcal{K}_L) \circ \cdots \circ \sigma(W_1 + \mathcal{K}_1) \circ \mathcal{P}$, where $\mathcal{P}$ and $\mathcal{Q}$ are pointwise neural networks for encoding and decoding, $W_l$ are linear operators, $\mathcal{K}_l$ are integral kernel operators, and $\sigma$ are activation functions.

The FNO proposes a specific convolution operator for $\mathcal{K}$, defined as $(\mathcal{K}v_l)(x) = \mathcal{F}^{-1}(R \cdot T_K \mathcal{F} v_l)(x)$, where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the Fourier transform and its inverse, $R$ is a learnable transformation, and $T_K$ truncates to the lowest $K$ Fourier modes. This formulation allows FNO to be discretization-invariant, producing high-quality solutions for query points not in the training grid and enabling transfer between different grid resolutions and discretizations.

### C.2. Gradient Low-Rank Projection

GaLore was introduced as a memory-efficient training strategy for Large Language Models (LLMs). GaLore leverages the observation that gradients in deep neural networks often exhibit low-rank structures during training. Instead of storing and updating full-rank gradients, GaLore projects gradients onto low-rank subspace, significantly reducing memory requirements while maintaining model performance. In its original formulation, GaLore operates on weight matrices $W \in \mathbb{R}^{m \times n}$ and their corresponding gradient matrices $G \in \mathbb{R}^{m \times n}$. The key steps of GaLore are as follows:

1. Compute the Singular Value Decomposition (SVD) of the gradient matrix:

$$G = USV^T \approx \sum_{i=1}^{r} s_i u_i v_i^T \tag{4}$$

   where $U$ and $V$ contain the left and right singular vectors, $S$ is the diagonal matrix of singular values, and $r$ is the chosen rank.

2. Form projection matrices using the top $r$ singular vectors:

$$P = [u_1, \ldots, u_r] \in \mathbb{R}^{m \times r}, \quad Q = [v_1, \ldots, v_r] \in \mathbb{R}^{n \times r} \tag{5}$$

3. Project the gradient onto the low-rank subspace:

$$G_{\text{proj}} = P G_{\text{core}} Q^T, \quad \text{where} \quad G_{\text{core}} = P^T G Q \in \mathbb{R}^{r \times r} \tag{6}$$

4. Update the optimizer states and model parameters using the projected gradient $G_{\text{proj}}$.

This approach allows GaLore to maintain a low memory footprint by storing and updating only the low-rank representations of gradients.

### C.3. SVD

1. **Equivalence to SVD in 2D**: In the special case of 2D tensors (matrices), the Tucker decomposition reduces to the familiar SVD. The core tensor $\mathcal{G}$ becomes equivalent to the diagonal matrix $\Sigma$ in SVD, while the factor matrices correspond to the orthogonal matrices $U$ and $V$ [6]. This property ensures that our method seamlessly extends the principles of matrix-based techniques to higher-order tensors.

2. **Orthogonality of factor matrices**: The factor matrices $U^{(n)}$ in Tucker decomposition are orthogonal, mirroring the properties of $U$ and $V$ in SVD. This orthogonality is crucial for the efficiency and stability of the GaLore method. Specifically:

   (a) *Projection efficiency*: The orthogonality allows us to project tensors onto the subspace spanned by these matrices through simple matrix multiplication, without the need for costly inverse computations.

   (b) *Easy inversion*: When we need to reverse the projection, we can simply use the transpose of these orthogonal matrices instead of computing their inverses. This property is expressed mathematically as $(U^{(n)})^T U^{(n)} = I$, where $I$ is the identity matrix.

   (c) *Numerical stability*: Orthogonal matrices have a condition number of 1, ensuring that the projection and its inverse are numerically stable operations, even for high-dimensional tensors.

