Training Block-wise Sparse Models Using Kronecker Product Decomposition

Ding Zhu **Zhiqun Zuo** **Mohammad Mahdi Khalili**
CSE Department, The Ohio State University, Columbus, OH 43210
`zhu.3723@osu.edu` `zuo.167@osu.edu` `khalili.17@osu.edu`

Abstract

Large-scale machine learning (ML) models are increasingly being used in critical domains like education, lending, recruitment, healthcare, criminal justice, etc. However, the training, deployment, and utilization of these models demand substantial computational resources. To decrease computation and memory costs, machine learning models with sparse weight matrices are widely used in the literature. Among sparse models, those with special sparse structures (e.g., models with block-wise sparse weight matrices) fit generally better with the hardware accelerators and can decrease the memory and computation costs during the inference. Unfortunately, while weight matrices with special sparsity patterns can make the models efficient during inference, there is no efficient method for training these models. In particular, existing training methods for block-wise sparse models start with full and dense models leading to an inefficient training process. In this work, we focus on training models with *block-wise sparse matrices* and propose an efficient training algorithm to decrease both computation and memory costs during the training. Our extensive empirical and theoretical analyses show that our proposed algorithms can decrease the computation and memory costs significantly without a performance drop compared to baselines.

1 Introduction

Deep learning models have achieved remarkable success across various domains, but training these models on resource-constrained devices remains challenging due to their computational and memory requirements. To decrease memory and computational costs, sparse machine learning models have been widely used in literature. Sparse networks are mainly categorized into two groups: fine-grained sparse networks and coarse-grained sparse networks (see Figure 1 which provides an example of a fine-grained sparse weight matrix and two examples of coarse-grained sparse weight matrices). In fine-grained sparse networks, the weight matrices are sparse but they do not have any special structure. This type of sparsity generally improves the storage cost but does not improve the inference time significantly. This is because the random distribution in fine-grained sparse weight matrices does not fit with the hardware accelerators, and they can speed up the inference time only if the sparsity ratio is higher than 95% [45, 46]. On the other hand, coarse-grained sparse matrices are better alternatives to speed up inference [34, 9, 10].

To train coarse-grained (structured) sparse weight matrices, it is common to use iterative pruning [40, 50] or group lasso techniques [1, 36, 18]. Iterative pruning works as follows. First, a sparsity pattern is defined (e.g., 2:4 sparsity[32] or channel-wise sparsity[28]). Based on this pattern, the weight parameters are divided into separate groups. A deep model is trained and then by looking at weight matrices, we prune some of the groups that are not impacting the performance of the model. Then, we re-train (fine-tune) the remaining weights and again prune those groups that have the smallest impact on the performance. This procedure is repeated until we achieve the desired sparse

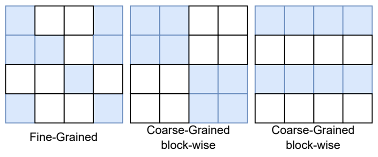

Figure 1: Examples for fine-grained and coarse-grained sparse matrices. White entries represent zero value.

network. Group LASSO technique adds a regularizer to the objective function which ensures that weights in several groups go to zero during the training. In particular, for Group LASSO, first, we divide the parameters in each weight matrix into several groups (each block can be considered as a group) and add a regularizer for each group to the objective function. As a result, the weights in certain groups with minimal impact on accuracy will eventually go to zero during training. Nevertheless, iterative pruning and group lasso techniques mainly focus on reducing memory and computation costs at the time of inference. However, they can be very costly during the training as the training process starts with the full network (i.e., all the weights are non-zero at the beginning) and the gradient needs to be calculated with respect to all the model parameters.

In this paper, we propose a technique that enables us to train block-wise sparse matrices that optimize both computation and memory costs during training and inference. We leverage the Kronecker product decomposition to propose a new matrix factorization technique suitable for block-wise sparse matrices. Our contributions in this work can be summarized as follows,

- To the best of our knowledge, this is the first work that focuses on efficient training for block-wise sparse models. This work is also the first work that makes a connection between block-wise sparse matrices and the Kronecker product decomposition for efficient training. While there have been several algorithms for training block-wise sparse models [2, 26, 23, 33], they are not efficient during training.
- We theoretically demonstrate every block-wise sparse matrix can be represented using our proposed decomposition resulting in no significant performance drop.
- Our theoretical analyses show that our proposed approach decreases the number of flops and training parameters during the training compared to existing methods for training block-wise sparse models.
- Through extensive empirical study, we show that in some cases, our proposed method can reduce the number of training parameters and training FLOPs by 97% with a minimal accuracy drop.

2 Preliminary

Group LASSO: LASSO is a method for learning sparse models by adding a penalty term to the cost function, which is proportional to the l_1 norm of the model's coefficients. This encourages the model to be sparse and sets some of the model parameters to be zero. Group LASSO is an extension of LASSO which divides the model parameters into several groups and imposes a regularizer on each group. This encourages the coefficient in several groups to go to zero during the training. Group LASSO can be used to train block-wise sparse matrices (see Figure 2) by defining each block as a group. In particular, consider a neural network with L layers, with $\mathbf{W}$ being the weight matrices of the whole network and $W^{[l]}$ being the weight matrix in layer l. Also, let $W_g^{[l]}$ denotes the block/group g in layer l. Then, the group LASSO solves the following optimization problem,

$$\hat{\mathbf{W}}_\lambda = \arg\min_{\mathbf{W}} \mathcal{J}(W^{[1]}, \dots, W^{[L]}; \mathcal{D}) + \lambda \sum_{l=1}^{L} \sum_{g} ||W_g^{[l]}||_F, \tag{1}$$

where $\hat{\mathbf{W}}_\lambda$ is the optimized weight matrices with hyperparameter λ, $\mathcal{J}$ is the loss function, and $\mathcal{D}$ is the training dataset, and $||.||_F$ denotes the Frobenius norm.

