TWO SPARSE MATRICES ARE BETTER THAN ONE: SPARSIFYING NEURAL NETWORKS WITH DOUBLE SPARSE FACTORIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural networks are often challenging to work with due to their large size and complexity. To address this, various methods aim to reduce model size by sparsifying or decomposing weight matrices, such as magnitude pruning and low-rank or block-diagonal factorization. In this work, we present **Double Sparse Factorization (DSF)**, where we factorize each weight matrix into two sparse matrices. Although solving this problem exactly is computationally infeasible, we propose an efficient heuristic based on alternating minimization via ADMM that achieves state-of-the-art results, enabling unprecedented sparsification of neural networks. For instance, in a one-shot pruning setting, our method can reduce the size of the LLaMA2-13B model by 50% while maintaining better performance than the dense LLaMA2-7B model. We also compare favorably with Optimal Brain Compression, the state-of-the-art layer-wise pruning approach for convolutional neural networks. Furthermore, accuracy improvements of our method persist even after further model fine-tuning.

1 INTRODUCTION

Sparse neural networks have gained attention due to their potential to reduce computational costs and memory usage, making them more efficient for deployment on resource-constrained devices (LeCun et al., 1989; Han et al., 2015; Hoefler et al., 2021). By reducing the number of non-zero parameters, sparse networks can achieve accuracy similar to dense networks while requiring fewer operations. Reducing network size also decreases the number of weights that must be loaded into the processing unit from memory, which is crucial since memory bandwidth often becomes a bottleneck in neural network deployments, particularly during single-sample LLM inference (Xia et al., 2023).

In this work, we propose an improvement over a typical neural network sparsification. Instead of replacing each dense weight matrix with a sparse matrix, we replace each dense matrix with a product of two sparse matrices. We devise a heuristic algorithm for calculating sparse matrix factorization and achieve significant improvements over a wide range of models, including large language models and convolutional neural networks.

Summary of contributions. We propose a practical algorithm for factorizing a matrix into two sparse matrices called **Double sparse factorization (DSF)**. We extend it for the layer-wise pruning scenario where one wants to preserve layer behavior for a given set of calibration inputs. Our sparse factorization algorithm is a heuristic based on alternating minimization where each subproblem is solved using the ADMM algorithm for solving a sparse regression problem (Boža, 2024).

Our algorithm obtains superior results in the layer-wise pruning scenarios, where we fix the number of non-zero entries in each layer. We compare favorably to Optimal Brain Compression (Frantar & Alistarh, 2022) for pruning convolutional image models. We also produce state-of-the-art layer-wise pruning results for large language models, where the larger pruned model is clearly better than the dense smaller model (as far as we know, this is the first time for the uniform layer-wise pruning).

One could argue that our method requires storing one more pruning mask. We thus evaluate a scenario where one of the sparse factors mask (weights can be tuned, but nonzeros location is fixed)

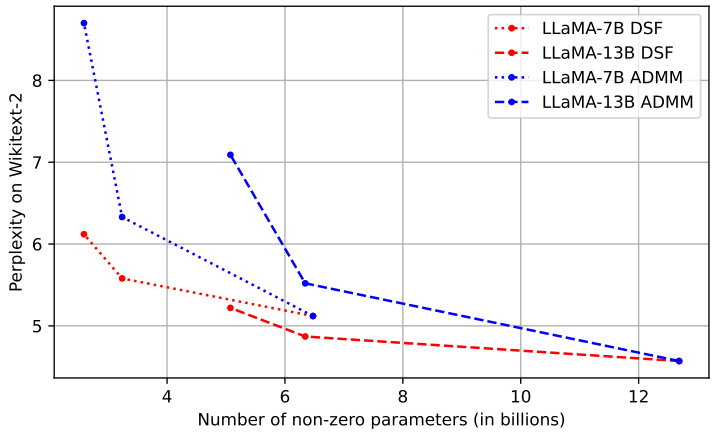

Figure 1: Comparison of LLaMA2 models pruned either using our Double Sparse Factorization (DSF) or using previously state-of-the-art ADMM pruning. We prune models using 0, 50, and 60% sparsities.

is randomly generated and fixed over the whole neural network. Our approach is better even in this scenario, which has almost the exact storage requirements as regular pruning.

Finally, we also show that our factorized pruning brings benefits even when sparsified models are further fine-tuned after pruning and achieve competitive results for pruning convolutional networks on CIFAR and ImageNet datasets.

2 Related Work

Neural network weight pruning and layer-wise one-shot pruning. Post-training network pruning compresses the already training network by removing redundant weights (LeCun et al., 1989; Han et al., 2015; Blalock et al., 2020; Liu et al., 2018; Hoefler et al., 2021; Srinivas et al., 2022).

Some approaches focus on splitting the network into individual layer-wise problems, where one wants to preserve layer behavior over a small set of calibration inputs. Optimal Brain Compression (OBC) (Frantar & Alistarh, 2022) removes one weight at a time and optimally updates the remaining weights in the layer. However, this approach is not feasible for large language models due to high computational cost. SparseGPT (Frantar & Alistarh, 2023) uses various approximations and turns OBC into a more practical algorithm at the expense of higher approximation error. Wanda (Sun et al., 2023) proposes to skip the weight update and prune weights based on the product of absolute magnitude and input norm. Finally, Boža (2024) obtains state-of-the-art layer-wise pruning results using an ADMM-based algorithm, which uses gradual pruning combined with Wanda mask selection and ADMM (Boyd et al., 2011) weight update.

Compression based on matrix factorization. Instead of turning weight matrices into sparse matrices, one can replace them with a product of multiple smaller matrices. A typical example is a low-rank factorization (Li & Shi, 2018; Jaderberg et al., 2014) where one turns an $n \times m$ matrix into a product of $n \times k$ and $k \times m$ matrices, where $k << \min(n, m)$. More complicated examples include butterfly matrices (Dao et al., 2019) and Monarch matrices (Dao et al., 2022), where individual factors have some specific structure. Monarch matrices are the product of block-diagonal, permutation, and another block-diagonal matrix. The projection of a matrix into a set of monarch matrices is done by splitting the original matrix into blocks and then running a low-rank decomposition of each block. Another option is to decompose the matrix as a sum of low-rank and sparse matrix (Nikdan et al., 2024; Yu et al., 2017; Ke & Kanade, 2005; Wright et al., 2009).

