Compute Better Spent: Replacing Dense Layers with Structured Matrices

Shikai Qiu [*1] **Andres Potapczynski** [*1] **Marc Finzi** [2] **Micah Goldblum** [1] **Andrew Gordon Wilson** [1]

Abstract

Dense linear layers are the dominant computational bottleneck in foundation models. Identifying more efficient alternatives to dense matrices has enormous potential for building more compute-efficient models, as exemplified by the success of convolutional networks in the image domain. In this work, we systematically explore structured matrices as replacements for dense matrices. We show that different structures often require drastically different initialization scales and learning rates, which are crucial to performance, especially as models scale. Using insights from the Maximal Update Parameterization, we determine the optimal scaling for initialization and learning rates of these unconventional layers. Finally, we measure the scaling laws of different structures to compare how quickly their performance improves with compute. We propose a novel matrix family containing Monarch matrices, the Block Tensor-Train (BTT), which we show performs better than dense matrices for the same compute on multiple tasks. On CIFAR-10/100 with augmentation, BTT achieves exponentially lower training loss than dense when training MLPs and ViTs. BTT matches dense ViT-S/32 performance on ImageNet-1k with 3.8 times less compute and is more efficient than dense for training small GPT-2 language models.

1. Introduction

Regardless of their architectures, most neural networks consist of interleaved linear layers and simple non-linearities. In large foundation models such as GPT-3 (Brown et al., 2020), these linear layers consume the vast majority of the parameters and computation (Kaplan et al., 2020), and are primarily represented by dense matrices. Substituting these dense matrices with structured matrices with fast matrix-vector multiplies (MVMs) has the potential to significantly improve the computational efficiency of these models. Unfortunately, there often isn't an obvious algebraic structure to exploit in the linear layers of such models, which process end-to-end learned token embeddings rather than objects with clear structures like images (Vaswani et al., 2017).

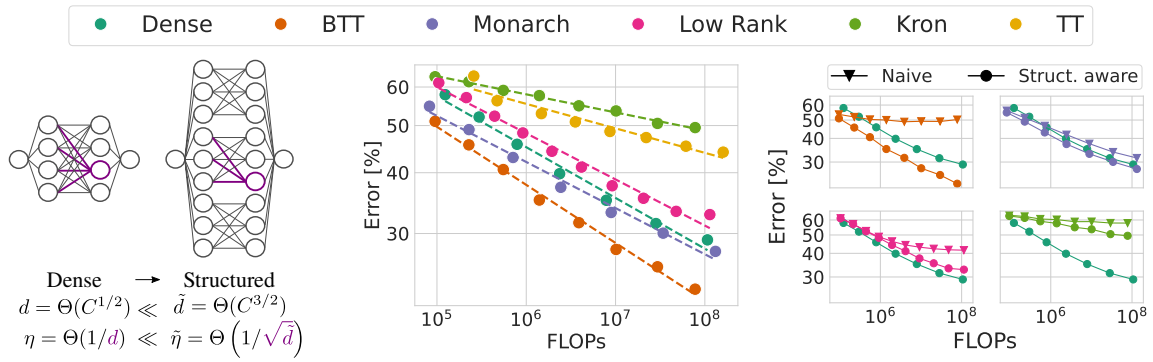
Structured matrices, however, are not limited to encoding domain-specific inductive biases. They can also offer advantages over dense matrices by enabling different allocations of the same computational budget. For example, a structured layer can be much wider than a dense layer given the same number of parameters and compute. The compute cost C of an MVM is $\mathcal{O}(d^2)$ for a $d \times d$ dense matrix, but only $\mathcal{O}(d^{3/2})$ for a block diagonal matrix with $\sqrt{d}$ blocks. Consequently, given the same compute C, the width can be at most $\mathcal{O}(C^{1/2})$ for a dense layer, but $\mathcal{O}(C^{2/3})$ for such a block diagonal layer. We can replace a dense layer of width 1024 with a $10\times$ wider block diagonal layer, as illustrated in Figure 1a. Both layers have the same number of parameters and compute costs, but a larger width enables the model to potentially store more information in its activations and use more non-linearities to model complex functions. In this light, structured matrices do not merely approximate dense matrices but enable different ways of scaling up the models with compute that make them potentially more expressive.

To study how structured layers compare against dense layers as a function of compute, we will compare their scaling laws: how compute translates to performance as the models scale up. Across domains such as language, image, and video modeling, the loss or error rate E of a well-trained neural network has shown to be highly predictable as a function of the compute C required by the model, often well-described by a power law $E \propto C^{-\alpha}$ when data is not a bottleneck (Kaplan et al., 2020; Sharma & Kaplan, 2022; Hoffmann et al., 2022). If structured layers can achieve better scaling laws, they will outperform dense layers at scale, delivering exponentially better performance per unit compute if they can improve the scaling exponent α.

In this work, we systematically study whether structured

[*]Equal contribution [1]New York University [2]Carnegie Mellon University. Correspondence to: Shikai Qiu <sq2129@nyu.edu>, Andres Potapczynski <ap6604@nyu.edu>, Marc Finzi <maf820@nyu.edu>, Micah Goldblum <goldblum@nyu.edu>, Andrew Gordon Wilson <andrewgw@cims.nyu.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).



(a) Same compute, wider layers (b) CIFAR-10 train error (c) Structure-aware LR is crucial

Figure 1. **Controlling for compute, replacing dense layers with structured matrices enables wider models and can lead to better performance.** (a) A neural network with structured matrices can be made much wider, but its learning rate needs to be scaled differently as a function of width since not all connections are present (Section 3). The width d of a dense layer scales as $C^{1/2}$ where C is the compute per forward pass, while the width $\tilde{d}$ of a block diagonal layer is exponentially larger, scaling as $C^{2/3}$. The optimal learning rate η of the dense layer and $\tilde{\eta}$ of the block diagonal layer scales differently as d^{-1} and $\tilde{d}^{-1/2}$ respectively. (b) Structured matrices can improve the training error scaling laws of MLPs on CIFAR-10 with data augmentation (Section 4). (c) Scaling the learning rate in a structure-aware fashion (●) is crucial for performance (Section 3), without which the benefit of structured layers does not emerge (▼).

matrices can have better scaling laws than dense matrices, without relying on domain-specific algebraic structures so that our findings can apply to training foundation models broadly.

- We show that structured layers often require drastically different learning rates and initialization scales compared to their dense counterparts, because their underlying trainable parameter matrices tend to be much smaller in size than the width of the layer (Figure 1a). Naively using dense layer learning rates, structured layers often significantly underperform dense layers, as shown in Figure 1c.

- Leveraging insights from μP (Yang et al., 2023a) on how to optimally scale the initialization and learning rates for dense layers as a function of width, we show how to automatically determine the appropriate initialization and learning rate scales for structured linear layers. This structure-aware technique enables us to effectively train and scale a wide range of structured layers without additional tuning.

- We measure scaling laws for neural networks employing structured matrices as they scale, showing that structured layers can have better scaling exponents than dense matrices on some tasks. These results suggest that the scaling exponents are not necessarily determined solely by the task as previously hypothesized (Bahri et al., 2021; Michaud et al., 2023).

- We identify matching parameter count to FLOPs[1] as a principle shared by the best-performing structures. Conversely, commonly used structures such as the Kronecker product and Tensor-Train decomposition violate this principle and underperform dense matrices in our experiments. Adhering to this principle can serve as important guidance for future work on designing more efficient linear layers.

- We introduce Block Tensor-Train (BTT) as a new family of expressive structured matrices, containing the Monarch matrices (Dao et al., 2022) as a special case. The BTT family has better scaling laws than dense matrices on multiple tasks. On CIFAR-10/100 with augmentation, BTT achieves exponentially lower training loss than dense when training MLPs and ViTs. On ImageNet-1k, BTT matches dense ViT-S/32 performance with 3.8 times less compute.

- We study divergences in training transformers with BTT layers, showing that weight normalization is required to avoid divergence due to unbounded growth of the activation.

We make our code available available here. We use the `Linear Operator` abstractions in CoLA (Potapczynski et al., 2024) to prototype and compute efficient MVMs for structured matrices.

2. Structured Alternatives to Dense Layers

We now introduce the types of structured matrices we consider in this work. We review their computational properties and modeling assumptions, summarized in Table 1. Without loss of generality, we consider $d \times d$ square matrices for notational simplicity.

Low-rank. A low-rank matrix can be parameterized as $\mathbf{W} = \mathbf{UV}$ where $\mathbf{U} \in \mathbb{R}^{d \times r}$, $\mathbf{V} \in \mathbb{R}^{r \times d}$ and $r \leq d$ is its rank. It has $2rd$ parameters and its MVM costs $2rd$

[1] Here and elsewhere in the paper we use the more familiar term FLOPs as a stand-in for MACs (Multiply-Accumulate) operations to highlight when they match the number of parameters, even though 1 MAC is technically 2 FLOPs.

Structure	MVM FLOPs	# Params	Modeling assumptions	Example applications
Dense	d^2	d^2	General linear maps	MLPs, Transformers
Low-Rank	$2rd$	$2rd$	Compression	Bottleneck layers, Linear attention
Convolution	pd	p	Translation equivariance	Images, Time-series
Kronecker	$2d^{3/2}$	$2d$	Sets, Graphs, Grids	GPs, Deep Sets, Attention, GNNs
Monarch	$2d^2/b$	$2d^2/b$	Flexible	Compute-efficient linear layers
TT	$2rd^{3/2}$	$2rd$	Subsystems, Local interactions	Hidden Markov Models, Spin systems
BTT	$2rd^{3/2}$	$2rd^{3/2}$	Flexible	Compute-efficient linear layers

Table 1. **Overview of the computational properties, modeling assumptions, and applications of structured matrices we consider.** Some structures require the same FLOPs as parameters for a matrix multiply, while others require more FLOPs. d is the size of the matrix, r is the rank in low-rank, TT, and BTT, p is the kernel size in a convolution, and b is the number of blocks in Monarch. We assume 2 cores each of size $\sqrt{d}$ for Kronecekr, TT and BTT.