## Appendix D. Tucker Decomposition

Tensors are multidimensional arrays that generalize the concepts of vectors (first-order tensors) and matrices (second-order tensors) to higher orders. An $N$th-order tensor $\mathcal{X} \in \mathbb{C}^{I_1 \times I_2 \times \cdots \times I_N}$ is an $N$-way array where each mode $n$ has dimension $I_n$. Like matrices, in tensors we can decompose the tensors into low-rank factors using the Tucker decomposition, also known as the higher-order SVD (HOSVD), that decomposes a tensor into a core tensor multiplied by a matrix along each mode:

$$\mathcal{X} \approx \mathcal{G} \times_1 U^{(1)} \times_2 U^{(2)} \cdots \times_N U^{(N)} = [\![\mathcal{G}; U^{(1)}, U^{(2)}, \ldots, U^{(N)}]\!] \tag{7}$$

where $\mathcal{G} \in \mathbb{C}^{R_1 \times R_2 \times \cdots \times R_N}$ is the core tensor, $U^{(n)} \in \mathbb{C}^{I_n \times R_n}$ are factor matrices, and $\times_n$ denotes the $n$-mode product. Two critical aspects make it crucial as discussed in the previous section ie Appendix C.

## Appendix E. Challenges of applying GaLore to neural operators

In order to use standard GaLore on tensor weights, the weights must first be reshaped into a matrix to compute the SVD for projection into a low-rank space. GaLore takes one rank parameter $r$, and projects high-rank gradients onto the first $r$ basis vectors of the corresponding SVD rotation matrix. When the weight matrix corresponds to an operator that maps between vectors, a single rank cutoff can be applied while preserving most information.

However, in the tensor case, weights correspond to higher-order maps between function spaces. Depending on the chosen strategy for reshaping tensor weights into a matrix, applying a single-dimension rank cutoff to the matrix may discard key information - for instance, for a tensor $W \in \mathbb{C}^{A \times B \times m \times m}$, where $A$ is the number of input channels, $B$ is the number of output channels, and

$m$ is the number of truncated Fourier basis modes along each dimension, reshaping $W$ into $W' \in \mathbb{C}^{ABm \times m}$ and cutting off the first dimension at rank $r$ may remove all information about Fourier modes along the first dimension, making function learning impossible. We call this method *GaLore* and provide several comparisons to demonstrate its flaws.

One particular flaw is the **Loss of mode-specific information**: that is by collapsing multiple tensor dimensions into one matrix dimension, we lose the ability to preserve different amounts of information along each tensor mode. The other is fact that we have an **imbalanced projection**: Projecting only on one side of the reshaped matrix (e.g. only $U$ or only $V$ from the SVD) can severely limit the operator's capacity. But projecting on both sides often leads to training instability and failure to converge. There is also the fact that we have a **rank selection issues**: Choosing a single rank cutoff for the reshaped matrix makes it difficult to balance information preservation across all the original tensor dimensions. A rank that preserves enough information for one dimension may be too restrictive for another.

## Appendix F. Main Algorithm

**Algorithm 1** Adam with Tensor-GaLore

**Require:** A layer weight tensor $\mathcal{W} \in \mathbb{C}^{N_1 \times N_2 \times N_3 \times N_4}$. Step size $\eta$, scale factor $\alpha$, decay rates $\beta_1, \beta_2$, rank $r$, subspace change frequency $T$.