Separable convolutions. Convolutional layer can be naturally factorized into depthwise (applying filter per input channels) and pointwise convolution (mixing multiple channels). The idea was

found initially in MobileNets (Howard, 2017; Sandler et al., 2018), but they placed nonlinearity between the depthwise and pointwise convolutions. However, some works successfully use separable convolutions without nonlinearity between them (Perešíni et al., 2021; Kriman et al., 2020).

Sparse matrix factorization. Factorization of the matrix into (sometimes more than two) sparse factors has already been studied. It was shown that this problem is NP-hard even when the sparsity pattern for factors is given (Le et al., 2021). Le Magoarou & Gribonval (2016) provides a heuristic based on the proximal gradient step called palm4msa, which is then used by Giffon et al. (2021) for compression of neural networks, but with very limited practical success. In the experiments section, we compare the quality of our factorization algorithm with palm4msa. There were also works using sparse matrix factorization for parameter efficient fine-tuning (Chen et al., 2024).

3 PRELIMINARIES

In this work, we work with the post-training neural network sparsification scenario. We are given an already-trained network, and we will replace each weight matrix with a matrix that can be represented more efficiently, such as a sparse (Hoefler et al., 2021) or Monarch matrix (Dao et al., 2022). Usually, the replacement is done by solving the **projection** problem, where we are looking for a matrix closest (typically using the Frobenius norm) to the original one. For example, when the target matrix is sparse, solving the projection problem is just the magnitude pruning (Han et al., 2015).

In many cases, the sparsified network is often fine-tuned further. This can be prohibitive in some applications, especially involving large language models. We often resort to **one-shot** pruning in such cases. We capture relevant statistics for each layer and prune them during one forward pass. This is usually done by solving the **layer-wise pruning** problem (Frantar & Alistarh, 2022; 2023; Boža, 2024), where given calibration input X, original matrix W, one looks for sparse matrix W_p, such that the layer-wise error $||XW - XW_p||_2^2$ is minimized.

3.1 LAYER-WISE PRUNING VIA ADMM

Boža (2024) solves the layer-wise pruning problem by application of the alternating direction method of multipliers (Boyd et al., 2011) (ADMM). ADMM solves convex problems of the form: find minimum of $f(x) + g(y)$, subject to $Ax + By = C$, using iterations:

$$
\begin{aligned}
x^{k+1} &= \arg\min_x f(x) + (\rho/2)||Ax + Bz^k - c + u^k||_2^2 \\
z^{k+1} &= \arg\min_z g(z) + (\rho/2)||Ax^{k+1} + Bz - c + u^k||_2^2 \\
u^{k+1} &= u^k + Ax^{k+1} + Bz^{k+1} - c
\end{aligned}
$$

Note that when the pruning mask is fixed, the layer-wise pruning problem is a convex problem (we have one linear regression for each output with a different set of inputs).

This can solved via ADMM as follows: Given X, W, and pruning mask M, we are looking for W_p such that $(1 - M) \odot W_p = 0$ and layer-wise error is minimized. In ADMM formulation, $f(W)$ represents the layer-wise error, and $g(Z)$ would be an indicator function, which has a value of 0 when Z has the correct mask and ∞ otherwise. Then we can apply following ADMM iterations (ρ is penalty factor usually set to one, U represents scaled dual variables):

$$
\begin{aligned}
Z^0 &= M \odot W \\
U^0 &= 0 \\
\widehat{W}^{(k+1)} &= (X^T X + \rho I)^{-1}(X^T XW + \rho(Z^{(k)} - U^{(k)}) \\
Z^{(k+1)} &= M \odot (\widehat{W}^{(k+1)} + U^{(k)}) \\
U^{(k+1)} &= U^k + \widehat{W}^{(k+1)} - Z^{(k+1)}
\end{aligned} \tag{1}
$$

We will take final $W_p = Z^{(m)}$ as the output.

Boža (2024) then applies the following improvements: Preconditioning is applied first to improve convergence. All input feature norms are normalized to one (and the matrix W is multiplied by original input norms), which means that the diagonal of $X^T X$ contains only ones.

A pruning mask is found heuristically during the optimization process. During the first iterations, gradual magnitude pruning (Boža (2024) found that dropping smallest values from $W^{k+1} + U^k$ is better than dropping smallest values from current valid solution Z^k) with cubic schedule (Zhu & Gupta, 2018) is applied (in the original paper, cubic prune is applied during the first 15 iterations out of 20).

4 Double Sparse Factorization

In typical neural network pruning, we replace weight matrix W with matrix W_p which has at most z nonzeros, i.e. $||W_p||_0 \leq z$. Here, we propose to replace weight matrix W with shape $n \times m$ with a product of two sparse matrices AB such that they have at most z nonzeros in total, i.e. $||A||_0 + ||B||_0 \leq z$. We call this a **double sparse factorization**. Usually, we assume that A is a matrix with shape $n \times n$, B is a matrix with shape $n \times m$, and $n \leq m$; if not, we transpose the matrix W.

During neural network inference, we multiply some input X of shape $b \times n$ with matrix W. After our double sparse factorization which replaces W with product AB, we will first multiply X by A and then by B, i.e. doing $(XA)B$. Note, that the total number of multiplications is bz, the same as in typical neural network pruning.

4.1 Expressiveness and Efficiency of Double Sparse Factorization

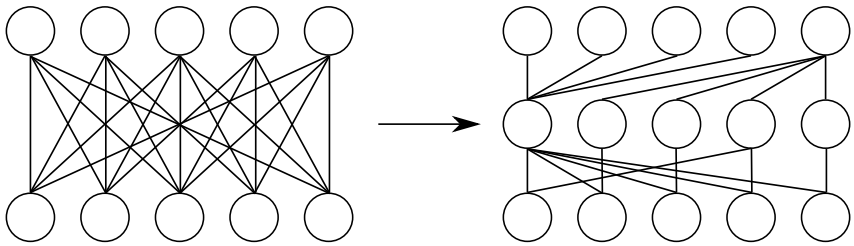

Figure 2: Graphical illustration of double sparse factorization. A dense layer is turned into two sparse layers. With enough weights in sparse matrices, most connections will be covered by a path through sparse matrices.