FLOPs. By first performing a dimension reduction on the input via $\mathbf{V}$, a low-rank matrix assumes that only a subspace of the input space is relevant to the task and is natural for compression (Zhao et al., 2024; Wang et al., 2020).

Convolution. Convolutions, or Toeplitz matrices, naturally model systems with translational symmetries such as images (LeCun et al., 1998a; Krizhevsky et al., 2012; He et al., 2015b) and time-series (Wilson & Adams, 2013). A convolution with kernel size p has p parameters and requires $\mathcal{O}(pd)$ FLOPs. Each parameter is used $\mathcal{O}(d)$ times in a convolution to impose translational symmetry. Alternatively, the Fast Fourier transform allows the convolution to be computed in $\mathcal{O}(d \log d)$ FLOPs.

Kronecker. Kronecker product structure naturally arises in applications with structured data (Perez et al., 2017; Titsias, 2009; Maron et al., 2020; Saatçi, 2012; Wilson & Nickisch, 2015). A Kroncker product $\mathbf{W} = \mathbf{L} \otimes \mathbf{R}$ with $\mathbf{L} \in \mathbb{R}^{d_1 \times d_1}$, $\mathbf{R} \in \mathbb{R}^{d_2 \times d_2}$, $d = d_1 \cdot d_2$, specifies a matrix whose MVM $\mathbf{y} = \mathbf{W}\mathbf{x}$ can be efficiently computed as $y_{\alpha\beta} = \sum_\gamma L_{\alpha\gamma} \sum_\delta R_{\beta\delta} x_{\gamma\delta}$, after reshaping the input $\mathbf{x}$ in row-major order into a $d_1 \times d_2$ matrix and followed by flattening $\mathbf{y}$ back to a vector. Assuming $d_1 = d_2 = \sqrt{d}$, $\mathbf{W}$ has $2d$ parameters and requires $2d^{3/2}$ FLOPs for an MVM. The Kronecker product uses each parameter $\sqrt{d}$ times, which can be made explicit by interpreting $\sum_\delta R_{\beta\delta} x_{\gamma\delta}$ (the same argument applies to the sum involving $\mathbf{L}$) as multiplying the vector $\mathbf{x}$ by a block-diagonal matrix $\bigoplus_{\gamma=1}^{\sqrt{d}} \mathbf{R}_\gamma$, where all the blocks $\mathbf{R}_\gamma \in \mathbb{R}^{\sqrt{d} \times \sqrt{d}}$ are shared: $\mathbf{R}_\gamma = \mathbf{R}, \gamma = 1, \ldots, \sqrt{d}$. This parameter-sharing naturally corresponds to the assumption that the input $\mathbf{x}$ represents a set of objects of the same kind, such as nodes in a graph (Kipf & Welling, 2016), patches of an image (Tolstikhin et al., 2021), points on a grid (Saatçi, 2012), or words in a sentence (Vaswani et al., 2017; Elhage et al., 2021).

Monarch. Introduced in Dao et al. (2022), a Monarch matrix is defined as the product $\mathbf{PLP}^\top\mathbf{R}$ where $\mathbf{P}$ is a row-major to column-major permutation and $\mathbf{L}, \mathbf{R}$ are two block-diagonal matrices: $\bigoplus_{\beta=1}^{\sqrt{d}} \mathbf{L}_\beta, \bigoplus_{\gamma=1}^{\sqrt{d}} \mathbf{R}_\gamma$. Monarch requires $2d^{3/2}$ FLOPs for an MVM and has $2d^{3/2}$ parameters. The efficient multiply for Monarch can be written as $y_{\alpha\beta} = \sum_\gamma L_{\alpha\beta\gamma} \sum_\delta R_{\beta\gamma\delta} x_{\gamma\delta}$, where $R_{\beta\gamma\delta} = (\mathbf{R}_\gamma)_{\beta\delta}$ and $L_{\alpha\beta\gamma} = (\mathbf{L}_\beta)_{\alpha\gamma}$ and we have colored the block dimensions β, γ. Monarch can be viewed as a relaxation of the Kronecker product where parameters that were shared across the block dimensions are now made independent. Monarch matrices do not make strong assumptions about the structure of the input. In practice, the number of blocks b in $\mathbf{L}$ and $\mathbf{R}$ are often chosen to be much less than $\sqrt{d}$ to reduce sparsity (Dao et al., 2022; Fu et al., 2023). In this case, Monarch has $2d^2/b$ parameters and requires $2d^2/b$ FLOPs for an MVM.

Tensor-Train. The Tensor-Train (TT) decomposition (Oseledets, 2011) specifies a set of c cores $\mathbf{G}^{(i)} \in \mathbb{R}^{r_i \times m_i \times n_i \times r_{i-1}}$ for $i = 1, \ldots, c$ where $d = \prod_i m_i = \prod_i n_i$, $r_i \in \mathbb{N}$ and $r_0 = r_c = 1$. For ease of notation, we will focus on $c = 2$ with $m_1 = m_2 = n_1 = n_2 = \sqrt{d}, r_1 = r$, $\mathbf{G}^{(1)} = \mathbf{R} \in \mathbb{R}^{r \times \sqrt{d} \times \sqrt{d}}, \mathbf{G}^{(2)} = \mathbf{L} \in \mathbb{R}^{\sqrt{d} \times \sqrt{d} \times r}$, though we present the general case in Appendix C. With the input and output as reshaped as $\sqrt{d} \times \sqrt{d}$ matrices, a TT matrix is equivalent to a sum over r Kronecker products indexed by $\sigma = 1, \ldots, r$:

$$y_{\alpha\beta} = \sum_{\gamma\sigma} L_{\alpha\gamma\sigma} \sum_\delta R_{\sigma\beta\delta} x_{\gamma\delta}. \tag{1}$$

By increasing r, referred to as the TT-rank, TT becomes more expressive relative to the Kronecker product. When $r = d$, it can represent any $d \times d$ dense matrix. TT has $2rd$ parameters and costs $2rd^{3/2}$ FLOPs for an MVM. Like Kronecker, TT shares parameters along the block dimensions β, γ and therefore uses each parameter $\sqrt{d}$ times in an MVM. The TT structure is natural for modeling systems that decompose into subsystems with local pairwise interactions, such as quantum spin chains and hidden Markov

models (Fannes et al., 1992; Critch et al., 2014).

Block Tensor-Train. We propose a novel family of structured matrices called Block Tensor-Train (BTT) matrices, by removing the parameter-sharing along the block dimensions β, γ in the TT structure. In the two core ($c = 2$) case, a BTT matrix of BTT-rank r is defined by two parameter tensors $\mathbf{R} \in \mathbb{R}^{r\times\sqrt{d}\times\sqrt{d}\times\sqrt{d}}$ and $\mathbf{L} \in \mathbb{R}^{\sqrt{d}\times\sqrt{d}\times\sqrt{d}\times r}$. Its MVM is given by

$$y_{\alpha\beta} = \sum_{\gamma\sigma} L_{\alpha\beta\gamma\sigma} \sum_{\delta} R_{\sigma\beta\gamma\delta} x_{\gamma\delta}. \tag{2}$$

In Appendix C, we study the expressiveness of BTT, present a simple algorithm for projection onto the BTT family, and show BTT with rank $r = \sqrt{d}$ can represent any dense matrix (in constrast to $r = d$ for TT) when $c = 2$ and analogous results for $c > 2$. Therefore, by varying the BTT rank, we effectively interpolate between Monarch matrices and dense matrices.

We use the `Linear Operator` abstractions available in `CoLA` (Potapczynski et al., 2024) to compute MVMs for these structures efficiently. In Appendix B, we show the structures we consider have asymptotically the same MVM runtimes as dense matrices as a function of FLOPs because they can be implemented through the same dense matrix multiply primitives, though they introduce non-trivial overhead for small matrix sizes with our current implementation.

3. Optimizing Structured Matrices

To study the performance and scaling laws of unconventional layers, we must determine how to optimize them effectively by choosing appropriate initialization and learning rates as the models scale. As Figure 1c illustrates, the optimal settings for structured matrices can differ significantly from dense matrices. We develop a technique based on the Maximal Update Parameterization (μP) (Yang & Hu, 2021; Yang & Littwin, 2023; Yang et al., 2021) to automatically determine the optimal initialization and learning rate scaling for a generic structured layer given its structure and size, enabling us to train and scale various structured layers with good hyperparameters and minimal tuning. We focus on the Adam optimizer (Diederik P. Kingma, 2015) but discuss extensions to other optimizers in Appendix H.

3.1. Maximal Update Parameterization

The Maximal Update Parameterization (μP) (Yang & Hu, 2021; Yang & Littwin, 2023; Yang et al., 2021) specifies how to scale the initialization and learning rate of neural networks as their widths increase while maximizing feature learning in every layer (Yang & Hu, 2021). Yang et al. (2023a) provides an elementary derivation based on the spectral norm, which we now review.