Kronecker Product Decomposition: The Kronecker product, denoted by $\otimes$, is a mathematical operation that combines two matrices to form a larger matrix. Given two matrices A and B, if A is of

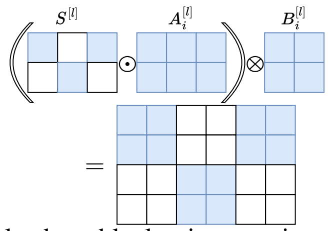

Figure 2: Illustration of why (3) leads to block-wise sparsity when $S^{[l]}$ is sparse. White entries represent zero value.

size $m_1 \times n_1$ and B is of size $m_2 \times n_2$, then the Kronecker product of A and B results in a matrix of size $m_1 m_2 \times n_1 n_2$. Let W be an m by n matrix, where $m = m_1 m_2, n = n_1 n_2$ we can decompose this matrix using the Kronecker product decomposition as follows [44],

$$W = \sum_{i=1}^{R} A_i \otimes B_i = \sum_{i=1}^{R} W_i, \tag{2}$$

where A_i is an m_1 by n_1 matrix, B_i is an m_2 by n_2 matrix, and $R = \min\{m_1 n_1, m_2 n_2\}$. Given this decomposition, Wx (where $x \in \mathbb{R}^{n \times 1}$) also can be calculated as follows, $Wx = vec(\sum_{i=1}^{R} B_i \check{x} A_i^T)$, where $\check{x}$ is n_2 by n_1 matrix and can be obtained by re-arranging the elements of vector x [44].

It turns out that the low-rank approximation is a special case of decomposition (2). More precisely, if we set $m_2 = 1$ and $n_1 = 1$, Equation 2 is equivalent to the low-rank approximation. Let's assume that we want to approximate matrix W by r terms of (2). In this case, $W \approx W_r = \sum_{i=1}^{r} A_i \otimes B_i$, where W_r is expressed by $r(m_1 n_1 + m_2 n_2)$ parameters.

It is worth mentioning that similar to the low-rank approximation, Kronecker product decomposition can be used for compressing deep models [19, 8]. However, to the best of our knowledge, there is no tensor/matrix factorization for generating and training block-wise sparse matrices. In the next section, we will explain how we can take advantage of Kronecker product decomposition to efficiently train block-wise sparse matrices and reduce the memory footprint during the training process.

3 Problem Statement and Proposed Solution

There are several methods including iterative pruning or group LASSO to train block-wise sparse weight matrices (see Section A.1 for related literature). However, these methods have to start with a full uncompressed model and sparsify the weight matrices gradually during the training. As a result, these methods do not decrease computation and memory during training. However, in this part, we propose a new matrix decomposition method leveraging the Kronecker product decomposition algorithm to train block-wise sparse matrices with fewer training parameters and fewer flops for forward and backward propagation compared to the group LASSO and structured pruning approaches. To demonstrate how to learn a structured sparse weight matrix efficiently, assume that $W^{[l]}$ is the weight matrix associated with layer l (if we are working with a convolutional neural network, $W^{[l]}$ can be a tensor.) Instead of learning $W^{[l]}$, we propose to estimate it by

$$W_r^{[l]} = \sum_{i=1}^{r_l} (S^{[l]} \odot A_i^{[l]}) \otimes B_i^{[l]}, \tag{3}$$

where r_l is a hyper-parameter called rank, $\odot$ is element-wise product, $S^{[l]}$ and $A_i^{[l]}$ are m_1 by n_1 matrices and $B_i^{[l]}$ is an m_2 by n_2 matrices. Then, we can train $S^{[l]}$, $(A_i^{[l]}, B_i^{[l]})_{i=1}^{r_l}$ directly during the training process (we calculate the gradient of the loss function with respect to these parameters). By imposing an l_1 regularizer on $S^{[l]}$, we can make sure that $S^{[l]}$ is sparse in the following problem,

$$\min_{[S^{[l]}, A_i^{[l]}, B_i^{[l]}]_{i \le r_l, l \le L}} \mathcal{J}([S^{[l]}, A_i^{[l]}, B_i^{[l]}]_{i \le r_l, l \le L}, \mathcal{D}) + \lambda \sum_{l=1}^{L} ||S^{[l]}||_1, \tag{4}$$

where λ is a constant and controls the sparsity rate. If $S^{[l]}$ is an unstructured sparse matrix, $W_r^{[l]} = \sum_{i=1}^{r_l} (S^{[l]} \odot A_i^{[l]}) \otimes B_i^{[l]}$ will be a block-wise sparse matrix (see Figure 2). After training, depending our application, we can use $W_r^{[l]}$ during the inference time or we can use $S^{[l]}$, $(A_i^{[l]}, B_i^{[l]})_{i=1}^{r_l}$ directly during the inference time. It is worth mentioning that the decomposition in (3) provides several degrees of freedom and hyper-parameters (i.e., m_1, m_2, n_1, n_2, r_l). Note that the block size in $W_r^{[l]}$ is determined by the size of matrix $B_i^{[l]}$ (i.e., (m_2, n_2)). If our only goal is to minimize the number of parameters using (3), then the hyper-parameters can be determined by an optimization problem. In particular, we can set $r_l = 1$ and solve the following integer programming,