Many matrix factorizations mentioned previously in the literature can be (often trivially) rewritten to double sparse representation with the same number of non-zeros. For example, low-rank factorization commonly done via SVD (Li & Shi, 2018; Stewart, 1993) is already in the double sparse form. Monarch factorization (Dao et al., 2022), which is a product of block-diagonal, permutation, and block-diagonal matrices, can represented in double sparse form by fusing a permutation matrix with one of the block-diagonal matrices. Also, DSF can efficiently represent a matrix that consists of multiple disjoint low-rank submatrices.

The tricky case is an ordinary sparse matrix W_p. It can be represented in double sparse form as a product of identity and the original matrix: IW_p. However, this comes with the cost of additional non-zero entries for the identity matrix. Nevertheless, in the experiments, we will show that the double sparse representation represents the original dense matrix much better than an ordinary sparse matrix with the same number of non-zeros.

4.2 Heuristic Algorithm for Double Sparse Factorization

First, we look into the **projection** problem. Given matrix W, we want to replace it with some compressed matrix W_c, where their difference $||W_c - W||_F$ is minimized.

In our case, W_c is a product of two sparse matrices A, B. Thus, we are given matrix W and are looking for matrices A, B such that:

$$\begin{aligned} \text{minimize} \quad & ||AB - W||_F \\ \text{subject to} \quad & ||A||_0 + ||B||_0 \leq z \end{aligned}$$

This problem is NP-hard even when the sparsity pattern for matrices A and B is given (Le et al., 2021) thus we solve our problem heuristically.

First, we decide how many nonzeros we allocate for each matrix, so our condition changes to $||A||_0 \leq z_a, ||B||_0 \leq z_b$. These allocations were determined manually in our experiments. In general, we found that it is beneficial to give one of the matrices approximately 1/3 of the nonzeros and 2/3 to the other one. Then, we continue with an alternating minimization algorithm. We fix the value of A and find the best possible B, then fix the value of B and try to find the best possible value of A. We repeat this process multiple times.

One inner step of our algorithm can formalized as: Given W, and B, find A such that:

$$\begin{aligned} \text{minimize} \quad & ||AB - W||_F \\ \text{subject to} \quad & ||A||_0 \leq z_a \end{aligned}$$

This problem is just an L_0 constrained linear regression. We solve it using an iterative ADMM solver Boža (2024) mentioned in preliminaries, which heuristically finds matrix mask and corresponding values. We also apply the following heuristic[1] improvements.

Algorithm 1 Heuristical sparse matrix factorization for solving projection problem. Given matrix W, number of outer iterations n, number of inner iterations m and number of nonzero elements z_a, z_b we find A, B such that $||A||_0 \leq z_a$, $||B||_0 \leq z_b$ and AB is as close as possible to W.

Initialize $A^{(0)}, B^{(0)}$
$U_a^{(0)} = 0 \cdot A, U_b^{(0)} = 0 \cdot B$
for $k = 1..n$ **do**
  $\rho_0 = \min(1.0, k/(n-3))^3$
  $B^{(k)}, U_b^{(k)} \leftarrow$ solve $\arg\min ||AB - W||_F$, st. $||B||_0 \leq z_b$ via m iterations of ADMM
    with starting point $B^{k-1}, U_b^{(k-1)}$ and starting ρ_0
  $A^{(k)}, U_a^{(k)} \leftarrow$ solve $\arg\min ||AB - W||_F$, st. $||A||_0 \leq z_a$ via m iterations of ADMM
    with starting point $A^{k-1}, U_a^{(k-1)}$ and starting ρ_0
end for

Warm starting the inner iterations. To improve the convergence of ADMM iteration, we can warm-start it using the result from the previous step. To do that, we use not only the resulting sparse matrix but also all the dual variables U from the ADMM algorithm. This allows us to decrease the number of inner iterations and speed up our algorithm.

Annealing. We found that our algorithm is often quickly stuck in some local optima. To prevent that, we propose a simple annealing scheme. The first step of ADMM for finding B is $\widehat{W}_b^{(1)} = (A^T A + \rho I)^{-1}(A^T W + \rho(B^{(0)} - U_b^{(0)}))$. Instead of using ρ in the first iteration, we use smaller ρ_0 (we use default $\rho = 1$ in the remaining steps of ADMM). This gives lower weight to the previous solution and allows us to escape from local minima at the first steps of the optimization. We gradually increase ρ_0 from 0 to 1 throughout the optimization; we found that a simple cubic schedule works best. We also found that using more outer iterations (n) and fewer inner iterations (m) leads to better results.

Initialization. To run our algorithm, we must assign an appropriate starting values to matrices A and B. We tested several choices (ablations are provided in the Appendix A.2), including random initialization and singular value decomposition, but we settled on initializing A as an identity matrix and B with magnitude pruning of the original input matrix.

[1] some people call this a dark magic

4.3 Application of Sparse Factorization to Layer-wise Pruning

Next, we look into **layer-wise pruning problem**. In our case of the sparse factorization, we are given calibration input X, original weight matrix W and are looking for sparse A, B such that the reconstruction error $||XW - XAB||_2^2$ is minimized.

We solve this problem by first running the weight projection algorithm from the previous section. However, for the pruning of LLMs, we found that it is better to project the weight matrix multiplied by input feature norms. This was previously done in Wanda pruning algorithm (Sun et al., 2023). We then scale one of the factors back. More formally we first find A' and B such that error $|| \, ||X||_2 \odot W - A'B||_2^2$ is minimized and then compute $A = \frac{1}{||X||_2} \odot A'$. We do not do this rescaling for vision models.

We then proceed with the **finalization** step. We fix all sparsity masks and apply the ADMM algorithm for finding B so that $||XW - XAB||_2^2$ is minimized. This is a straightforward modification of the ADMM algorithm.

However, finding A is tricky and sometimes numerically unstable. In the inner iteration of ADMM, we need to find A such that (Z, U are other variables from ADMM optimization): $||XW - XAB||_2^2 + \rho/2||A - Z + U||_2^2$ is minimized. After taking gradients, we solve the equation: $X^TXABB^T + \rho A = X^TXWB^T + \rho(Z - U)$. This is a special type of Sylvester equation Roth (1952); Jiang & Wei (2003), which can be solved using the eigendecomposition of X^TX and BB^T. We provide a solution to this problem in the Appendix A.1. We found that optimizing A is only helpful for compressing vision models; we do not use it when compressing large language models.