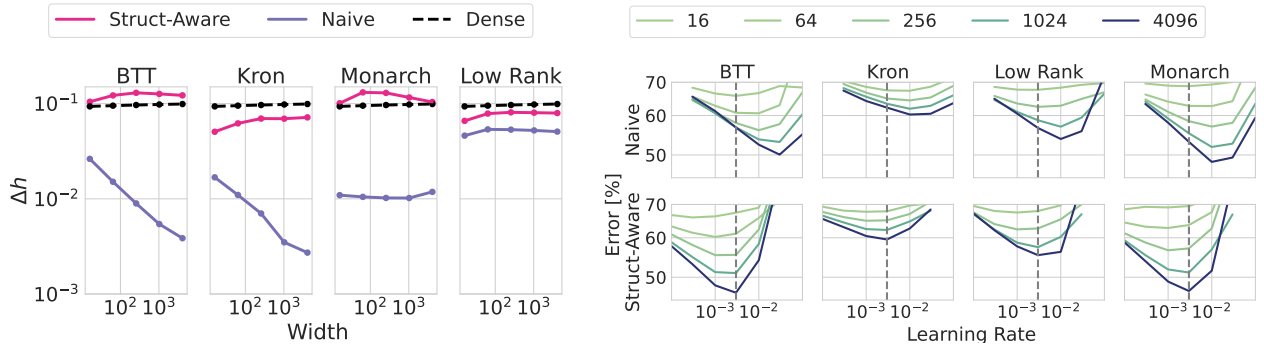
In μP, initialization and learning rates are chosen so that entries of each layer's output have size $\Theta(1)$ and are updated at a rate of $\Theta(1)$ per step throughout training. Here, big-Θ notation denotes scaling in the layer's width, omitting dependence on other quantities. If these conditions do not hold, the layer's output or update will either diverge or vanish for sufficiently large widths. For a dense matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}$, input $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$, output $\mathbf{h} = \mathbf{W}\mathbf{x} \in \mathbb{R}^{d_{\text{out}}}$, and output update $\Delta\mathbf{h} = \Delta\mathbf{W}\mathbf{x}$ due to a weight update $\Delta\mathbf{W}$, μP requires $\|\mathbf{h}\|_2 = \Theta(\sqrt{d_{\text{out}}})$ and $\|\Delta\mathbf{h}\|_2 = \Theta(\sqrt{d_{\text{out}}})$. During training, gradient descent aligns $\mathbf{x}$ with the top singular subspace of $\mathbf{W}$ and $\Delta\mathbf{W}$ (Yang et al., 2023a; Yang & Littwin, 2023), so $\|\mathbf{h}\|_2 = \Theta(\|\mathbf{W}\|_2\|\mathbf{x}\|_2)$ and $\|\Delta\mathbf{h}\|_2 = \Theta(\|\Delta\mathbf{W}\|_2\|\mathbf{x}\|_2)$. Assuming $\mathbf{x}$ is entry-wise $\Theta(1)$, we want $\|\mathbf{W}\|_2 = \Theta(\sqrt{d_{\text{out}}/d_{\text{in}}})$ and $\|\Delta\mathbf{W}\|_2 = \Theta(\sqrt{d_{\text{out}}/d_{\text{in}}})$. To ensure the desired spectral norm at initialization, entries of $\mathbf{W}$ are drawn from $\mathcal{N}(0, \sigma^2)$ with $\sigma = \Theta(\sqrt{\min(d_{\text{in}}, d_{\text{out}})/d_{\text{in}}^2})$. For the updates, the gradient $\nabla_{\mathbf{W}}\mathcal{L} = \frac{1}{B}\sum_{i=1}^{B} \nabla_{\mathbf{h}_i}\mathcal{L} \cdot \mathbf{x}_i^\top$ has $\Theta(1)$ stable rank, assuming the batch size B is constant, so its spectral norm scales the same way as its Frobenius norm. Since Adam normalizes the gradient to be entry-wise $\Theta(1)$, the normalized gradient has Frobenius norm $\Theta(\sqrt{d_{\text{in}}d_{\text{out}}})$. Therefore, an Adam learning rate of $\Theta(1/d_{\text{in}})$ ensures the desired spectral norm.

Once the optimal learning rate η^* is found for a particular width d_{in}, it can be transferred to any other width d'_{in} by setting the new learning rate as $\eta^* \cdot \frac{d_{\text{in}}}{d'_{\text{in}}}$, assuming d_{in} and d'_{in} are sufficiently large (Yang et al., 2021). For architectures, μP deviates from conventional initializations mainly in the last layer, where $\sigma = \Theta(1/d_{\text{in}})$ according to μP but $\sigma = \Theta(\sqrt{1/d_{\text{in}}})$ according to more conventional strategies (LeCun et al., 2002; Glorot & Bengio, 2010; He et al., 2015a).

3.2. Identifying μP for Structured Matrices

The above scaling of learning rate and initialization assume dense matrices and don't immediately carry over to arbitrarily structured matrices. For example, for a Kronecker product $\mathbf{W} = \mathbf{L} \otimes \mathbf{R}$ where $\mathbf{W} \in \mathbb{R}^{d\times d}$ and $\mathbf{L}, \mathbf{R} \in \mathbb{R}^{\sqrt{d}\times\sqrt{d}}$, one intuitively expects that the optimal learning rates for parameters $\mathbf{L}$ and $\mathbf{R}$ in this layer to scale as $\Theta(1/\sqrt{d})$, the size of the actual learnable parameter matrices, rather than naively as $\Theta(1/d)$ based only on the width of the layer.

Since many structured matrices are ultimately compositions of smaller dense matrices and fixed, norm-preserving linear transformations (e.g. reshapes), as exemplified in Section 2, we can decompose the problem by applying the same spectral considerations to each dense component separately, effectively treating each structured layer as a deep linear network. Suppose the MVM $\mathbf{W}\mathbf{x}$ can be computed as $\mathbf{W}\mathbf{x} = \mathbf{G}_k\mathbf{P}_k \ldots \mathbf{G}_1\mathbf{P}_1\mathbf{x}$ where each $\mathbf{P}_i$ is a fixed, norm-



(a) Stable feature learning

(b) Stable optimal learning rate

Figure 2. **Structure-aware learning rate scaling results in stable feature learning and stable optimal learning rate as we vary the structure and model size**. (a) The RMS of the changes Δh of the last layer features is stable as the models are scaled up in width, but is smaller or vanishes if we naively use the learning rate for the dense model. (b) The optimal learning rate is stable as we vary the structure and width, provided we use structure-aware learning rates. Here we use Monarch with 16 blocks.

preserving linear transformation, such as the product of a permutation and a reshape, and multiplication by $\mathbf{G}_i$ denotes a batched MVM, i.e., $(\mathbf{G}_i\mathbf{x})_{b\mu} = \sum_\nu (G_i)_{b\mu\nu} x_{b\nu}$ for some dense tensor $\mathbf{G}_i \in \mathbb{R}^{B_i \times d^i_{\text{out}} \times d^i_{\text{in}}}$, where b is an abstract batch-like dimension. Then to ensure that the activations have size $\Theta(1)$ and all parameters are updated as much as possible to maximize feature learning (Yang et al., 2023a), we require the initialization and updates to each slice $(\mathbf{G}_i)_b \in \mathbb{R}^{d^i_{\text{out}} \times d^i_{\text{in}}}$ of $\mathbf{G}_i$ to have $\Theta\left(\sqrt{d^i_{\text{out}}/d^i_{\text{in}}}\right)$ spectral norm. Thus we initialize each $\mathbf{G}_i$ with standard deviation $\Theta\left(\sqrt{\min(d^i_{\text{in}}, d^i_{\text{out}})/(d^i_{\text{in}})^2}\right)$ and set its Adam learning rate as $\Theta\left(1/d^i_{\text{in}}\right)$. When used in the last linear layer in a residual block, we zero-initialize the last component $\mathbf{G}_k$, which is compatible with μP by setting the hidden constant in $\Theta(\cdot)$ to 0 (Yang et al., 2021).

Transferring learning rate between structures. Once the optimal learning rate η^* is known for a $d_{\text{out}} \times d_{\text{in}}$ dense layer, we can infer the optimal learning rate η^*_i of each component $\mathbf{G}_i$ of the corresponding structured layer as $\eta^*_i = \kappa_i \cdot \eta^*$, where $\kappa_i = \frac{d_{\text{in}}}{d^i_{\text{in}}} \cdot \delta_i$ for some constant δ_i. Here $\frac{d_{\text{in}}}{d^i_{\text{in}}}$ accounts for the $\Theta(\text{width}^{-1})$ scaling of optimal learning rate prescribed by μP, with width identified with d_{in} and d^i_{in} respectively for the dense matrix and $\mathbf{G}_i$, and δ_i accounts for potential differences in the constants omitted by $\Theta(\cdot)$ for the dense matrix and $\mathbf{G}_i$. While the precise value of δ_i is not theoretically determined by μP, we adopt the heuristic $\delta_i = 1/k$ where k is the number of learnable dense components so that the overall updates to the output of this layer is roughly preserved, since $\Delta\mathbf{h}$ has k leading order terms:

$$\Delta\mathbf{h} = \sum_{i=1}^{k} \mathbf{G}_k\mathbf{P}_k \ldots \Delta\mathbf{G}_i\mathbf{P}_i \ldots \mathbf{G}_1\mathbf{P}_1\mathbf{x} + \mathcal{O}(\Delta\mathbf{G}^2). \tag{3}$$

δ_i can be further tuned empirically around $1/k$ to maximize performance, though we will show the $1/k$ heuristic is sufficently good in practice.

As $\mathbf{G}_i$'s are often much smaller in size than the matrix $\mathbf{W}$ it parameterizes, the required learning rate multiplier κ_i is often a large number. For example, suppose we initially represent $\mathbf{W} \in \mathbb{R}^{d\times d}$ as a dense matrix and find η is an effective learning rate during training. If we now instead represent $\mathbf{W} = \mathbf{L} \otimes \mathbf{R}$ where $\mathbf{L}, \mathbf{R} \in \mathbb{R}^{\sqrt{d}\times\sqrt{d}}$, we would then need to scale up the learning rate for both $\mathbf{L}, \mathbf{R}$ by a factor of $\Theta(\sqrt{d})$, which grows arbitrarily large for large d. We show the Adam learning rate multipliers required for various structures in Table 2, adopting our heuristic of $\delta_i = 1/k$.