$$\min_{m_1,n_1,m_2,n_2} 2m_1n_1 + m_2n_2, s.t., m_1m_2 = m, n_1n_2 = n, \tag{5}$$

where the objective function is equal to the number of parameters in (3). The above optimization problem is nonconvex. We can solve it by setting $m_2 = m/m_1$ and $n_2 = n/n_1$. The above optimization problem reduces to $\min_{m_1,n_1} 2m_1n_1 + \frac{mn}{m_1 \cdot n_1}$. By the first order condition, the minimizer of $2m_1n_1 + \frac{mn}{m_1 \cdot n_1}$ is $m_1n_1 = \sqrt{0.5 \cdot mn}$ if $\sqrt{0.5 \cdot mn}$ is integer (if $\sqrt{0.5 \cdot mn}$ is not an integer, we can use integer programming tools like branch and bound to solve the problem). To clarify, we provide an example.

Example 1 *Let $m = 2^3$ and $n = 2^8$. In this case, for the minimum possible space complexity under factorization* (3), *we need to have $m_1n_1 = 32$. For example, we can set $m_1 = 4, n_1 = 8, m_2 = 2, n_2 = 32$. In this case, the total number of parameters would be* 128. *The original matrix $W^{[l]}$, has* 2048 *training parameters. Therefore, at the time of training, using* (3), *we need to train* 128 *variables while the group LASSO technique needs to train* 2048 *variables. At the inference time, we can use directly sparse matrix $(S^{[l]} \odot A_1^{[l]})$ and $B_1^{[l]}$ to make an inference. We can also use block-wise sparse matrix $W_1^{[l]}$ to make an inference.*

The above example shows the proposed factorization in (3) can reduce the training parameters significantly. In the remaining part of this section, we explain how any block-wise sparse matrix can be represented by (3) and why the proposed factorization can decrease the number of flops during forward and backward propagation compared to group LASSO/Structured Pruning.

Proposition 1 *Let $\hat{W}^{[l]}$ be a block-wise sparse matrix trained by group LASSO or iterative pruning. If the blocks have the same size, then there exists $\hat{r}_l$ and $\hat{S}^{[l]}$ and $(\hat{A}_i^{[l]}, \hat{B}_i^{[l]})_{i=1}^{r_l}$ such that $\hat{W}^{[l]} = \sum_{i=1}^{\hat{r}_l} (\hat{S}^{[l]} \odot \hat{A}_i^{[l]}) \otimes \hat{B}_i^{[l]}$.*

Intuitively, the above proposition implies that if the hyper-parameters r_l, n_1, n_2, m_1, m_2 are chosen correctly, then training matrices $S^{[l]}, (A_i^{[l]}, B_i^{[l]})_{i=1}^{r_l}$ should have the same performance of a model trained by the group LASSO or pruning technique.

Proposition 2 *[Number of Flops for Forward and Backward Passes in A Linear Model] Consider multivariate linear regression model $h(x) = Wx$, where W is an m by n matrix, and $x \in \mathbb{R}^n$ is the input feature vector. Let $\mathcal{D} = \{(x_j, y_j)|j = 1, \ldots, N\}$ be the training dataset, and $\mathcal{J}(W; \mathcal{D}) = \sum_{j=1}^N ||Wx_j - y_j||_2^2$ be the objective function. If we estimate W by $\sum_{i=1}^r (S \odot A_i) \otimes B_i$ and write $\mathcal{J}(S, (A_i, B_i)_{i=1}^r; \mathcal{D}) = \sum_{j=1}^N || \sum_{i=1}^r [(S \odot A_i) \otimes B_i] x_j - y_j||_2^2$, then,*

- ***Forward pass:*** *Number of flops for calculating $\mathcal{J}(W; \mathcal{D})$ is $\mathcal{O}(2(n+1)mN)$. On the other hand, $\mathcal{J}(S, (A_i, B_i)_{i=1}^r; \mathcal{D})$ can be calculated by $\mathcal{O}\left(2Nrm_1n_1(m_2 + n_2) - Nr(m + 2m_2n_1) + 3Nm\right)$ flops.*
- ***Backward pass:*** *Number of flops for calculating gradient of $\mathcal{J}(W; \mathcal{D})$ with respect to W is $\mathcal{O}(mN(2n+1))$. On the other hand, $\frac{\partial \mathcal{J}(S,(A_i,B_i)_{i=1}^r;\mathcal{D})}{\partial A_i}$ and $\frac{\partial \mathcal{J}(S,(A_i,B_i)_{i=1}^r;\mathcal{D})}{\partial B_i}$ can be calculated by $\mathcal{O}(2Nn_1m_2(n_2 + m_1) - Nn_1m_2 + m_1n_1)$ flops and $\mathcal{O}(2Nm_1n_2(m_2 + n_1) + m_1n_1 - Nn_2m_1)$ flops, respectively. In addition, $\frac{\partial \mathcal{J}(S,(A_i,B_i)_{i=1}^r;\mathcal{D})}{\partial S}$ needs $\mathcal{O}\left(Nm + Nr(4m_1m_2n_1 - m_2n_1 + 2m_2n_1n_2)\right)$ flops.*