1: Initialize first-order moment $\mathcal{M}_0 \in \mathbb{C}^{N_1 \times N_2 \times N_3 \times N_4} \leftarrow 0$
2: Initialize second-order moment $\mathcal{V}_0 \in \mathbb{C}^{N_1 \times N_2 \times N_3 \times N_4} \leftarrow 0$
3: Initialize step $t \leftarrow 0$
4: **repeat**
5: $\quad \mathcal{G}_t \in \mathbb{C}^{N_1 \times N_2 \times N_3 \times N_4} \leftarrow -\nabla_{\mathcal{W}} \phi_t(\mathcal{W}_t)$
6: $\quad$ **if** $t \bmod T = 0$ **then**
7: $\quad\quad \mathcal{C}, \{U^{(n)}\}_{n=1}^4 \leftarrow \text{Tucker}(\mathcal{G}_t, \text{rank} = r)$ ▷ Initialize projector.
8: $\quad$ **else**
9: $\quad\quad \mathcal{C}, \{U^{(n)}\}_{n=1}^4 \leftarrow \mathcal{C}_{t-1}, \{U_{t-1}^{(n)}\}_{n=1}^4$ ▷ Reuse the previous projector.
10: $\quad$ **end if**
11: $\quad \mathcal{R}_t \leftarrow \mathcal{G}_t \times_1 {U^{(1)}}^\top \times_2 {U^{(2)}}^\top \times_3 {U^{(3)}}^\top \times_4 {U^{(4)}}^\top$ ▷ Project gradient into compact space.
12: $\quad$ **UPDATE**($\mathcal{R}_t$) by Adam:
13: $\quad\quad \mathcal{M}_t \leftarrow \beta_1 \cdot \mathcal{M}_{t-1} + (1 - \beta_1) \cdot \mathcal{R}_t$
14: $\quad\quad \mathcal{V}_t \leftarrow \beta_2 \cdot \mathcal{V}_{t-1} + (1 - \beta_2) \cdot |\mathcal{R}_t \bar{\mathcal{R}}_t|$ ▷ We use the complex conjugate update.
15: $\quad\quad \mathcal{M}_t \leftarrow \mathcal{M}_t / (1 - \beta_1^t)$
16: $\quad\quad \mathcal{V}_t \leftarrow \mathcal{V}_t / (1 - \beta_2^t)$
17: $\quad\quad \mathcal{N}_t \leftarrow \mathcal{M}_t / (\sqrt{\mathcal{V}_t} + \epsilon)$
18: $\quad \tilde{\mathcal{G}}_t \leftarrow \alpha \cdot \mathcal{N}_t \times_1 U^{(1)} \times_2 U^{(2)} \times_3 U^{(3)} \times_4 U^{(4)}$ ▷ Project back to original space.
19: $\quad \mathcal{W}_t \leftarrow \mathcal{W}_{t-1} + \eta \cdot \tilde{\mathcal{G}}_t$
20: $\quad t \leftarrow t + 1$
21: **until** convergence criteria met.
22: **return** $\mathcal{W}_t$

## Appendix G. Datasets

### G.1. Burgers' Equation

We consider the one-dimensional Burgers' equation on the torus:

$$\partial_t u + uu_x = \nu u_{xx}, \quad x \in \mathbb{T}, t \in (0, T] \tag{8}$$

with initial condition $u_0 \in L^2(\mathbb{T}; \mathbb{C})$ and viscosity $\nu > 0$. We set $T = 1$ and $\nu = 0.01$. Input functions are sampled from a Gaussian random field, and solutions are obtained using a pseudo-spectral method. We use 1000 samples for training and 200 for testing, with an input resolution of 128 and an output resolution of 128.

### G.2. Darcy Flow

The Darcy flow problem is defined by the elliptic PDE:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in (0, 1)^2 \tag{9}$$

with boundary conditions $u(x) = 0$ for $x \in \partial(0, 1)^2$. The input $a$ is sampled from a Gaussian random field, and $f$ is fixed. We use 4000 training samples and 100 test samples, with the domain discretized on a 421 × 421 grid.

### G.3. Electromagnetic Wave Propagation

Lastly, we present a dataset that represents complex-valued data inherently. We consider the propagation of optical pulses in a nonlinear waveguide with second-order nonlinearity ($\kappa^2$). The problem is governed by the nonlinear Schrödinger equation (NLSE) with additional terms for second-harmonic generation:

$$\frac{\partial A}{\partial z} = -\mathrm{i}\frac{\beta_2}{2}\frac{\partial^2 A}{\partial t^2} + \mathrm{i}\gamma|A|^2 A + \mathrm{i}\kappa A^* e^{\mathrm{i}\Delta k z} \tag{10}$$

where $A$ is the complex electric field envelope, i is the imaginary unit, $z$ is the propagation distance, $t$ is time, $\beta_2$ is the group velocity dispersion, $\gamma$ is the nonlinear parameter, $\kappa$ is the coupling coefficient for second-harmonic generation, and $\Delta k$ is the phase mismatch. Our dataset consists of 800 training samples and 200 testing samples. The input consists of several parameters: the poling region length ranging from 2 mm to 15 mm, the poling period mismatch varying from -50 nm to +50 nm, and the pump pulse energy spanning from a few fJ to thousands of fJ. Additionally, the input includes the complex electric field envelope of the input pulse. The output of the system is the complex electric field envelope of the resulting output pulse.