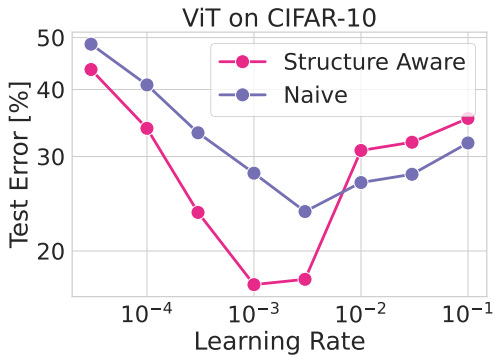


Figure 3. **Structure-aware learning rates improve performance even after tuning the learning with grid search.** Test error of ViT ($d = 1024$) on CIFAR-10 where the feed-forward layers are replaced using BTT.

3.3. Empirical Validation

We now empirically validate the effectiveness of our structure-aware learning rate scaling. We compare it to the naive, structure-agnostic approach that parameterizes the

learning rate η_i for each parameter tensor $\mathbf{G}_i$ in a $d_{\text{out}} \times d_{\text{in}}$ structured layer as $\eta_i = \eta_0 \frac{d_0}{d_{\text{in}}} \propto 1/d_{\text{in}}$, where the base learning rate η_0 and the base width d_0 are constants, corresponding to scaling the learning rate optimally according to μP if the layer were dense. The structure-aware approach additionally applies the structure-dependent learning rate multipliers κ_i in Table 2 so that $\eta_i = \eta_0 \frac{d_0}{d_{\text{in}}} \kappa_i$. We use $d_0 = 64$ throughout this section.

Stable feature learning. We train an MLP with 2 hidden layers without bias on CIFAR-10 with width $d \in \{16, 64, 256, 1024, 4096\}$ and a base learning rate $\eta_0 = 3 \cdot 10^{-3}$. For a given width, we track the root mean square (RMS) of $\Delta \mathbf{h}_t = \mathbf{h}_{t+1} - \mathbf{h}_t$ at every step t, where $\mathbf{h}_t \in \mathbb{R}^d$ is the activation of the last layer before the classification head. We then plot the the average RMS over 500 steps for different widths and structures. As seen in Figure 2, structure-aware learning rate scaling produces consistent feature learning for all structures used with no tuning. In contrast, the naive approach causes much smaller or vanishing updates to the features. The effect is most pronounced for BTT and Kronecker, for which κ_i grows without bound for both $\mathbf{L}$ and $\mathbf{R}$ as the width increases.

Stable optimal learning rate. We test if the structure-aware learning scaling preserves the learning rate landscape for all structures so that once an optimal learning rate is found for the dense model with some width, it can be directly transferred to all other structures and widths. We train a 2-layer MLP on CIFAR-10 with augmentation (see Section 4 for details) for 100 epochs, using a base learning rate of $3 \cdot 10^{-3}$, the optimal value for a dense model at with $d_0 = 64$. In the first row of Figure 2b, we show the train error as a function of the base learning rate η_0 when scaled to other widths and structures using the naive approach, which is optimal for the dense model but clearly not for the other structures. By contrast, in the second row, the structure-aware approach approximately stabilizes the learning rate landscape across structures and widths, significantly reducing the cost for exploring different structures. Slight deviation at small widths is expected because the optimality of μP relies on convergence to the infinite-width limit (Yang & Hu, 2021).

Improved performance even after tuning. Finally, we show in Figure 1c the performance of structured models quickly saturate as they are scaled up without structure-aware learning rates. Monarch is an exception, for which the multipliers in Table 2 are closer to 1 because we use $b = 4$. In this case, the learning rate multiplier required for Monarch is only 2 and independent of scale, which may explain why Dao et al. (2022) still achieves good performance with Monarch by reusing the dense learning rates.

Furthermore, the structure-aware approach not only reduces the tuning cost for structured layers, but is necessary for optimal performance if the structures differ across layers, even when we perform a grid search over the base learning rate η_0. Consider a transformer of hidden dimension d where only the feed-forward layers (FFN) are replaced with BTT and the attention projection matrices are dense. Since the optimal learning rate is $\Theta(1/\sqrt{d})$ for the FFN layer but $\Theta(1/d)$ for the attention projection, the naive approach would have to choose between using a learning rate too large for the attention projection or a learning rate too small for the FFN, whereas the structure-aware approach does not have this problem. In Figure 3, we show that for a ViT with BTT-structured FFNs, the structure-aware approach indeed achieves much better performance even if we tune the base learning rate.

4. Scaling Laws of Structured Matrices

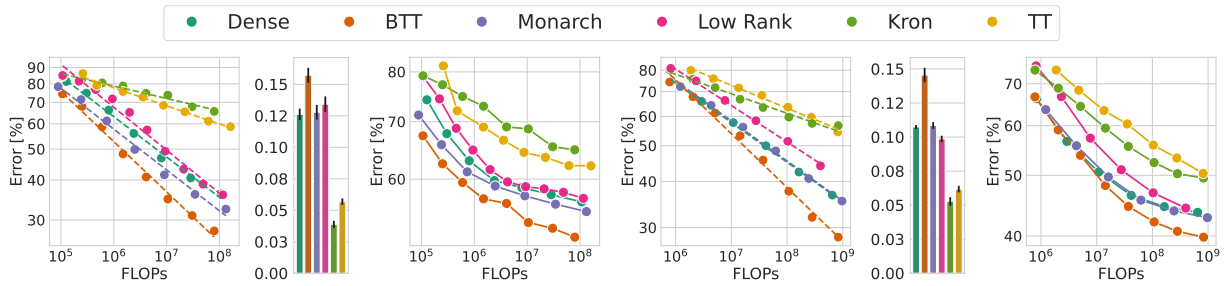
Having developed an effective procedure to automatically scale the initialization and learning rates for structured layers, we now aim to understand how various structures compare in performance.

When data is not a bottleneck, a neural network's test error or loss on a task follows a power law $E \propto P^{-\alpha_P}$ if trained to (near) convergence, where P is the number of parameters and α_P is a constant (Kaplan et al., 2020; Hoffmann et al., 2022; Henighan et al., 2020). For dense models, compute per forward pass $C \propto P$, so $E \propto C^{-\alpha_C}$ for some constant α_C. We explore how different structures change how E scales with C, as P does not consistently relate to training or inference cost when varying the structure (Table 1).

We train all models for a fixed number of iterations T, so the total training compute $C_{\text{tot}} \propto C$. Thus, the scaling laws in C can differ from compute-optimal scaling laws, which require carefully optimizing the allocation of $C_{\text{tot}} \propto CT$ between C and T (Kaplan et al., 2020; Hoffmann et al., 2022), which we leave to future work.

To compare multiple structures across compute scales, we conduct experiments primarily using MLPs and ViTs on CIFAR-10 and CIFAR-100. In Section 5, we present larger-scale experiments on ImageNet and language modeling. With limited training data in CIFAR-10 and CIFAR-100, we apply heavy augmentation to alleviate over-fitting. The augmented training set is sufficiently large, resulting in relatively clean power-law scaling of training error with C. We extract these power law parameters, reflecting the expressivity afforded by each structure as a function of C, and visualize the scaling of test error with C, which is not well-described by a power law due to train-test discrepancy.

Experimental setup. We use CIFAR-10 and CIFAR-100 datasets, applying random crop, random flip, MixUp ($\alpha_{\text{mixup}} = 0.8$) augmentations, and label smoothing of 0.3, following Bachmann et al. (2023). We use the same MLP architecture as in Bachmann et al. (2023), but apply a fixed



(a) MLP train error and exponent (b) MLP test error (c) ViT train error and exponent (d) ViT test error

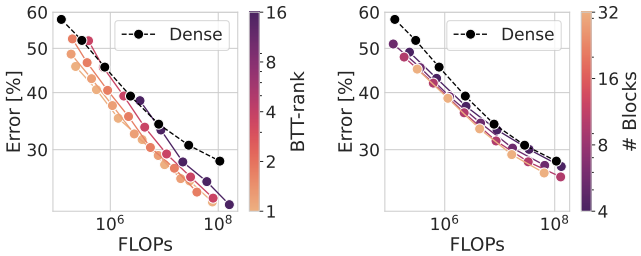
Figure 4. **Using structured matrices changes the scaling laws of MLPs and ViTs trained on CIFAR-100.** We find 1) BTT achieves the best scaling, and 2) structures with FLOPs equal to parameters scale better than those with parameter sharing (Kronecker and TT)

random permutation to the pixels before feeding them to the MLP so our results will more likely generalize to non-image data. We also use ViTs (Dosovitskiy et al., 2020) with 8×8 patches. We train MLPs for 500 epochs with batch size of 1024, and ViTs for 200 epochs with batch size of 256. To scale up the model, we increase its width while holding the depth constant. For structured models, we replace all except the classification layer with structured layers, though we keep the input layer dense for low rank to avoid an information bottleneck at the first layer. For Monarch, we set the number of blocks $b = 4$ following Dao et al. (2022) unless stated otherwise. We use BTT with two cores and various BTT-ranks. Further experiment details are in Appendix E.

Scaling exponents are structure-dependent. In Figure 1b and Figure 4, we find the training error E has an approximate power law relation to the compute C : $E \propto C^{-\alpha_C}$, for both MLPs and ViTs, where the exponent α_C varies significantly among structures. We show the best-fit exponent α_C and its standard error for each structure and plot the fitted power law trends. Monarch ($b = 4$) achieves equal or lower train and test error than dense for the same amount of compute, though it does not improve the scaling exponent of training error. BTT has the largest scaling exponent and consistently outperforms all other structures. We use BTT with two cores and rank 1, equivalent to a Monarch with $\sqrt{d}$ blocks, but BTTs with higher ranks also improve scaling as we will soon show.