The above proposition implies forward and backward passes for $\mathcal{J}(S, (A_i, B_i)_{i=1}^r, \mathcal{D})$ can be more efficient compared to calculating $\mathcal{J}(W, \mathcal{D})$ if right values for parameters m_1, n_1, m_2, n_2, r are chosen. This is because the forward and backward passes for $\mathcal{J}(W, \mathcal{D})$ needs $\mathcal{O}(mnN)$ while mn does not appear in the time complexity of forward and backward passes of $\mathcal{J}(S, (A_i, B_i)_{i=1}^r, \mathcal{D})$. In addition to flops for calculating the gradient, in each epoch of gradient descent for $\mathcal{J}(W, \mathcal{D})$, we need to update

mn parameters which needs $\mathcal{O}(mn)$ flops. On the other hand, in each epoch of gradient descent for $\mathcal{J}(S, (A_i, B_i)_{i=1}^r, \mathcal{D})$, after calculating the gradient, we update $r(m_1n_1 + m_2n_2)$ parameters which need $\mathcal{O}(r(m_1n_1 + m_2n_2))$ flops. This is another reason that the training process can be more efficient under decomposition (3).

We want to emphasize that our experimental results also show that number flops during the training can be significantly decreased by decomposition (3) (e.g., Table 2 shows 97$\%$ reduction in number flops with 1 percentage point performance drop compared to the dense model.)

4 Experiment

4.1 LeNet on MNIST

In this part, we conduct an experiment with the LeNet-5 Network [21] and the MNIST dataset. Similar to the previous part, we compare our algorithm with group LASSO, elastic group LASSO, and unstructured iterative pruning. The results are shown in table1. Since LeNet-5 has three fully connected layers, the column *block size* indicated the block sizes for these layers. For example, (16,8)(8,4)(4,2) indicates that the block size in the first layer is 16 by 8, in the second layer is 8 by 4, in the third layer is 4 by 2. The rank of the decomposition under our algorithm is 5 for all the layers in all the experiments. We can see that the performance and sparsity rate of our method and the baselines are the same. However, the number of FLOPs and the number of training parameters of our algorithms remains significantly lower compared to baselines.

Table 1: Accuracy, sparsity rate, number of training parameters, and number of FLOPs for LeNet-5 network trained on MNIST dataset for different block sizes.

Block-size	Methods	Accuracy	Sparsity Rate	Train Param	FLOPs
(16, 8) (8, 4) (4, 2)	group LASSO	98.31 ± 0.54	49.43 ± 0.06	61k	435.85k
(16, 8) (8, 4) (4, 2)	Elastic group LASSO	98.23 ± 0.60	49.47 ± 1.02	61k	435.85k
(16, 8) (8, 4) (4, 2)	Ours	98.55 ± 0.56	50.85 ± 0.70	6.2k	270.59k
(8, 4) (4, 4) (2, 2)	group LASSO	97.96 ± 0.51	72.97 ± 14.01	61k	435.85k
(8, 4) (4, 4) (2, 2)	Elastic group LASSO	98.02 ± 0.51	63.28 ± 12.89	61k	435.85k
(8, 4) (4, 4) (2, 2)	Ours	99.06 ± 0.52	52.49 ± 1.23	22.6k	287.54k
(4, 4) (4, 4) (2, 2)	group LASSO	98.08 ± 0.60	52.58 ± 4.94	61k	435.85k
(4, 4) (4, 4) (2, 2)	Elastic Group LASSO	98.17 ± 0.55	52.77 ± 5.93	61K	435.85k
(4, 4) (4, 4) (2, 2)	Ours	99.08 ± 0.53	54.02 ± 1.53	13.7k	306.74k
(4, 4) (2, 2) (2, 2)	group LASSO	98.08 ± 0.53	63.29 ± 9.21	61k	435.85k
(4, 4) (2, 2) (2, 2)	Elastic Group LASSO	99.08 ± 0.68	54.30 ± 1.59	61k	435.85k
(4, 4) (2, 2) (2, 2)	Ours	99.08 ± 0.68	54.30 ± 1.59	9.7k	319.34k
(2, 2) (2, 2) (2, 2)	group LASSO	98.27 ± 0.73	49.38 ± 0.02	61k	435.85k
(2, 2) (2, 2) (2, 2)	Elastic group LASSO	97.58 ± 0.60	84.43 ± 8.43	61k	435.85k
(2, 2) (2, 2) (2, 2)	Ours	98.66 ± 0.59	56.27 ± 2.71	6.1k	357.74k
-	iterative pruning	98.02 ± 0.82	58.56 ± 1.32	61k	425.85k

4.2 ViT/Swin-Transformer on CIFAR-100

We conduct another experiment with our approach with the ViT-tiny [5, 43] and Swin-transformer Tiny [27] on CIFAR-100 image classification dataset [12]. The dataset has 60 thousand pictures of 100 different categories. The model is trained using this dataset for 300 epochs. We keep the rank of our algorithm equal to 4. To train the model by group LASSO, we train the network from scratch and increase the regularizer parameters of the group LASSO gradually. Since most of the modules in transformer architecture are linear layers, our method can significantly decrease the number of parameters. It can be seen in Table 2, that our model's number of parameters during the training is only 3% of that of the original model for ViT. Moreover, in the case of the Swin Transformer, we can also achieve 80% compression rate during the training. We want to emphasize that both (Elastic) Group LASSO and our method can achieve high accuracy. However, as we expected, the number of training parameters and FLOPs are significantly smaller under our proposed algorithm. We want to also mention that there is no efficient training algorithm for training block-wise sparse matrices. As a result, other baselines for training block-wise sparse matrices would be more expensive than our proposed method during the training.