## Appendix H. Profiling Methodology

To analyze the performance and memory usage of our Tensor-GaLore method, we implemented a comprehensive profiling setup using PyTorch's built-in profiler. This allowed us to gain detailed insights into the computational and memory requirements of our algorithm compared to baseline methods.

**Detailed Memory Breakdown.** We implemented a detailed memory tracking system to distinguish between various types of memory usage, including Model parameters, Optimizer states, Input

data, Activations, Gradients, Autograd details, Temporary buffers. To provide a comprehensive understanding of memory utilization in our experiments, we developed a classification system to distinguish between different types of memory usage. This granular approach allows us to precisely identify where memory savings occur when using Tensor-GaLore compared to baseline methods. :

- **Model Parameters.** Model Parameters are udentified by tracking tensors that are registered as model parameters (instances of ‘nn.Parameter‘). It is typically constant throughout training unless using techniques like weight decay.
- **Optimizer States.** Optimizer States are tracked by instrumenting the optimizer to log memory allocations for momentum buffers, adaptive learning rate parameters, etc. For Adam optimizer, this includes first and second moment estimates.
- **Input Data.** Input is monitored by tracking memory allocations that occur during data loading and preprocessing steps.
- **Activations.** Activations are identified as temporary tensors created during the forward pass of the model. it is tracked using hooks on module forward methods to capture intermediate outputs.
- **Activations.** Activations are identified as temporary tensors created during the forward pass of the model. it is tracked using hooks on module forward methods to capture intermediate outputs.
- **Gradients.** Gradients ared recognized as tensors with ‘requires_grad=True‘ that are outputs of operations on model parameters or inputs.
- **Autograd Details.** It is captured by profiling PyTorch’s autograd engine internals, including memory used for storing computational graphs and intermediate results needed for backpropagation.
- **Temporary Buffers.** Temporary Buffers are short-lived tensors that are created and destroyed within a single operation or a small set of operations. For tensor-galore, it is often used in complex computations like FFTs or tensor decompositions within galore.

To implement this detailed profiling, we used a combination of PyTorch’s memory-profiler, custom context managers, and function decorators. Key aspects of our implementation include:

- Wrapping key operations with context managers to track memory allocation and deallocation
- Using PyTorch hooks to monitor intermediate activations and gradients
- Instrumenting the optimizer to log memory usage for each parameter update
- Implementing custom memory tracking for Tensor-GaLore specific operations

The results of this analysis formed the basis for our discussions on memory efficiency in Sections 5 and 6 of the main paper, and provided the data for Figure 2, which illustrates the memory usage breakdown for different numbers of frequency modes in FNOs.

## Appendix I. Memory usage

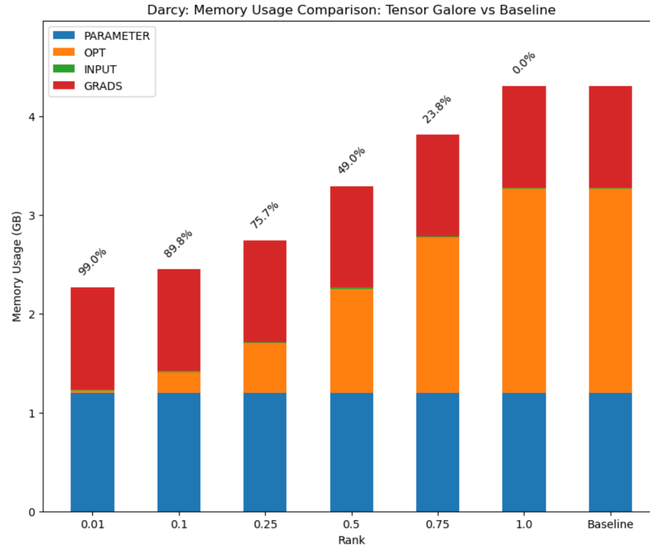


Figure 3: Memory usage of Tensor-GaLore vs Baseline on the Darcy Dataset. The text above the bar plots, represent the memory reduction of the optimizer in % compared to the baseline.