Parameters equal FLOPs leads to better scaling laws. Figure 1b and Figure 4 reveal a qualitative difference between the scaling behavior of structures that perform parameter-sharing, i.e. Kronecker and TT, and those that do not, having parameters equal to FLOPs. Structures that do not share parameters are more flexible per unit of compute, and consistently achieve better scaling laws.

Recent works proposing to explain scaling laws from the data manifold dimension (Bahri et al., 2021; Sharma & Kaplan, 2022) can naturally explain worse scaling exponents due to parameter-sharing. This theory predicts the scaling

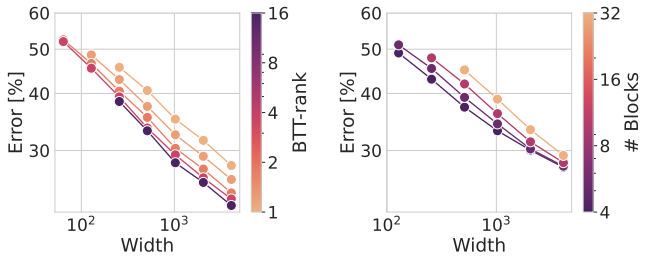


(a) BTT (b) Monarch

Figure 5. **Less compute per dimension is more compute-efficient on CIFAR-10.** (a) BTT with a lower rank achieves lower train error per FLOP. (b) Monarch with more blocks achieves lower train error per FLOP. A lighter color indicates less compute per dimension.

exponent α_P with respect to parameters is determined only by the intrinsic dimension of the data manifold, explaining why architectural details often only have minor impacts on the scaling laws (Kaplan et al., 2020). If changing the matrix structure leaves α_P invariant, then the scaling exponent α_C will depend on the structure in a simple way: if $C \propto P^{\beta}$, then $\alpha_C = \alpha_P/\beta$, that is, the more parameters sharing, the smaller the exponent α_C. For example, $\beta = 1$ for dense, low-rank, and BTT, but $\beta = 3/2$ for Kronecker and TT. However, this exact factor underestimates the observed differences in the exponents between Kronecker, TT, and dense, and does not explain why BTT has a larger exponent. A more accurate model is needed to explain the observed structure-dependence of the scaling exponents.

Optimizing compute spent per dimension. Both BTT and Monarch have hyperparameters (BTT-rank r and number of blocks b) that control how well they can approximate a dense matrix of the same dimension. We can scale up the compute C in a structured layer by increasing either its dimension d or its compute per dimension $\xi := C/d$ (compute cost for an MVM normalized by d), which is controlled by these hyperparameters. From Table 1, the compute per dimension is d for dense, $2r\sqrt{d}$ for BTT (with 2 cores), and $2d/b$ for Monarch. To maximize performance as a function



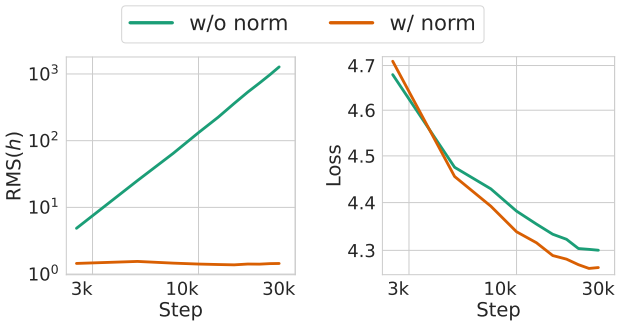
(a) BTT (b) Monarch

Figure 6. **More compute per dimension is more memory-efficient on CIFAR-10.** (a) BTT with a higher rank achieves lower train error per unit width. (b) Monarch with fewer blocks achieves lower train error per unit width. A smaller width means less memory is required to store the activations. A lighter color indicates less compute per dimension.

of C, we need to optimally allocate it between the dimension d of the layer and the compute spent per dimension ξ. In Figure 5a, we show that while higher rank BTTs scale better than dense matrices on CIFAR-10, lower rank BTTs are more compute-efficient. Similarly, in Figure 5b, Monarch matrices with more blocks and higher sparsity are more compute-efficient. These results illustrate that the optimal compute per dimension on this task is much smaller than d, and structured matrices beat dense matrices by making a favorable trade-off between dimension and compute per dimension. In Appendix F, we show that for BTT with $c \geq 3$ cores and different BTT-ranks, smaller ranks lead to better compute-efficiency, and using c greater than 2 does not significantly improve compute efficiency on CIFAR-10, despite compute per dimension scaling as $\mathcal{O}(d^{1/c})$.

The optimal way to scale ξ with d is likely non-trivial and task-dependent. The extremes are $\xi = d$ for a dense matrix and $\xi = 0$ for the identity. The latter is clearly suboptimal, and neither is the former in light of our findings.

Compute-memory trade-off. While lowering the compute per dimension can increase compute efficiency, it sacrifices memory efficiency if the memory cost is dominated by storing activations, such as when training with large batch sizes. In this case, the memory for storing activations scales at least as the layer width d. Since we can increase the expressivity of BTT and Monarch by increasing the rank or decreasing the number of blocks without increasing d, these hyperparameters enable us to trade off compute-efficiency with memory-efficiency, as demonstrated in Figure 6. While dense matrices are the least compute-efficient, they are the most efficient in terms of activation memory by packing the most parameters and compute into each dimension. The most compute-efficient yet memory-feasible structure will vary depending on the specific memory budget.



(a) Activation growth (b) Validation loss

Figure 7. **Weight normalization is necessary to stabilize GPT-2 training with BTT.** (a) RMS entry size of the final layer activations stabilizes around 1 with normalization but grows without bound otherwise. (b) Normalization improves validation loss.

5. Training Structured Transformers

We now apply structured layers to train larger transformer models for ImageNet classification and language modeling. We also introduce a technique required to prevent training divergence in these experiments.

5.1. Stabilizing Training with Weight Normalization

When training on ImageNet and OpenWebText with BTT layers, we found the activations grow without bound slowly over time as illustrated in Figure 7a for GPT-2, which does not happen in the dense model. We found we can eliminate this behavior without sacrificing expressivity through the following reparameterization:

$$\tilde{\mathbf{M}} = \gamma_{\mathbf{M}} \min\left(1, \frac{\sigma_{\mathbf{M}}}{\mathrm{RMS}(\mathbf{M})}\right)\mathbf{M},$$

where $\mathbf{M} \in \{\mathbf{L}, \mathbf{R}\}$. It normalizes the BTT cores $\mathbf{L}$ and $\mathbf{R}$ to have RMS entry sizes no larger than their initialization scales $\sigma_{\mathbf{L}}$ and $\sigma_{\mathbf{R}}$, and scaling them by learnable scalars $\gamma_{\mathbf{L}}$ and $\gamma_{\mathbf{R}}$ to allow the singular values to grow in size if needed, similar to what is proposed in Salimans & Kingma (2016). In Figure 7, we show a 12-layer GPT-2 model with $d = 128$ using BTT layers trained on OpenWebText with or without normalization. Weight normalization eliminates the unbounded growth of the activations before the last layer normalization, which will eventually lead to NaN. Weight normalization also improves validation loss, which is in contrast to alternatives such as lowering the learning rate and increasing weight decay which we found to only reduce the rate of growth at the cost of worse performance.

5.2. ViT on ImageNet

We train ViTs with patch size 32 on ImageNet for 300 epochs. We provide full experimental details in Appendix G. In Figure 8, we find both BTT with rank $r \in \{1, 2\}$ and Monarch with $b \in \{4, 16\}$ blocks outperform dense for the

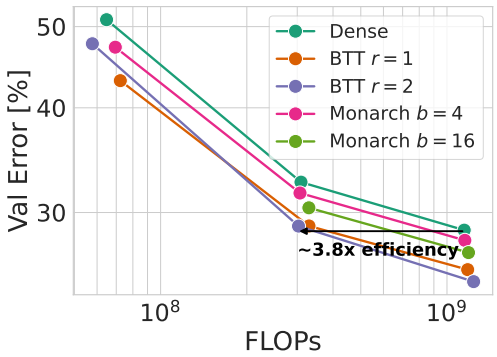
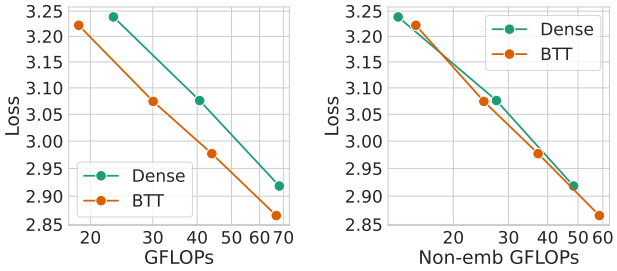


Figure 8. **ViTs trained on ImageNet with structured layers are more compute-efficient**. We use ViTs with patch size 32 trained for 300 epochs. BTT reaches the same performance of a dense ViT-S/32 with up to $3.8\times$ fewer FLOPs.



(a) All compute (b) Non-embedding compute

Figure 9. **GPT-2 with all BTT layers is more compute-efficient**. (a) When including language modeling head compute, BTT is more efficient than dense. (b) When excluding language modeling head compute, BTT and dense perform similarly.

same amount of compute for training ViTs on ImageNet. BTT reaches the same performance of a dense ViT-S/32 (the larger dense model shown) with up to $3.8\times$ fewer FLOPs. We find Monarch with 16 blocks is more compute-efficient than with 4 blocks, the original version used in Dao et al. (2022) and in the Monarch Mixer architecture (Fu et al., 2023), consistent with our finding on CIFAR-10 that less compute per dimension is more compute-efficient.