4.4 Computational considerations for DSF

The obvious drawback of DSF is having two sparse matrices compared to one. Here, we argue that doing computation with two sparse matrices needs resources comparable to doing computation with one sparse matrix with the same total number of nonzeros.

Storage requirements. Storing actual non-zero values has the same memory footprint for one sparse matrix and for DSF. The difference lies in storing nonzero positions (sparsity masks).

If we store sparsity masks as bit vectors, DSF would need to store two masks rather than one. This leads to a 2x increase in storage costs for square matrices but a smaller increase for rectangular matrices (for example, in Llama-2-7B, we have matrices with size 4096x11008, which would lead to a 37% increase in storage cost for masks). Remember that we also store nonzero values (usually 16-bit floats), and they take the majority of the storage costs. Overall, when measured on Llama-2-7B with 50% density, regular pruning would have a model size of 7.3GiB and DSF 7.7GiB. In the Appendix A.5, we compare the quality of the pruned LLM with the actual storage size.

The storage requirements will be identical if we store sparsity as positions (e.g., using a compressed sparse row format). However, this format is preferable only for higher sparsities since storing one position index usually takes 16 (or more) bits. There are also various other storage formats (e.g., delta coding), but they are unexplored in the context of neural network sparsity.

Inference time. One could think that doing two sparse multiplications would be much slower than doing just one because of the sparse matrix multiplication overhead. However, when looking at actual benchmarks, we find that in lower sparsity ranges (50-95%), doing two sparse multiplications is just slightly slower than doing one sparse multiplication with the same number of nonzeros. For example, Xia et al. (2023) reports in their benchmark, that multiplication with 60% sparsity takes 0.36s and multiplication with 80% sparsity takes 0.21s, which would translate into 0.42s for DSF. Similar trends can be found for GPU kernels done by Gale et al. (2020). We also tested CPU inference and found that DSF has similar speed as ordinary sparse multiplication, see Appendix A.3.

We would like to point out that in the case of single sample LLM inference, the bottleneck is loading weights from memory to computational unit as reported in (Xia et al., 2023), and the current token activations can usually reside in the fast cache. Also, one goal of the pruning is to fit the model into available GPU memory, and in some cases, inference time can be sacrificed.

Table 1: Perplexity on Wikitext-2 for layer-wise pruning of large language models. Density refers to the total % of nonzero weights compared to the dense model.

Density	Method	1-7B	2-7B	2-13B	2-70B
100%	Dense	5.68	5.12	4.57	3.12
50%	Wanda	7.26	6.42	5.56	3.98
	ADMM	7.06	6.33	5.52	3.95
	DSF	**6.12**	**5.58**	**4.87**	**3.44**
	DSF no fin.	6.17	5.61	4.89	3.45
	DSF one mask fix	6.57	6.05	5.31	3.67
40%	Wanda	10.66	9.71	7.75	4.98
	ADMM	9.22	8.70	7.09	4.81
	DSF	**6.66**	**6.12**	**5.22**	**3.79**
	DSF no fin.	6.76	6.29	5.32	3.81
	DSF one mask fix	7.82	7.47	6.21	4.27
30%	Wanda	80.26	74.41	44.57	10.35
	ADMM	18.66	17.51	13.82	7.80
	DSF	**8.33**	**8.01**	**6.43**	**4.56**
	DSF no fin.	9.13	10.82	7.5	4.59
	DSF one mask fix	15.07	16.49	10.87	5.99

Fine-tuning considerations. Another concern is that during possible fine-tuning of the sparsified model, we need to store additional intermediate activations (in the middle of the double sparse factorization). This is true, but we found that with gradient checkpointing turned on and storing weights in compressed format (not as dense matrices), we can fine-tune on almost similarly sized sequences with DSF as when using regular pruning. We provide more details in the Appendix A.4

5 Experiments

We evaluate our proposed Double Sparse Factorization in multiple settings. First, we test it on layer-wise pruning of large language models. We compare our algorithms to ADMM pruning (Boža, 2024), which produces high-quality solutions in a reasonable time, even for large-scale models. Then we test it also on layer-wise pruning of vision models and compare it with Optimal Brain Compression (Frantar & Alistarh, 2022), state of the art layer-wise pruning algorithm.

We then proceed with the evaluation of the quality of our algorithm on the matrix projection problem. We compare with various matrix compression algorithms, including palm4msa (Le Magoarou & Gribonval, 2016), Monarch decomposition (Dao et al., 2022), and SVD. Finally, we also test whether models compressed with DSF can be successfully fine-tuned.

5.1 Layer-wise Pruning of Large Language Models

Setup. We follow same setup as in Wanda (Sun et al., 2023) and ADMM pruning (Boža, 2024). We use 128 calibration samples from the C4 training dataset (Raffel et al., 2020) and prune layers sequentially in order. We prune LLaMA (Touvron et al., 2023a) and LLaMA-2 (Touvron et al., 2023b) models. Similarly to previous works, we measure perplexity on held-out Wikitext (Merity et al., 2016). When factorizing square matrices (mainly in self-attention), we set the sparsity of one sparse factor to 16%. When factorizing rectangular matrices, the smaller factor will have 25% sparsity. The number of nonzeros in the other factor is just the target number of nonzeros minus the number of nonzeros in the first factor.

Compared methods. We compare our Double Sparse Factorization in three settings. The first one is the default one, solving the layer-wise pruning problem. Then, we disable the finalization step; thus, we only approximate the original dense matrix scaled by the input feature norms and solve the matrix projection problem. Finally, we fix one of the sparse masks to a random mask shared across all layers (but we run the finalization step). We compare our method with two layer-wise pruning algorithms: Wanda (Sun et al., 2023), which prunes weights with the smallest product of value and

Table 2: Comparision of our Double Sparse pruning vs. Optimal Brain Compression on Resnet50 using Imagenet dataset.

Number of nonzeros	FLOP reduction	Method	Test accuracy [%]
25.5M	-	Dense	76.13
16.8M	2x	OBC	75.65
		DSF	**75.78**
12.3M	3x	OBC	75.01
		DSF	**75.56**
10.2M	4x	OBC	74.05
		DSF	**74.95**

activation norm, and ADMM pruning (Boža, 2024), which also updates weights during the pruning using alternating direction method of multipliers.