Table 2: Experiment with CIFAR-100 dataset. Our method can significantly reduce the training parameters and training FLOPs while maintaining the accuracy close to the original model and (Elastic) Group LASSO method.

Method	Block-size	accuracy	Sparsity Rate	Training Params	FLOPs
ViT-t (Original Model)	-	64.32 ± 1.92	-	5.5M	2.16G
Group LASSO	4×4	60.41 ± 4.24	49.99 ± 0.02	5.5M	2.16G
Elastic group LASSO	4×4	61.92 ± 3.01	49.92 ± 0.11	5.5M	2.16G
Ours	4×4	62.99 ± 0.73	49.64 ± 0.72	0.16M	65.37M
Swin-t(Original Model)	-	81.44 ± 0.05	-	27.60M	26.18G
Group LASSO	4×4	75.87 ± 2.17	50.24 ± 0.13	27.60M	26.18G
Elastic group LASSO	4×4	76.34 ± 0.82	50.19 ± 0.25	27.60M	26.18G
Ours	4×4	77.54 ± 0.42	53.25 ± 0.36	5.3M	167.33M

4.3 Ablation Experiments

Table 3: The impact of rank in decomposition (3) on the model's accuracy, sparsity rate, number of training parameters, and number of FLOPs.

Model	Rank	accuracy	sparsity	Training Params	Training FLOPs
Linear	1	48.40 ± 0.40	53.57 ± 2.43	0.26k	0.56k
Linear	2	66.79 ± 0.91	53.57 ± 1.57	0.46k	1.13k
Linear	4	84.58 ± 3.55	55.36 ± 0.72	0.85k	2.24k
Linear	6	88.19 ± 0.32	51.79 ± 0.56	1.24k	3.36k
ViT-t	1	36.86 ± 2.41	52.20 ± 0.13	0.88M	0.54M
ViT-t	2	59.71 ± 2.63	50.74 ± 0.27	1.22M	1.02M
ViT-t	4	62.99 ± 0.73	49.64 ± 0.72	1.88M	1.88M
Swin-t	1	58.46 ± 0.16	51.39 ± 0.67	3.53M	55.78M
Swin-t	2	68.22 ± 0.04	54.37 ± 1.01	5.25M	108.65M
Swin-t	4	77.54 ± 0.42	53.25 ± 0.36	8.69M	167.33M

In this part, we conduct an experiment to understand the impact of the rank on the accuracy, number of training parameters, and number of training flops during the forward path and backward path. We set the block size equal to 4×4. We conduct ablation experiments on the linear model, ViT-tiny, and Swin Transformer separately. As we expected, we can improve the accuracy of the model by increasing the rank. Moreover, we observe that as the rank increases, the accuracy improvements diminish. In this experiment, we kept the regularizer parameter the same for different ranks. As a result, the sparsity rate is not sensitive to the rank and remains almost the same for different ranks.

5 Conclusion

In this paper, we introduce a novel approach for training block-wise sparse matrices using Kronecker product decomposition. This method offers an alternative to group LASSO and structured pruning, enabling training block-wise sparse matrices with fewer parameters and FLOPs. Our theoretical results show that our proposed method can decrease the number of training parameters and the number of FLOPs without hurting accuracy compared to the group LASSO and structured pruning algorithms. Our experiments demonstrate the effectiveness of our approach in terms of efficiency and accuracy.

6 Acknowledgment

This material is based upon work supported by the U.S. National Science Foundation under award IIS-2301599 and CMMI-2301601, by grants from the Ohio State University's Translational Data Analytics Institute and College of Engineering Strategic Research Initiative.

References

[1] Maximiliana Behnke and Kenneth Heafield. Pruning neural machine translation for speed using group lasso. In *Proceedings of the sixth conference on machine translation*, pages 1074–1086, 2021.

[2] Yaohui Cai, Weizhe Hua, Hongzheng Chen, G. Edward Suh, Christopher De Sa, and Zhiru Zhang. Structured pruning is all you need for pruning cnns at initialization, 2022.

[3] Jang Hyun Cho and Bharath Hariharan. On the efficacy of knowledge distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4794–4802, 2019.

[4] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in neural information processing systems*, 30, 2017.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[6] Ali Edalati, Marzieh Tahaei, Ahmad Rashid, Vahid Partovi Nia, James J Clark, and Mehdi Rezagholizadeh. Kronecker decomposition for gpt compression. *arXiv preprint arXiv:2110.08152*, 2021.

[7] Tommaso Furlanello, Zachary Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. In *International Conference on Machine Learning*, pages 1607–1616. PMLR, 2018.

[8] Marawan Gamal Abdel Hameed, Marzieh S Tahaei, Ali Mosleh, and Vahid Partovi Nia. Convolutional neural network compression through generalized kronecker product decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 771–779, 2022.