5.3. GPT-2 on OpenWebText

We train GPT-2 models on OpenWebText for $600,000$ steps with a batch size of $245,760$ tokens at a sequence length of 512. We provide full experimental details in Appendix G. We replace all linear layers, including the language modeling head, which accounts for a significant fraction of the compute, with BTT layers. In Figure 9a, we show the resulting GPT-2 model with BTT layers outperforms the original dense GPT-2 as a function of compute. However, in Figure 9b, we find they perform similarly when controlling for non-embedding compute, which excludes the compute spent in the language modeling head (Kaplan et al., 2020). While the improvement is significant, Figure 9b suggests that the improvement primarily comes from reducing the compute spent in the language modeling head and may therefore diminish at larger scales where the fraction of compute spent in the language modeling head becomes negligible.

6. Discussion

The exponential growth in the computational cost of training foundation models in recent years has made the development of more compute-efficient architectures and training procedures a critical area of research. While structured matrices have traditionally been used in machine learning to approximate dense matrices or encode constraints such as equivariance, our work shows their promise in serving as general-purpose linear layers, a universal compute bottleneck in current foundation models, while offering improved compute efficiency relative to dense matrices.

Our work uncovers several key insights in designing more compute-efficient linear layers with structured matrices:

- *Careful optimization is crucial:* structure-aware learning rates based on μP are essential to realize the performance benefits of structured matrices.
- *Better scaling laws than dense are possible:* structured matrices can sometimes exponentially outperform dense matrices as we increase compute.
- *Relaxing parameter sharing produces compute-efficient and general-purpose structures:* By learning more parameters with the same compute, Monarch and BTT can provide better performance as general linear layers than the parameter-sharing Kronecker product and Tensor-Train structures.
- *Compute per dimension is an impactful yet neglected hyperparameter:* dense matrices consume the most compute per dimension, but they can underperform structured matrices that trade less compute per dimension for more dimensions, resulting in wider models.

Extending our evaluation to larger-scale models and datasets, studying the compute-optimal scaling laws, and developing a theoretical understanding of when and why structured matrices can improve scaling laws based on data and model characteristics are exciting directions for future work.

Acknowledgements

We thank Sanae Lotfi, Alan Amin, and Bayan Bruss for helpful discussions, and Christopher Ferri for HPC assistance. This work is supported by NSF CAREER IIS-2145492, NSF CDS&E-MSS 2134216, NSF HDR-2118310, BigHat Biosciences, Capital One, and an Amazon Research Award.

Impact Statement

This work aims to improve the performance of MLPs and transformers per unit of compute. Making neural networks more efficient has the potential to reduce energy consumption of training and inference, and more efficient neural networks can also make deep learning accessible where compute resources are scarce. However, we caution that the matrix structures we use should be tested in new domains, at new architectural scales, and within new architectures, to ensure that our results extrapolate for a practitioner's specific individual needs.

References

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer Normalization. *Preprint arXiv 1607.06450*, 2016.

Bachmann, G., Anagnostidis, S., and Hofmann, T. Scaling mlps: A tale of inductive bias. *arXiv preprint arXiv:2306.13575*, 2023.

Bahri, Y., Dyer, E., Kaplan, J., Lee, J., and Sharma, U. Explaining neural scaling laws. *arXiv preprint arXiv:2102.06701*, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chekalina, V., Novikov, G., Gusak, J., Oseledets, I., and Panchenko, A. Efficient GPT Model Pre-training using Tensor Train Matrix Representation. *Preprint arXiv 2306.02697*, 2023.

Critch, A., Morton, J., et al. Algebraic geometry of matrix product states. *SIGMA. Symmetry, Integrability and Geometry: Methods and Applications*, 10:095, 2014.

Dao, T., Chen, B., Sohoni, N., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive Structured Matrices for Efficient and Accurate Training. *International Conference on Machine Learning (ICML)*, 2022.

Diederik P. Kingma, J. B. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations (ICLR)*, 2015.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *Preprint arXiv 2010.11929*, 2020.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T., Clark, J., Kaplan, J., McCandlish, S., and Olah, C. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Fannes, M., Nachtergaele, B., and Werner, R. F. Finitely correlated states on quantum spin chains. *Communications in mathematical physics*, 144:443–490, 1992.

Finzi, M., Stanton, S., Izmailov, P., and Wilson, A. G. Generalizing convolutional neural networks for equivariance to lie groups on arbitrary continuous data. In *International Conference on Machine Learning*, pp. 3165–3176. PMLR, 2020.

Frankle, J. and Carbin, M. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks: w. *International Conference on Learning Representations (ICLR)*, 2018.

Fu, D. Y., Arora, S., Grogan, J., Johnson, I., Eyuboglu, S., Thomas, A. W., Spector, B., Poli, M., Rudra, A., and Ré, C. Monarch Mixer: A Simple Sub-Quadratic GEMM-Based Architecture. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *The 4th International Conference on Learning Representations (ICLR)*, 2016.

Hayou, S., Ghosh, N., and Yu, B. Lora+: Efficient low rank adaptation of large models. *arXiv preprint arXiv:2402.12354*, 2024.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015a.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. *Preprint arXiv 1512.03385*, 2015b.

Hendrycks, D. and Gimpel, K. Gaussian Error Linear Units (GELUs). *Preprint arXiv 1606.08415*, 2016.

Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., et al. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.

Henry, A., Dachapally, P. R., Pawar, S., and Chen, Y. Query-key normalization for transformers. *arXiv preprint arXiv:2010.04245*, 2020.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-Rank Adaptation of Large Language Models. *Preprint arXiv 2106.09685*, 2021.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Krizhevsky, A., Sutskever, I., , and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM, Volume 60, Issue 6*, 2012.

LeCun, Y., Bottou, L., Bengio, Y., , and Haffner, P. Gradient-Based Learning Applied to Document Recognitio. *Proceedings of the IEEE, Volume: 86, Issue: 11*, 1998a.

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

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–50. Springer, 2002.

Lee, C. and Kim, H.-S. Differentiable learning of generalized structured matrices for efficient deep neural networks. *arXiv preprint arXiv:2310.18882*, 2023.

Lialin, V., Muckatira, S., Shivagunde, N., and Rumshisky, A. Relora: High-rank training through low-rank updates. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. Learning Efficient Convolutional Networks through Network Slimming. *International Conference on Computer Vision (ICCV)*, 2017.

Maron, H., Litany, O., Chechik, G., and Fetaya, E. On learning sets of symmetric elements. In *International conference on machine learning*, pp. 6734–6744. PMLR, 2020.

Michaud, E. J., Liu, Z., Girit, U., and Tegmark, M. The quantization model of neural scaling. *arXiv preprint arXiv:2303.13506*, 2023.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating Sparse Deep Neural Networks. *Preprint arXiv 2104.08378*, 2021.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning Convolutional Neural Networks for Resource Efficient Inference. *International Conference on Learning Representations (ICLR)*, 2016.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. Tensorizing Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

Oseledets, I. V. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing*, 2011.

Pan, Y., Su, Z., Liu, A., Jingquan, W., Li, N., and Xu, Z. A unified weight initialization paradigm for tensorial convolutional neural networks. In *International Conference on Machine Learning*, pp. 17238–17257. PMLR, 2022.

Perez, E., Strub, F., de Vries, H., Dumoulin, V., and Courville, A. FiLM: Visual Reasoning with a General Conditioning Layer. *Association for the Advancement of Artificial Intelligence (AAAI)*, 2017.

Potapczynski, A., Finzi, M., Pleiss, G., and Wilson, A. G. Cola: Exploiting compositional structure for automatic and efficient numerical linear algebra. *Advances in Neural Information Processing Systems*, 36, 2024.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language Models are Unsupervised Multitask Learners. *OpenAI*, 2019.

Saatçi, Y. *Scalable inference for structured Gaussian process models*. PhD thesis, Citeseer, 2012.

Salimans, T. and Kingma, D. P. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Sharma, U. and Kaplan, J. Scaling laws from the data manifold dimension. *Journal of Machine Learning Research*, 23(9):1–34, 2022.

Titsias, M. K. Variational Learning of Inducing Variables in Sparse Gaussian Processes. *International Conference on Artificial Intelligence and Statistics, pp. 567-574*, 2009.

Tolstikhin, I., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., and Dosovitskiy, A. MLP-Mixer: An all-MLP Architecture for Vision. *Preprint arXiv 2105.01601*, 2021.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-Attention with Linear Complexity. *Preprint arXiv 2006.04768*, 2020.

Wightman, R. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

Wilson, A. and Adams, R. Gaussian process kernels for pattern discovery and extrapolation. In *International conference on machine learning*, pp. 1067–1075. PMLR, 2013.

Wilson, A. and Nickisch, H. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International conference on machine learning*, pp. 1775–1784. PMLR, 2015.

Wortsman, M., Liu, P. J., Xiao, L., Everett, K., Alemi, A., Adlam, B., Co-Reyes, J. D., Gur, I., Kumar, A., Novak, R., et al. Small-scale proxies for large-scale transformer training instabilities. *arXiv preprint arXiv:2309.14322*, 2023.

Yang, G. and Hu, E. J. Feature Learning in Infinite-Width Neural Networks. *International Conference on Machine Learning (ICML)*, 2021.

Yang, G. and Littwin, E. Tensor Programs IVb: Adaptive Optimization in the Infinite-Width Limit. *International Conference on Learning Representations (ICLR)*, 2023.

Yang, G., Hu, E. J., Babuschkin, I., Sidor, S., Liu, X., Farhi, D., Ryder, N., Pachocki, J., Chen, W., and Gao, J. Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Yang, G., Simon, J. B., and Bernstein, J. A Spectral Condition for Feature Learning. *Preprint arXiv:2310.17813*, 2023a.