Results. Results are summarized in Tab. 1 and Fig. 1. Our Double Sparse Factorization is superior to previous layer-wise pruning methods. To our knowledge, this is the first time when a layer-wise pruned network has better perplexity than its dense counterpart (compare 50% pruned LLaMA2-13B with perplexity 4.87 to dense LLaMA2-7B with perplexity 5.12). Even when we fix one mask (and thus make the total size of the network the same as in regular pruning), our factorization produces favorable results. We also notice that in the lower sparsities, the finalization step is not that important but becomes noticeably important at higher sparsities. We also report additional results with respect to total model size in Appendix A.5.

Pruning speed. We can prune the 7B models in apx. 30 minutes on one Nvidia 4090 GPU (this includes both forward pass and sparse factorization times). Note that reported total running times for ADMM pruning and SparseGPT are around 10-15 minutes (Boža, 2024).

5.2 Comparison with Optimal Brain Compression

Optimal Brain Compression (Frantar & Alistarh, 2022) is a post-training layer-wise pruning algorithm, which prunes each network layer by removing one connection at a time and optimally updating the remaining weights. Compared to the ADMM update algorithm mentioned in the previous section, it is much more accurate, but at the expense of much longer running time, unsuitable for large language models. However, OBC is still usable for moderately sized vision neural networks like ResNet50 (He et al., 2016).

In this experiment, we evaluate the effectiveness of our Double Sparse Factorization of ResNet50 on Imagenet (Russakovsky et al., 2015) dataset. We first run the OBC pipeline to determine layer-wise pruning ratios. Using the same calibration dataset as OBC, we then factorize every convolutional layer into two sparse matrices with the same number of nonzero weights as the OBC solution. We treat convolutions as linear layers, where input is processed via the im2col procedure. The sparsity of the smaller factor is set to $max(0.16, s/2)$ where s is sparsity from OBC. The bigger factor will get the remaining nonzeros (so the total nonzeros of sparse factors match the number of nonzeros used by OBC). Results are summarized in Tab. 2. We see that our solution is superior to the solution found by OBC for every sparsity setting, and the gap grows wider with larger sparsities.

5.3 Comparison with Other Matrix Approximation Methods

Now, we evaluate multiple methods for the weight projection problem. We use weight matrices from Llama-7B and Resnet-50. We truncate them to square matrices with sizes 64, 256, 1024, or 4096 (to accommodate Monarch factorization without problems). We evaluate our Double Sparse Factorization, palm4msa from Faust library (Le Magoarou & Gribonval, 2016), which also factories matrix into two sparse matrices, magnitude pruning, which keeps values with the largest magnitude, singular value decomposition, which factorizes matrix into two low-rank matrices, and Monarch decomposition (Dao et al., 2022), which factorizes matrix into block-diagonal, permutation and block-diagonal matrix. In all cases, we aim for 4x compression, i.e., each method can produce matrices that contain at most 25% of non-zeros in total compared to the original matrix.

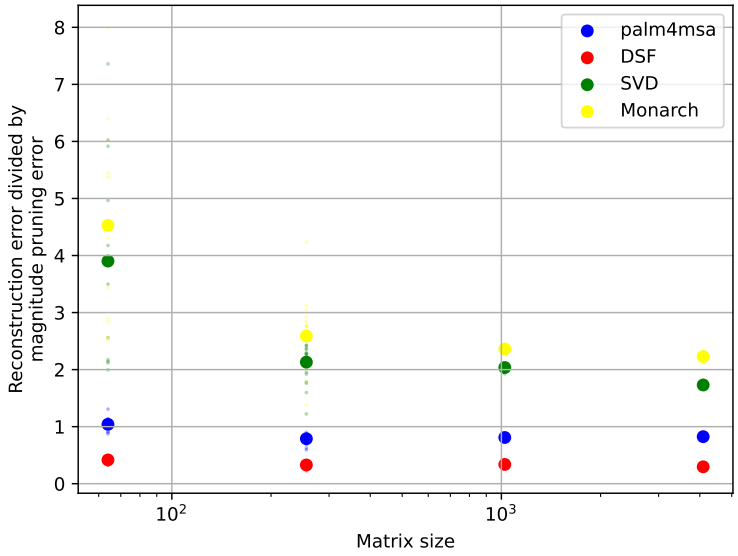

Figure 3: Reconstruction error of various compression methods on various matrix sizes for weight projection problem. We compress each matrix to 25% of the original size. We normalize error by error of magnitude pruning. The mean is denoted by a large dot and individual results with smaller dots.

Table 3: Test accuracy on CIFAR-10 using Resnet-20 with varying width. Density refers to the total % of nonzero weights compared to the dense model. FT refers to fine-tuning.

Density	Method	Resnet-20-16	Resnet-20-32
100%	Dense	92.2 ± 0.2	94.0 ± 0.1
20%	Magnitude w/o FT	70.6 ± 0.5	86.2 ± 0.3
	Double sparse w/o FT	80.0 ± 0.9	91.6 ± 0.1
	Magnitude w/ FT	91.2 ± 0.2	93.5 ± 0.1
	Double sparse w/ FT	$\mathbf{91.5 \pm 0.1}$	$\mathbf{93.6 \pm 0.1}$
10%	Magnitude w/o FT	29.0 ± 2.9	51.1 ± 5.2
	Double sparse w/o FT	48.9 ± 2.9	84.1 ± 0.5
	Magnitude w/ FT	89.3 ± 0.3	92.6 ± 0.1
	Double sparse w/ FT	$\mathbf{89.8 \pm 0.2}$	$\mathbf{93.0 \pm 0.2}$

Results are summarized in Fig. 3. We see that our DSF consistently outperforms other methods. Interestingly, palm4mse is not better for small matrix sizes than magnitude pruning. Also, Monarch decomposition seems to be worse than ordinary SVD.

5.4 Fine-tuning Models Pruned with Double Sparse Factorization

Finally, we test whether the Double Sparse Factorization accuracy advantage remains after fine-tuning a whole model. In this experiment, we only focus on the original matrix projection and do not perform any input-dependent finalization. We test the pruning of Resnet-20 (He et al., 2016) with varying starting widths (16 and 32) on the CIFAR-10 (Krizhevsky et al., 2009) dataset. We also test pruning Resnet-50 on Imagenet dataset (Russakovsky et al., 2015). In all experiments, we use the same sparsity in all layers. For CIFAR-10 experiments, we first train the dense network using the procedure from Liu et al. (2022). We train for 160 epochs using SGD with a starting learning rate of 0.1 and 0.9 momentum. We decay the learning rate by 10 on epochs 80 and 120. We then prune each layer (except the first and last one) to 10 or 20% of nonzeros using either magnitude pruning

Table 4: Test accuracy on Imagenet using Resnet-50. Density refers to the total % of nonzero weights compared to the dense model. FT refers to fine-tuning.