[9] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[10] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[11] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[13] Byeongho Heo, Jeesoo Kim, Sangdoo Yun, Hyojin Park, Nojun Kwak, and Jin Young Choi. A comprehensive overhaul of feature distillation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1921–1930, 2019.

[14] Byeongho Heo, Minsik Lee, Sangdoo Yun, and Jin Young Choi. Knowledge transfer via distillation of activation boundaries formed by hidden neurons. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3779–3787, 2019.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[16] Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. Language model compression with weighted low-rank factorization. *arXiv preprint arXiv:2207.00112*, 2022.

[17] Zehao Huang and Naiyan Wang. Like what you like: Knowledge distill via neuron selectivity transfer. *arXiv preprint arXiv:1707.01219*, 2017.

[18] Yasutoshi Ida, Yasuhiro Fujiwara, and Hisashi Kashima. Fast sparse group lasso. *Advances in neural information processing systems*, 32, 2019.

[19] Ameya D. Jagtap, Yeonjong Shin, Kenji Kawaguchi, and George Em Karniadakis. Deep kronecker neural networks: A general framework for neural networks with adaptive activation functions. *Neurocomputing*, 468:165–180, 2022.

[20] Jangho Kim, SeongUk Park, and Nojun Kwak. Paraphrasing complex network: Network compression via factor transfer. *Advances in neural information processing systems*, 31, 2018.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Snip: Single-shot network pruning based on connection sensitivity, 2019.

[23] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets, 2017.

[24] Quanquan Li, Shengying Jin, and Junjie Yan. Mimicking very efficient network for object detection. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 6356–6364, 2017.

[25] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Penksy. Sparse convolutional neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 806–814, 2015.

[26] Liyang Liu, Shilong Zhang, Zhanghui Kuang, Aojun Zhou, Jing-Hao Xue, Xinjiang Wang, Yimin Chen, Wenming Yang, Qingmin Liao, and Wayne Zhang. Group fisher pruning for practical network compression. In *International Conference on Machine Learning*, pages 7021–7032. PMLR, 2021.

[27] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021.

[28] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[29] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[30] Rongrong Ma, Jianyu Miao, Lingfeng Niu, and Peng Zhang. Transformed 1 regularization for learning sparse deep neural networks. *Neural Networks*, 119:286–298, 2019.

[31] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5191–5198, 2020.

[32] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks, 2021.

[33] Oyebade Oyedotun, Djamila Aouada, and Bjorn Ottersten. Structured compression of deep neural networks with debiased elastic group lasso. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2277–2286, 2020.

[34] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news*, 45(2):27–40, 2017.

[35] Wonpyo Park, Dongju Kim, Yan Lu, and Minsu Cho. Relational knowledge distillation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3967–3976, 2019.

[36] Nikhil Rao, Robert Nowak, Christopher Cox, and Timothy Rogers. Classification with the sparse group lasso. *IEEE Transactions on Signal Processing*, 64(2):448–463, 2015.

[37] R. Reed. Pruning algorithms-a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, 1993.

[38] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.

[39] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, June 2017.

[40] Chong Min John Tan and Mehul Motani. Dropnet: Reducing neural network complexity via iterative pruning. In *International Conference on Machine Learning*, pages 9356–9366. PMLR, 2020.

[41] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow, 2020.

[42] Yonglong Tian, Dilip Krishnan, and Phillip Isola. Contrastive representation distillation. *arXiv preprint arXiv:1910.10699*, 2019.

[43] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers amp; distillation through attention. In *International Conference on Machine Learning*, volume 139, pages 10347–10357, July 2021.

[44] Charles F Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1-2):85–100, 2000.

[45] Huan Wang, Qiming Zhang, Yuehai Wang, and Haoji Hu. Structured probabilistic pruning for convolutional neural network acceleration. In *BMVC*, volume 3, page 7, 2018.

[46] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

[47] Sanyou Wu and Long Feng. Sparse kronecker product decomposition: A general framework of signal region detection in image regression, 2022.

[48] Junho Yim, Donggyu Joo, Jihoon Bae, and Junmo Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4133–4141, 2017.

[49] Miao Yin, Huy Phan, Xiao Zang, Siyu Liao, and Bo Yuan. Batude: Budget-aware neural network compression based on tucker decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8874–8882, 2022.

[50] Lei Yu, Xinpeng Li, Youwei Li, Ting Jiang, Qi Wu, Haoqiang Fan, and Shuaicheng Liu. Dipnet: Efficiency distillation and iterative pruning for image super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1692–1701, 2023.

[51] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 67–76, 2017.

[52] Ying Zhang, Tao Xiang, Timothy M Hospedales, and Huchuan Lu. Deep mutual learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4320–4328, 2018.

A Appendix / supplemental material

A.1 Related Work

Pruning is an effective technique for reducing model parameters and has seen tremendous progress in recent years[37, 41, 11]. Pruning generally can be divided into structured pruning [2, 26, 23] and unstructured pruning[4, 38]. Unstructured pruning finds fine-grained sparse matrices by setting the unimportant weights to zero (See Figure 1). While unstructured pruning can decrease the number of model parameters at the inference time, it generally does not improve the inference time [45]. Structured pruning, on the other hand, trains coarse-grained sparse matrices leading to a decrease in the memory cost and inference time. It is important to note that pruning methods generally start with a dense full network and prune the network in one-shot or in several iterations and retrain the network to improve the performance. As a result, memory and computation costs during the training can be expensive. Recently, several pruning methods have been proposed to decrease the training cost as well. These methods try to prune the network right after initialization and train the sparse network [22]. The current Pruning After Initialization (PAI) methods are able to perform unstructured pruning, and they need large memory at the time of initialization which makes it impossible to do training on small devices. On the other hand, our proposed method is able to decrease the number of parameters and flops from the beginning of the initialization and train block-wise sparse matrices to decrease inference memory and time.