Yang, G., Yu, D., Zhu, C., and Hayou, S. Tensor programs vi: Feature learning in infinite-depth neural networks. *arXiv preprint arXiv:2310.02244*, 2023b.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

A. Related Work

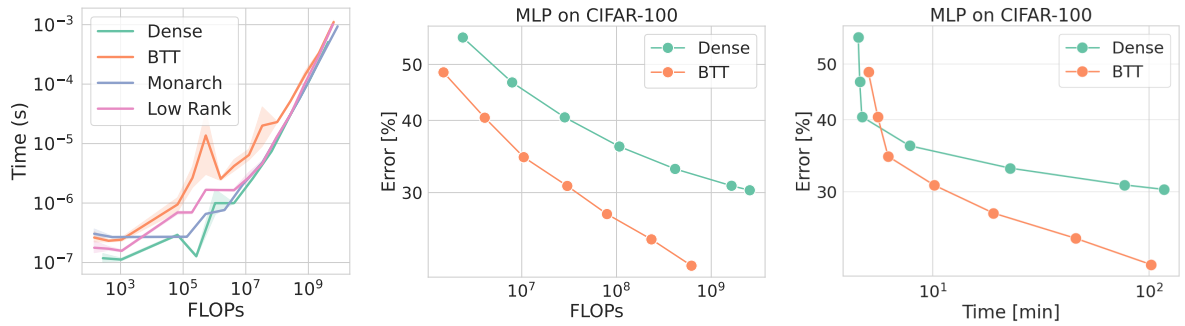
Compute-Efficient Alternatives to Dense Layers. Finding more compute-efficient alternatives to dense layers during training is an under-explored research topic. Convolutional networks and other equivariant models using structured matrices only offer an advantage in specific domains where the assumed symmetries exist (LeCun et al., 1998b; Finzi et al., 2020). Approaches such as pruning and quantization (Han et al., 2016; Molchanov et al., 2016; Liu et al., 2017; Frankle & Carbin, 2018; Mishra et al., 2021) mainly target reducing the inference cost after a model has been trained. Similarly, Lee & Kim (2023) introduce a differentiable approach to learn a sparse structure that contain sums of low-rank blocks, but the learned structure can only be made sparse after training. Efficient fine-tuning methods leveraging structured matrices, such as LoRA (Hu et al., 2021), only apply in the fine-tuning stage. Recent works have used low-rank structures to reduce the memory usage of training and accelerate the backward pass, but they still use dense matrices in the forward pass (Zhao et al., 2024; Lialin et al., 2023). While Tensor-Train decomposition can improve parameter efficiency of neural networks (Chekalina et al., 2023; Novikov et al., 2015), they have not been shown to improve their compute efficiency.

The recently proposed Monarch matrices (Dao et al., 2022) are a notable exception, which enable faster training of certain vision and language transformers by training with Monarch matrices for all or most of the training steps followed by only a small amount of dense training.

Initialization and Learning Rate for Structured Layers. The most popular initialization strategies such as Xavier (Glorot & Bengio, 2010), Kaiming (He et al., 2015a), and Lecun (LeCun et al., 2002) initializations set the initialization scales of the dense matrices so that the forward or backward pass is variance preserving at initialization. Pan et al. (2022) extended this analysis to tonsorial convolutional networks where the kernels are structured. In addition to considering only a subset of possible structures (dense and tensorial convolution), these strategies are not optimal because they only consider the initialization and not the training dynamics, as shown by μP (Yang et al., 2021). Specifically, μP uses an asymptotically smaller initialization variance compared to these methods when a layer's input dimension is asymptotically larger than its output dimension, such as the last layer.

To the best of our knowledge, there is no prior work that investigates how to scale the learning rate for general structured linear layers. Prior works using Tensor-Train Decomposition (Chekalina et al., 2023), low-rank matrices (Lialin et al., 2023), and Monarch matrices (Dao et al., 2022) to replace dense layers simply used global learning rates for all parameters and do not specify how they should be scaled as a function of width. The concurrent work LoRA+ (Hayou et al., 2024) studies the special case for low-rank matrices of the form $\mathbf{W} = \mathbf{U}\mathbf{V}, \mathbf{U} \in \mathbb{R}^{d\times r}, \mathbf{V} \in \mathbb{R}^{r\times d}, r \ll d$, and proposes that $\mathbf{U}$ should have a higher learning rate compared to $\mathbf{V}$, consistent with the more general analysis we present in this work that also applies to other structured matrices.

B. Runtime Comparisons



(a) MVM Runtime vs FLOPs (b) CIFAR-100 Train Error vs FLOPs (c) CIFAR-100 Train Error vs Runtime

Figure 10. **At large scales, runtime and FLOPs are equivalent for the structures we consider.** We omit Kronecker and TT in (a) because they are special cases of Monarch and BTT.

All structures in this work use the same dense matrix multiplication primitive on the GPU, so FLOPs are proportional to their runtimes for large matrix sizes. Only below a certain scale do runtimes vary noticeably between structures as a function of FLOPs. We verify this on an Nvidia A100 GPU in Figure 10a, showing the time for matrix-vector multiplication for different structures vs. FLOPs. For small matrices, runtimes vary between structures and don't reflect FLOPs due

Structure	Learning rate multiplier κ
Low-Rank $\mathbf{UV}$	$\kappa_{\mathbf{U}} = d/2r, \kappa_{\mathbf{V}} = 1/2$
Kronecker $\mathbf{L} \otimes \mathbf{R}$	$\kappa_{\mathbf{L}} = \sqrt{d}/2, \kappa_{\mathbf{R}} = \sqrt{d}/2$
Monarch $\mathbf{PLP}^\top\mathbf{R}$	$\kappa_{\mathbf{L}} = b/2, \kappa_{\mathbf{R}} = b/2$
TT$(\mathbf{L}, \mathbf{R})$	$\kappa_{\mathbf{L}} = \sqrt{d}/2r, \kappa_{\mathbf{R}} = \sqrt{d}/2$
BTT$(\mathbf{L}, \mathbf{R})$	$\kappa_{\mathbf{L}} = \sqrt{d}/2r, \kappa_{\mathbf{R}} = \sqrt{d}/2$

Table 2. **Learning rate multipliers for structured matrices.** We show the Adam learning rate multiplier κ we use for each parameter tensor of the structure when transferring the learning rate from a dense layer of the same width d. r refers to the rank in low rank, TT, and BTT, while b refers to the number of blocks in Monarch.

to inefficient tensor core utilization. For large matrices, runtimes converge to the same function in FLOPs. Optimizing structured matrix implementations can reduce their runtime overhead and will be essential to realizing the practical benefits of these structures.

Measuring FLOPs allows incorporating results from smaller experiments without letting the runtime inefficiencies at small scale obscure the scaling laws. Figure 10b and Figure 10 compare BTT with dense MLPs on CIFAR-100 in FLOPs and runtimes on an Nvidia A100. Below $\sim 10^7$ FLOPs, increasing FLOPs barely changes runtimes for dense and BTT, obscuring the scaling laws. BTT underperforms dense when controlling for runtime by incurring longer runtime per FLOP at this scale. However, as compute increases, scaling laws in FLOPs translate to scaling laws in runtimes, with BTT outperforming dense significantly.

C. General Expression for Tensor-Train and Block Tensor-Train

Here we describe the general expression for Tensor-Train and Block Tensor-Train, with an arbitrary number of cores and ranks. To make the expression more intuitive, we will use superscripts for output indices and subscripts subscripts for input indices. Rank indices appear once as a superscript when first introduced and once as a subscript when summed away.

Tensor-Train. Tensor-Train (TT) decomposition of a $d_{\text{out}} \times d_{\text{in}}$ matrix $\mathbf{W}$ is defined by a set of c cores $\mathbf{G}_t \in \mathbb{R}^{r_{t-1} \times m_t \times n_t \times r_t}$ for $t = 1, \ldots, c$, where $c \geq 2, d_{\text{out}} = \prod_t m_t, d_{\text{in}} = \prod_t n_t, r_0 = r_c = 1$ and $\{r_t\}_{t=1}^c$ being free integer hyperparameters. These cores specify the elements of an $n_1 \times \ldots \times n_t \times m_1 \times \ldots \times m_t$ tensor $\mathbf{T}$ via

$$T^{i_1,\ldots,i_c}_{j_1,\ldots,j_c} = \sum_{\alpha_1,\ldots,\alpha_{t+1}} \prod_{t=1}^{c} (G_t)^{\alpha_{t-1},i_t}_{j_t,\alpha_t}. \tag{4}$$

Identifying elements of $\mathbf{T}$ with elments of a $d_{\text{out}} \times d_{\text{in}}$ matrix $\mathbf{W}$, the efficient matrix-vector multiply against $\mathbf{W}$ does not involve materializing $\mathbf{W}$ but is simply given by a sequence of contractions against each core $\mathbf{G}_t$ from $t = c$ to $t = 1$:

$$(z_{t-1})^{\alpha_{t-1},j_1,\ldots,j_{t-1},i_t,\ldots,i_c} = \sum_{\alpha_t=1}^{r_t} \sum_{j_t=1}^{n_t} (G_t)^{\alpha_{t-1},i_t}_{j_t,\alpha_t} (z_t)^{\alpha_t,j_1,\ldots,j_t,i_{t+1},\ldots,i_c}, \tag{5}$$

where the initial $\mathbf{z}_c$ is obtained by reshaping the input $\mathbf{x}$ into an $n_c \times n_{c-1} \ldots \times n_1 \times 1$ tensor and the final $\mathbf{z}_0$ is flattened into an output vector. Suppose, for convenience, $d_{\text{in}} = d_{\text{out}} = d$, $n_t = m_t = d^{1/c}$ for all t, and $r_t = r$ for all $t \notin \{0, c\}$, then TT has $P = (2r + (c-2)r^2)d^{2/c}$ parameters, and an MVM costs $C = (2r + (c-2)r^2)d^{1+c^{-1}}$ FLOPs. Note we have $C = Pd^{1-c^{-1}}$, showing each parameter is used for $d^{1-c^{-1}} \geq \sqrt{d}$ times.