Density	Method	Test accuracy [%]
100%	Dense	76.13
20%	Magnitude w/o FT	54.43
	Double sparse w/o FT	71.85
	Magnitude w/ FT	75.43
	Cyclical pruning Srinivas et al. (2022)	75.3
	Double sparse w/ FT	**75.78**
10%	Magnitude w/o FT	9.87
	Double sparse w/o FT	55.76
	Magnitude w/ FT	73.32
	Cyclical pruning Srinivas et al. (2022)	73.3
	Double sparse w/ FT	**74.50**

or our double sparse factorization method (on the weight projection problem). Then, we fine-tune the model for 50 epochs, starting with a learning rate of 0.1 and linearly decay the learning rate to zero (this was inspired by (Zimmer et al., 2021)). We run each setting 5 times using different seed. Results are shown in Tab 3.

For the Imagenet experiment, we start with the pre-trained Resnet-50 from Torchvision (maintainers & contributors, 2016). We then uniformly sparsify every layer except the first and last one and fine-tune for 20 epochs using SGD, with a linear learning rate decay from 0.01 to zero and momentum of 0.9. We also compare with results reported by Srinivas et al. (2022), which prunes and fine-tunes the neural network in multiple cycles with resets (Cyclical pruning). Results are shown in Tab 4.

In all cases, starting test accuracy is higher for double sparse pruning and stays better when fine-tuned. This is especially evident at higher sparsities.

6 Conclusions and Future Work

In this work, we introduced Double Sparse Factorization (DSF), an approach to decompose weight matrices into two sparse matrices, enabling more efficient neural networks. By applying DSF, we significantly improved layer-wise pruning for both large language models (LLMs) and convolutional neural networks (CNNs). The method effectively reduced the number of parameters without sacrificing model accuracy, achieving state-of-the-art results compared to traditional pruning techniques. Furthermore, our approach kept its performance gains even after further fine-tuning. Our work is also one of the first to show that a sparse neural network can achieve more gains by employing a more complicated technique than just removing weights. One drawback of our solution is that individual layer-wise sparsities need to be determined manually beforehand (compared to magnitude pruning, which can work globally and determine sparsity in each layer automatically). Also, it is unclear how to integrate DSF with gradual pruning with fine-tuning the whole network between pruning steps. We leave these enhancements for future work.

References

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

Vladimír Boža. Fast and effective weight update for pruned large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL `https://openreview.net/forum?id=1hcpXd9Jir`.

Xuxi Chen, Tianlong Chen, Yu Cheng, Weizhu Chen, Ahmed Hassan Awadallah, and Zhangyang Wang. One is not enough: Parameter-efficient fine-tuning with multiplicative sparse factorization. *IEEE Journal of Selected Topics in Signal Processing*, 2024.

Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *International conference on machine learning*, pp. 1517–1527. PMLR, 2019.

Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pp. 4690–4721. PMLR, 2022.

Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. *Advances in Neural Information Processing Systems*, 35:4475–4488, 2022.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pp. 10323–10337. PMLR, 2023.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Luc Giffon, Stéphane Ayache, Hachem Kadri, Thierry Artières, and Ronan Sicre. Psm-nets: Compressing neural networks with product of sparse matrices. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2021.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

Andrew G Howard. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

Tongsong Jiang and Musheng Wei. On solutions of the matrix equations x- axb= c and x- axb= c. *Linear Algebra and its Applications*, 367:225–233, 2003.

Qifa Ke and Takeo Kanade. Robust l/sub 1/norm factorization in the presence of outliers and missing data by alternative convex programming. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pp. 739–746. IEEE, 2005.

Samuel Kriman, Stanislav Beliaev, Boris Ginsburg, Jocelyn Huang, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, and Yang Zhang. Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6124–6128. IEEE, 2020.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Quoc-Tung Le, Elisa Riccietti, and Rémi Gribonval. Spurious valleys, spurious minima and np-hardness of sparse matrix factorization with fixed support. *arXiv preprint arXiv:2112.00386*, 2021.

Luc Le Magoarou and Rémi Gribonval. Flexible multilayer sparse approximations of matrices and applications. *IEEE Journal of Selected Topics in Signal Processing*, 10(4):688–700, 2016.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Chong Li and CJ Shi. Constrained optimization based low-rank approximation of deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 732–747, 2018.

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. *arXiv preprint arXiv:2202.02643*, 2022.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. `https://github.com/pytorch/vision`, 2016.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Mahdi Nikdan, Soroush Tabesh, and Dan Alistarh. Rosa: Accurate parameter-efficient fine-tuning via robust adaptation. *arXiv preprint arXiv:2401.04679*, 2024.

Peter Perešíni, Vladimír Boža, Broňa Brejová, and Tomáš Vinař. Nanopore base calling on the edge. *Bioinformatics*, 37(24):4661–4667, 2021.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

William E Roth. The equations ax-yb= c and ax-xb= c in matrices. *Proceedings of the American Mathematical Society*, 3(3):392–396, 1952.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Suraj Srinivas, Andrey Kuzmin, Markus Nagel, Mart van Baalen, Andrii Skliar, and Tijmen Blankevoort. Cyclical pruning for sparse neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2762–2771, 2022.

Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4): 551–566, 1993.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

John Wright, Arvind Ganesh, Shankar Rao, Yigang Peng, and Yi Ma. Robust principal component analysis: Exact recovery of corrupted low-rank matrices via convex optimization. *Advances in neural information processing systems*, 22, 2009.

Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285*, 2023.

Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7370–7379, 2017.

Michael Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=Sy1iIDkPM`.

Max Zimmer, Christoph Spiegel, and Sebastian Pokutta. How i learned to stop worrying and love retraining. *arXiv preprint arXiv:2111.00843*, 2021.

A Appendix

A.1 Solving for A in the layer-wise reconstruction problem

Recall that we want to find sparse A such that $||XW - XAB||_2^2$ is minimized (where X is calibration input, W is the original weight matrix, and B is the other sparse factor).