Regularization is another method for training a network with sparse weight matrices. It is common to add l1 or l0 regularizer to the loss function [30, 29] to find unstructured sparse weight matrices. Group LASSO, an extension of the LASSO method, is a method that imposes a regularizer to pre-defined groups of model parameters leading to block-wise or group-wise sparsity structures in deep neural networks[25, 39, 1]. To improve the performance of group LASSO, recently a new variation of group LASSO called Elastic group LASSO [33] has been proposed. It is worth mentioning that group LASSO is only able to remove the computation and memory cost during inference. The training cost associated with group LASSO is relatively high as this method starts the training with a dense network.

Matrix/tensor factorization [51, 47] is a compression method that is able to reduce the training and inference cost by reducing the number of training parameters from the beginning of the training process. While matrix/tensor factorization has been widely used for model compression [16, 8, 6, 49], these methods are not able to find block-wise sparse matrices for deep models.

Knowledge Distillation (KD) methods train a smaller student network based on the guidance of a bigger teacher network [15, 24]. Knowledge distillation methods try to make sure that the student network mimics the behavior of teacher network by comparing outputs [3, 7, 31, 52] or intermediate features [13, 14, 17, 20, 35, 42]. The training process under KD can be computationally heavy as we need to train a teacher model first and then use the teacher model to train a student model [48].

In this paper, our goal is to propose a new method for finding block-wise sparse matrices that is efficient during both training and inference. The proposed method can decrease the number of training parameters and flops significantly without degrading the model performance compared to the baselines (e.g., group LASSO, iterative pruning).

A.2 Proofs

Proof [Proposition 1] In this section, we will prove that each block-wise sparse matrix trained by group LASSO could be decomposed by the decomposition in (3).

We assume the block-wise sparse matrix W has a size of m by n, the number of groups is $m_1 \times n_1$ and the block size is m_2 by n_2 where $m_1 \times m_2 = m$ and $n_1 \times n_2 = n$. We want to find S, A_i, B_i such that the following holds where W is given by the group LASSO method.

$$W = \textstyle\sum_{i=1}^{r} (S \odot A_i) \otimes B_i \tag{6}$$

Assume that T groups among $n_1 m_1$ groups are non-zero in matrix W and the index of the groups are $t_1, t_2, ..., t_T$. We set $r = T$ and we generate a series of A_i and B_i for $i = 1, ..., T$ to make sure (6) holds. In particular, we set B_i matrix to be equal to the block of t_i in matrix W, and A_i is a matrix that only has one entry equal to 1 which is associated with block t_i and the other entries are zeros. We also can set S to be a binary matrix. Each entry in S corresponds to a block in W. If a block in W is non-zero (resp. zero), then the entry associated with that block in S would be 1 (resp. 0). By this construction, S, A_i, B_i satisfy (6).

Proof [Proposition 2]

Consider a one layer network withou bias term. The input dimension of it is n. The output dimension of it is m. We can knoe the shape of the weight matrix W is $\mathbb{R}^{m\times n}$. Since we have N data points, the input matrix $X \in \mathbb{R}^{N\times n}$.

Forward FLOPs with full matrix When using full weight matrix, the first step is to compute the output $O \in \mathbb{R}^{N\times m}$ as

$$O = XW^T. \tag{7}$$

The FLOPs of this step is $Nm(2n-1)$. Then we calculate the loss as

$$\mathcal{J} = \|O - Y\|_F^2, \tag{8}$$

where $Y \in \mathbb{R}^{N\times m}$ is the label matrix. The FLOPs for this step is $3Nm - 1$. Therefore, the FLOPs of the forward computation is

$$Nm(2n-1) + (3Nm-1) = \mathcal{O}\left(2Nm(n+1)\right). \tag{9}$$

Backward FLOPs with full matrix In the backward process, we need to calculate the gradient of $\mathcal{J}$ on W. Using chain rule, the first step is to compute

$$\frac{\partial \mathcal{J}}{\partial O} = 2(O - Y). \tag{10}$$

Since $O - Y$ has been calculated in the forward pass, the FLOPs is Nm. The gradient of W is

$$\frac{\partial \mathcal{J}}{\partial W} = \left(\frac{\partial \mathcal{J}}{\partial O}\right)^T X. \tag{11}$$

The FLOPs for this step is $mn(2N-1)$. Therefore, the FLOPs for the backward pass is

$$Nm + mn(2N-1) = \mathcal{O}\left(Nm(2n+1)\right). \tag{12}$$

Forward FLOPs with sparse matrix With Kronecker product decomposition, we replace W by $\sum_{i=1}^r (S \odot A_i) \otimes B_i$. S and $A_i \in \mathbb{R}^{m_1\times n_1}$, $B_i \in \mathbb{R}^{m_2\times n_2}$, where $m_1 m_2 = m$, $n_1 n_2 = n$.