## Appendix J. Additional Results

Table 2: Model performance on Darcy-flow.

| Model | Test Loss (1e-2) at Rank Ratio | | | | | | Gain (%) |
|---|---|---|---|---|---|---|---|
| | 0.01 | 0.1 | 0.25 | 0.5 | 0.75 | 1.0 | |
| FNO Baseline | - | - | - | - | - | 0.205 | / |
| FNO - Tensor-GaLore | **0.147** | **0.108** | **0.105** | **0.107** | **0.140** | **0.173** | **49** |
| FNO - GaLore (d=1) | 0.256 | 0.232 | 0.212 | 0.245 | 0.201 | 0.190 | 8 |
| FNO - GaLore (d=2) | 0.203 | 0.192 | 0.168 | 0.178 | 0.170 | 0.180 | 19 |
| FNO - GaLore (d=3) | 0.234 | 0.212 | 0.201 | 0.193 | 0.196 | 0.182 | 11 |

## Appendix K. Architecture and Training Details

**Sobolev Loss for PDE Training** In training NOs for PDEs we employ both the $L^2$ and Sobolev $H^1$ losses to provide a comprehensive assessment of model performance. While the $L^2$ loss measures point-wise accuracy of predictions, the $H^1$ loss, defined as $\|u - \hat{u}\|_{H^1}^2 = \|u - \hat{u}\|_{L^2}^2 + \|\nabla u - \nabla \hat{u}\|_{L^2}^2$, accounts for both the function values and their gradients. This is particularly crucial for PDEs, as it ensures that the learned solutions not only match the target values but also preserve the smoothness and differential properties inherent in the physical systems being modeled.

**Sobolev Loss for Complex Wave Phenomena** The Sobolev $H^1$ loss proves especially valuable when dealing with complex wave phenomena, as demonstrated in our experiments with the EM Dataset using Complex-FNOs. In this case, the $H^1$ loss not only measures the accuracy of the predicted complex electric field envelope but also ensures that its spatial derivatives are correctly captured. This is crucial for accurately representing the rapid oscillations and sharp peaks characteristic of EM waves. Our results show that Tensor-GaLore with a rank ratio of 0.25 achieved an $11\%$ improvement in overall test loss compared to the baseline, with the $H^1$ loss decreasing from 0.1902 to 0.1681. This improvement is particularly significant given the challenging nature of the EM dataset, which involves predicting the complex electric field envelope resulting from nonlinear interactions in waveguides. The enhanced performance in $H^1$ loss indicates that our model not only matches the amplitude of the EM waves more accurately but also better captures the rapid spatial variations and peak formations. This is critical in applications such as optical pulse propagation, where precise modeling of field gradients and peak intensities is essential for predicting phenomena like second-harmonic generation and phase matching.

| **Dataset** | **Model** | **Architecture Details** | **Optimizer & Scheduler** |
|---|---|---|---|
| Burgers | FNO | • 4 layers, 90 modes<br>• 256 hidden channels, 256 projection channels<br>• Skip Connections: 'linear'<br>• Positional embedding: 'grid' | Adam with step LR $3e-4$, weight decay $2e-6$ 500 epochs, batch size 16. Trained with $H_1$ loss. |
| Darcy Flow | FNO | • 4 layers, 64 modes<br>• 128 hidden channels, 128 projection channels<br>• Skip: 'linear' | Adam with step LR $1e-3$, weight decay $1e-4$, 250 epochs, batch size 2. Trained with $L_2$ loss. |
| EM Wave | Complex-FNO | • 8 layers, 128 modes<br>• 128 hidden channels, 128 projection channels<br>• Skip: 'linear'<br>• Complex data: True<br>• Complex activation function: True | Complex Adam with step LR 1e-4, weight decay 2e-6, batch size 32, 1000 epochs. Trained with $H_1$ loss. |

Table 3: Detailed FNO Architecture Specifications for Different Datasets