In the inner iteration of the ADMM, we need to find A such that (Z, U are other variables from ADMM optimization): $||XW - XAB||_2^2 + \rho/2||A - Z + U||_2^2$ is minimized.

After taking gradients, we solve the equation:

$$X^T XABB^T + \rho A = X^T XWB^T + \rho(Z - U)$$

We solve this equation using eigendecomposition and one simple trick. We use following eigendecompositions:$X^T X = QDQ^T, BB^T = RER^T$ (where D, E are diagonal matrices and Q, R are orthonormal).

We then multiply the equation by Q^T from left and R from right and get:

$$DQ^T ARE + \rho Q^T AR = Q^T(X^T XWB^T + \rho(Z - U))R$$

We will now use that D, E are diagonal and create an outer product of their diagonals: $F = \mathrm{Tr}(D) \otimes \mathrm{Tr}(E)$. Now, we can use Hadamard product to get:

$$F \odot Q^T AR + \rho Q^T AR = Q^T(X^T XWB^T + \rho(Z - U))R$$

And with slight abuse of notation (where $F + \rho$ means adding ρ to every element of F) we get:

$$Q^T AR = Q^T(X^T XWB^T + \rho(Z - U))R \oslash (F + \rho)$$

And thus:

$$A = Q(Q^T(X^T XWB^T + \rho(Z - U))R \oslash (\mathrm{Tr}(D) \otimes \mathrm{Tr}(E) + \rho))R^T$$

A.2 Ablation of DSF settings

We investigate some variations of DSF settings in Fig. 4. As in the experiments section, we target to have 25% of nonzeros compared to the original matrices. Running shorter iterations, especially our cubic first iteration weight schedule, benefits the final result. We also provide ablations for varying size of one factor (Fig. 5), and also for various initializations (Fig 6).

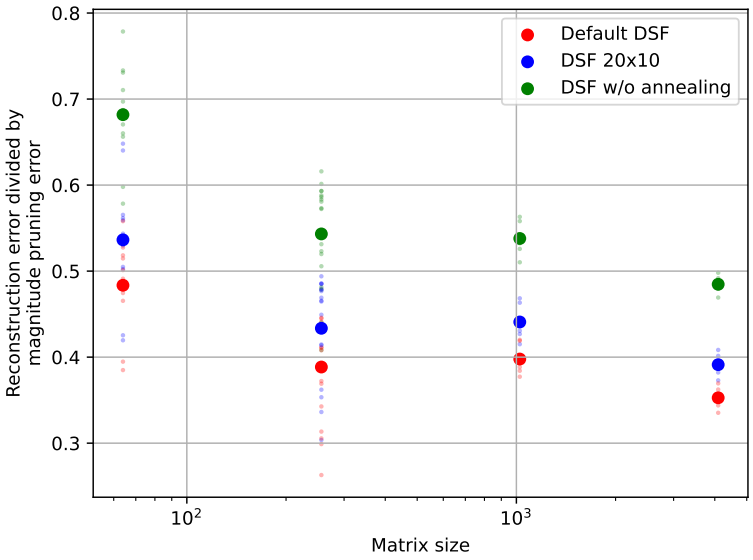

Figure 4: Reconstruction error of various settings of DSF. Default DSF used 40 outer and 5 inner iterations. DSF 20x10 refers to DSF with 20 outer and 10 inner iterations. DSF w/o annealing refers to DSF where we set first $\rho_0 = 1$.

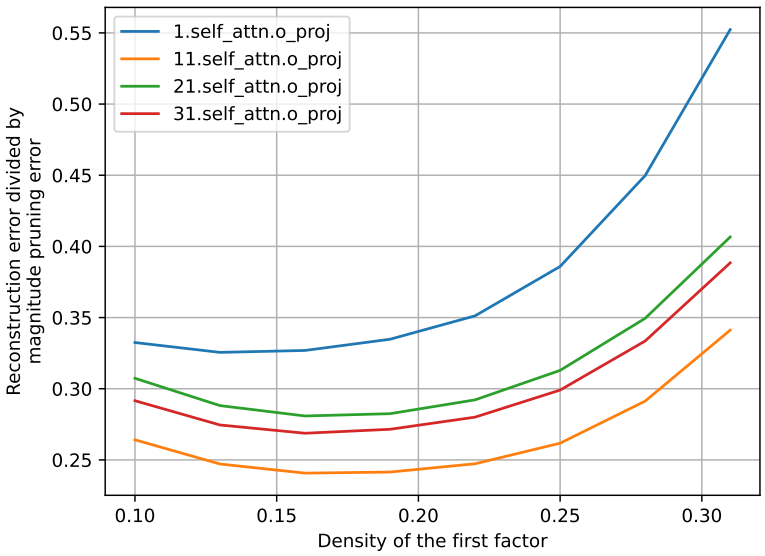

Figure 5: Reconstruction error for various densities of the first factor. We prune output projection from the attention layer in Llama-1 (matrix size 4096x4096) with a target total density of 40%. We see that the optimal density of the first factor is slightly less than half of the target total density.

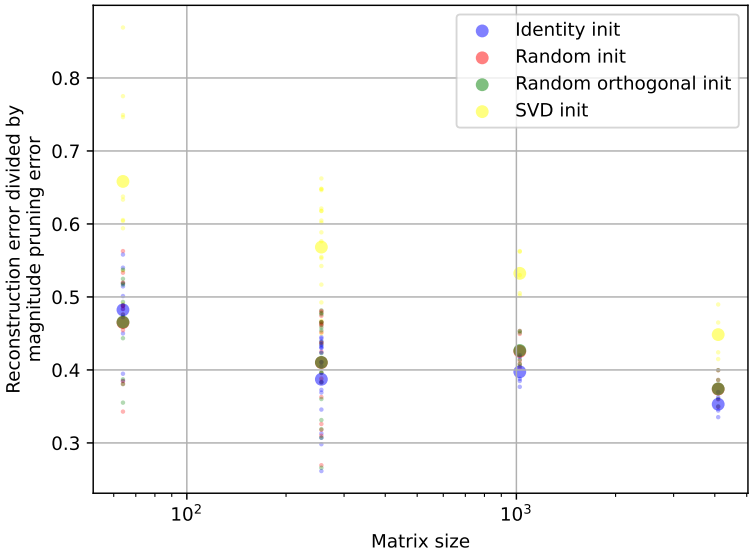

Figure 6: Reconstruction error for various initialization schemes of the first factor in DSF. We see that our choice of using identity init outperforms other common choices on all except the smallest matrices.