In the forward pass, we need to firstly reshape $X \in \mathbb{R}^{N\times n}$ into $\mathbf{reshape}(X) \in \mathbb{R}^{n_2\times Nn_1}$. Then we calculate $B_i\mathbf{reshape}(X) \in \mathbb{R}^{m_2\times Nn_1}$ with FLOPs $Nn_1m_2(2n_2-1)$. The result is reshape into $\mathbf{reshape}(B_i\mathbf{reshape}(X)) \in \mathbb{R}^{Nm_2\times n_1}$.

Then we calculate $S \odot A_i \in \mathbb{R}^{n_1\times m_1}$ with FLOPs m_1n_1. After this, we get

$$O_i = \mathbf{reshape}(B_i\mathbf{reshape}(X))(S\odot A_i)^T \in \mathbb{R}^{Nm_2\times m_1}, \tag{13}$$

with FLOPs $Nm_1m_2(2n_1-1)$. We denote O as the output of the layer, which is to say

$$O = \sum_{i=1}^r O_i = \sum_{i=1}^r \mathbf{reshape}(B_i\mathbf{reshape}(X))(S\odot A_i)^T. \tag{14}$$

The total FLOPs to get O is

$$r(Nm_1m_2(2n_1-1) + m_1n_1 + Nn_1m_2(2n_2-1)) + (r-1)Nm \tag{15}$$

Then we reshape O in to $\mathbf{reshape}(O) \in \mathbb{R}^{N\times m}$. The loss is calculated as

$$\mathcal{J} = \|\mathbf{reshape}(O) - Y\|_F^2 \tag{16}$$

The FLOPs for this step is $3Nm - 1$. Therefore the FLOPs of the forward computation is

$$\begin{aligned} & r(2Nm_1m_2n_1 - Nm_1m_2 + m_1n_1 + 2Nm_1n_1n_2 - Nm_2n_1) + (r-1)Nm + 3Nm - 1 \\ = & \mathcal{O}\left(2Nrm_1n_1(m_2+n_2) - Nr(m + 2m_2n_1) + 3Nm\right) \end{aligned} \tag{17}$$

Backward FLOPs with sparse matrix In the backward process, we need to calculate the gradient of $\mathcal{J}$ on S, A_i and B_i. Using chain rule, the first step is to compute

$$\frac{\partial \mathcal{J}}{\partial \mathbf{reshape}(O)} = 2(\mathbf{reshape}(O) - Y). \tag{18}$$

Since $\mathbf{reshape}(O) - Y$ has been calculated in the forward pass, the FLOPs is Nm. Then we reshape it into $\frac{\partial \mathcal{J}}{\partial O} \in \mathbb{R}^{Nm_2 \times m_1}$. Then we can get

$$\frac{\partial \mathcal{J}}{\partial (S \odot A_i)} = \left(\frac{\partial \mathcal{J}}{\partial O}\right)^T \mathbf{reshape}(B_i \mathbf{reshape}(X)). \tag{19}$$

Since $\mathbf{reshape}(B_i \mathbf{reshape}(X))$ has been obtained in the forward pass, the FLOPs for this step is $m_1 n_1 (2Nm_2 - 1)$. To get the gradient on S and A_i, we have

$$\frac{\partial \mathcal{J}}{\partial S} = \sum_{i=1}^{r} \frac{\partial \mathcal{J}}{\partial (S \odot A_i)} \odot A_i, \tag{20}$$

with FLOPs $rm_1 n_1 + (r-1) m_1 n_1$, and

$$\frac{\partial \mathcal{J}}{\partial A_i} = \frac{\partial \mathcal{J}}{\partial (S \odot A_i)} \odot S, \tag{21}$$

with FLOPs $m_1 n_1$. The gradient on $\mathbf{reshape}(B_i \mathbf{reshape}(X))$ is

$$\frac{\partial \mathcal{J}}{\partial \, \mathbf{reshape}(B_i \mathbf{reshape}(X))} = \frac{\partial \mathcal{J}}{\partial O} (S \odot A_i). \tag{22}$$

The FLOPs for this step is $Nm_2 n_1 (2m_1 - 1)$. We reshape the gradient into $\frac{\partial \mathcal{J}}{\partial \, B_i \mathbf{reshape}(X)} \in \mathbb{R}^{m_2 \times N n_1}$. So, we can get the gradient on B_i as

$$\frac{\partial \mathcal{J}}{\partial B_i} = \frac{\partial \mathcal{J}}{\partial \, B_i \mathbf{reshape}(X)} \mathbf{reshape}(X)^T, \tag{23}$$

with FLOPs of $m_2 n_2 (2Nn_1 - 1)$. Therefore, we can get the total FLOPs for the backward pass as

$$\begin{aligned} & Nm + rm_1 n_1 (2Nm_2 - 1) + rm_1 n_1 + (r-1) m_1 n_1 + rm_1 n_1 + rNm_2 n_1 (2m_1 - 1) + rm_2 n_2 (2Nn_1 - 1) \\ =& \mathcal{O}\left(Nm + Nr(4m_1 m_2 n_1 - m_2 n_1 + 2m_2 n_1 n_2)\right) \end{aligned} \tag{24}$$

B Computation resource

we used a server with 64 CPUs of AMD EPYC 7313 16-Core Processor. The server has 8 RTX A5000 GPUs, with 24GB memory for each one. For the experiment with linear model and LeNet, we used only one single GPU. And for the ViT-tiny experiment, we use 2 GPUs at the same time.