Table 5: Single CPU runtime for various number of layers and nonzeros in one layer. Each pair of rows simulates the comparison of ordinary sparsity vs DSF. Each layer has size of 1024x1024.

Number of layers	Nonzeros in one layer	Batch size	Execution time [s]
48	524288 (50%)	16	2.49
96	262144 (25%)	16	2.40
48	262144 (25%)	16	1.21
96	131072 (12.5%)	16	1.32
48	131072 (25%)	16	0.60
96	65536 (6.25%)	16	0.62
48	524288 (50%)	64	8.46
96	262144 (25%)	64	8.65
48	262144 (25%)	64	4.20
96	131072 (12.5%)	64	4.05
48	131072 (25%)	64	1.95
96	65536 (6.25%)	64	2.03

A.3 Sparse matrix efficiency on CPU

We use the following benchmark setup: We create 48 or 96 matrices of size 1024x1024 (the goal here is that all matrices do not fit into the cache and must be reloaded during inference). We then benchmark dense and sparse matrix multiplication over varying batch sizes and sparsity. We use a single-threaded matrix multiplication implementation from Intel MKL 2025.0 and store sparse layers using CSR format. Results are summarized in Fig. 7 and Table. 5, where we specifically simulate DSF by doubling the number of matrices and having them half of the nonzeros. We see, that DSF has comparable runtime to ordinary sparse matrix multiplication.

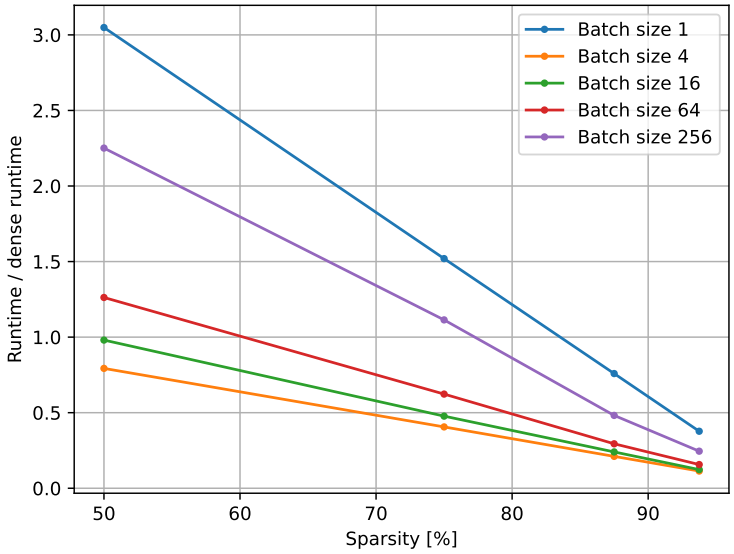

Figure 7: Singlethread CPU runtime for various sparsities compared to dense runtime. We run a single batch over 48 matrices with shape 1024x1024. We use latest Intel MKL for benchmarking.

Table 6: Maximum sequence length for various fine-tuning setups for Llama2-7B using A100 with 40GB of memory and batch size 16. We always have gradient checkpointing turned on.

Setup	Maximum sequence length
Dense (base Llama2-7B)	1900
50% sparsity	2600
DSF with 50% total density	2400

A.4 Effects of DSF on finetuning

DSF makes the network deeper, and during fine-tuning, one needs to store more intermediate activations. Moreover, if DSF results are stored as dense matrices, network size also increases.

We propose a simple storage solution for sparse matrices in Pytorch. We store nonzero values as parameters and packed masks as an auxiliary parameter. During the forward pass, we unpack everything into the dense matrix and then process the input. This incurs small time overhead (which gets smaller the bigger batch becomes). A custom kernel would obviously be a better solution, but writing custom GPU kernels is not the focus of this paper. This solution also only works if gradient checkpointing is enabled because otherwise, all of the unpacked dense matrices would be stored during the forward pass at once.

Here, we search for the maximum sequence length we can fine-tune. We finetune Llama2-7B using batch 16 on A100 GPU with 40GB of memory. We turn on gradient checkpointing and test dense, sparse, and DSF representations. As reported in Tab. 6, when using DSF, the maximum fine-tunable sequence length drops, but it is still higher than when using dense representation.

A.5 Storage requirements for LLM vs its quality

Now, we compare total storage size (nonzero values storage as 16-bit floats + binary mask) vs Wikitext2 perplexity. We measure various target densities for Llama2-7B. Results are shown in Fig 8. We can see that while DSF needs 0.4 GiB more storage, some same target density, it produces much better results and this trend is more pronounced at lower densities.

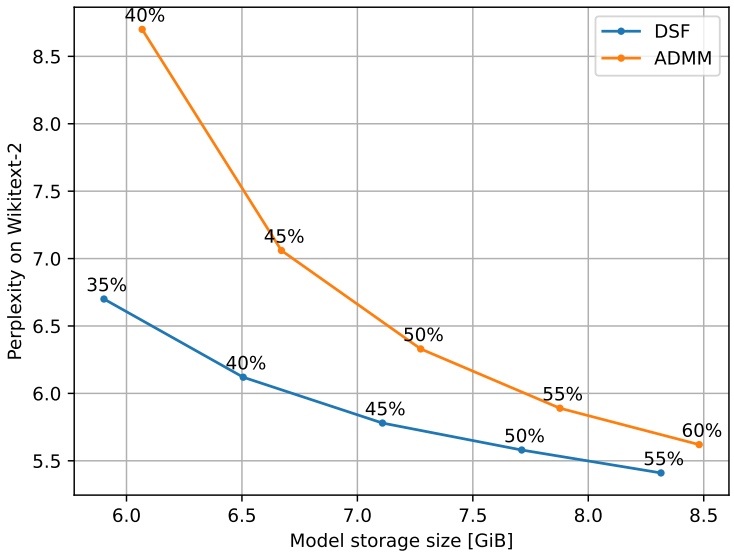

Figure 8: Comparison of total model size vs Wikitext2 perplexity for various target densities of Llama2-7B.