Block Tensor-Train. Block Tensor-Train (BTT) is defined simply by appending additional axes to each core in TT via the substitution

$$(G_t)^{\alpha_{t-1},i_t}_{j_t,\alpha_t} \to (G_t)^{\alpha_{t-1},i_t,i_{t+1},\ldots,i_c}_{j_1,\ldots,j_{t-1},j_t,\alpha_t}. \tag{6}$$

As before, multiplying the cores and summing out the rank axes, we have

$$T^{i_1,\ldots,i_c}_{j_1,\ldots,j_c} = \sum_{\alpha_1,\ldots,\alpha_{t+1}} \prod_{t=1}^{c} (G_t)^{\alpha_{t-1},i_t,i_{t+1},\ldots,i_c}_{j_1,\ldots,j_{t-1},j_t,\alpha_t}. \tag{7}$$

Efficient multiplication with the corresponding matrix is now given by

$$(z_{t-1})^{\alpha_{t-1},j_1,\ldots,j_{t-1},i_t,\ldots,i_c} = \sum_{\alpha_t=1}^{r_t} \sum_{j_t=1}^{n_t} (G_t)^{\alpha_{t-1},i_t,i_{t+1},\ldots,i_c}_{j_1,\ldots,j_{t-1},j_t,\alpha_t} (z_t)^{\alpha_t,j_1,\ldots,j_t,i_{t+1},\ldots,i_c}, \tag{8}$$

which costs the same FLOPs as for TT, while admitting more learnable parameters. Again we do not need to materialize $\mathbf{T}$. Suppose, for convenience, $d_{\text{in}} = d_{\text{out}} = d$, $n_t = m_t = d^{1/c}$ for all t, and $r_t = r$ for all $t \notin \{0, c\}$, then BTT has $P = (2r + (c-2)r^2)d^{1+c^{-1}}$ parameters, equal in number to the FLOPs for an MVM $C = (2r + (c-2)r^2)d^{1+c^{-1}}$. Thus, for the same amount of compute, BTT can learn a factor of $d^{1-c^{-1}} \geq \sqrt{d}$ more parameters than TT.

D. Expressivity of Block Tensor-Train

We start by providing an algorithm to approximate any existing dense matrix $\mathbf{A}$ with a BTT. The algorithm will then illustrate the expressivity of the BTT structure as a function of c and $\{r_t\}_{t=1}^{c}$. For simplicity, we will assume $\mathbf{A} \in \mathbb{R}^{d \times d}$, and the cores will be square, having size $d^{1/c}$ in each dimension, except for the rank dimension. Generalization to non-square $\mathbf{A}$ and non-square cores is straigtforward.

Projection onto Block Tensor-Train with $c = 2$. In the case where $c = 2$, we prove a closed-form expression for projecting an arbitrary dense matrix $\mathbf{A}$ to the closest rank-r (there is only one rank parameter so we omit the subscript) BTT $\mathbf{B}$ that minimizes the squared Frobenius norm $\|\mathbf{A} - \mathbf{B}\|_F^2$. Writing $\mathbf{A}$ and $\mathbf{B}$ as $\sqrt{d} \times \sqrt{d} \times \sqrt{d} \times \sqrt{d}$ tensors with $B^{ii'}_{jj'} = \sum_{\alpha=1}^{r} L^{ii'}_{j\alpha} R^{\alpha i'}_{jj'}$, we have

$$\begin{aligned}
&\|\mathbf{A} - \mathbf{B}\|_F^2 && (9)\\
&= \sum_{ii'jj'} \left(A^{ii'}_{jj'} - \sum_{\alpha=1}^{r} L^{ii'}_{j\alpha} R^{\alpha i'}_{jj'} \right)^2 && (10)\\
&= \sum_{i'j} \sum_{ij'} \left(A^{ii'}_{jj'} - \sum_{\alpha=1}^{r} L^{ii'}_{j\alpha} R^{\alpha i'}_{jj'} \right)^2 && (11)\\
&= \sum_{i'j} \left\| \mathbf{A}^{(i'j)} - \sum_{\alpha=1}^{r} \boldsymbol{\ell}^{(i'j)}_{\alpha} \mathbf{r}^{(i'j)\top}_{\alpha} \right\|_F^2, && (12)\\
& && (13)
\end{aligned}$$

where we have decomposed the minimization problem into multiple independent minimization problems: for each i', j, we wish to find the best rank-r approximation $\sum_{\alpha=1}^{r} \boldsymbol{\ell}^{(i'j)}_{\alpha} \mathbf{r}^{(i'j)\top}_{\alpha}$ to the matrix $\mathbf{A}^{(i'j)} \in \mathbb{R}^{\sqrt{d} \times \sqrt{d}}$. Thus, we obtain an optimal solution by finding these best rank-r approximation (e.g. via SVD) for each $\mathbf{A}^{(i'j)}$, and reassembling the vectors $\boldsymbol{\ell}^{(i'j)}_{\alpha}$ and $\mathbf{r}^{(i'j)}_{\alpha}$ into the tensors $\mathbf{L}$ and $\mathbf{R}$. This result is a straightforward generalization of the algorithm for projection onto Monarch matrices (Dao et al., 2022), which deals with the case where $r = 1$.

Generalization to $c > 2$. For convenience, let's relabel $\mathbf{L}$ found in the previous algorithm as $\tilde{\mathbf{L}}$, and the rank r as r_2. Having found $\tilde{\mathbf{L}}$ and $\mathbf{R}$, we can recursively apply the above algorithm on $\tilde{\mathbf{L}}$ to find its optimal 2-core rank-r_1 BTT approximation, with cores $\mathbf{L}$ and $\mathbf{M}$. Together, $\mathbf{L}, \mathbf{M}$, and $\mathbf{R}$ parameterize a 3-core BTT approximation with ranks r_1 and r_2. Similar to the recursive TT-SVD algorithm (Oseledets, 2011), the found solution will not necessarily be optimal for $c > 2$ due to its greediness.

It is sufficient to illustrate this algorithm in detail for $c = 3$. Reshaping $\mathbf{A}$ into a tensor $A^{i_1 i_2 i_3}_{j_1 j_2 j_3} \in \mathbb{R}^{d^{1/3} \times \ldots \times d^{1/3}}$, we wish to find $B^{i_1 i_2 i_3}_{j_1 j_2 j_3} = \sum_{\alpha=1}^{r} \sum_{\beta=1}^{r} L^{i_1 i_2 i_3}_{j_1 \beta} M^{\beta i_2 i_3}_{j_1 j_2 \alpha} R^{\alpha i_3}_{j_1 j_2 j_3}$ that approximates $\mathbf{A}$. We first group i_1, i_2 as a single index $(i_1 i_2)$ and

j_1, j_2 as a single index $(j_1 j_2)$, and then apply the previous algorithm for the 2-core case to find $\tilde{\mathbf{L}}, \mathbf{R}$ that minimizes

$$\sum_{(i_1 i_2) i_3 (j_1 j_2) j_3} \left(A^{(i_1 i_2) i_3}_{(j_1 j_2) j_3} - \sum_{\alpha=1}^{r_2} \tilde{L}^{(i_1 i_2) i_3}_{(j_1 j_2)\alpha} R^{\alpha i_3}_{(j_1 j_2) j_3} \right)^2, \tag{14}$$

forming the best following best rank-r_2 2-core approximation:

$$A^{(i_1 i_2) i_3}_{(j_1 j_2) j_3} \approx \sum_{\alpha=1}^{r_2} \tilde{L}^{(i_1 i_2) i_3}_{(j_1 j_2)\alpha} R^{\alpha i_3}_{(j_1 j_2) j_3}. \tag{15}$$

Setting $r_2 = \min(\#(i_1 i_2), \#j_3) = \sqrt{d}$ will lead to an exact decomposition, where $\#\chi$ denotes the length of the range of the index χ. Then we un-group the indicies to the obtain $\tilde{L}^{i_1 i_2 i_3}_{j_1 j_2 \alpha}, R^{\alpha i_3}_{j_1 j_2 j_3}$. Now grouping $i_2 i_3$ and $j_2 \alpha$ as single indices, we apply the previous algorithm again to find the best rank-r_1 2-core BTT approximation to $\tilde{\mathbf{L}}$ yielding the tensors $\mathbf{L}, \mathbf{M}$ that minimize

$$\sum_{i_1 (i_2 i_3) j_1 (j_2 \alpha)} \left(\tilde{L}^{i_1 (i_2 i_3)}_{j_1 (j_2 \alpha)} - \sum_{\beta=1}^{r_1} L^{i_1 (i_2 i_3)}_{j_1 \beta} M^{\beta (i_2 i_3)}_{j_1 (j_2 \alpha)} \right)^2. \tag{16}$$

Setting $r_1 = \min(\#i_1, \#(j_2 \alpha)) = \sqrt{d}$ will again lead to an exact decomposition, Now replacing $\tilde{L}^{i_1 i_2 i_3}_{j_1 i_2 \alpha}$ in Equation (15) by its approximation $\sum_{\beta=1}^{r_1} L^{i_1 i_2 i_3}_{j_1 \beta} M^{\beta i_2 i_3}_{j_1 j_2 \alpha}$, we have found the 3-core BTT approximation to $\mathbf{A}$ with ranks (r_1, r_2) :

$$A^{i_1 i_2 i_3}_{j_1 j_2 j_3} \approx B^{i_1 i_2 i_3}_{j_1 j_2 j_3} = \sum_{\beta=1}^{r_1} \sum_{\alpha=1}^{r_2} L^{i_1 i_2 i_3}_{j_1 \beta} M^{\beta i_2 i_3}_{j_1 j_2 \alpha} R^{\alpha i_3}_{j_1 j_2 j_3}. \tag{17}$$

Quantifying the expressivity of BTT. By applying the above recursive algorithm and always choosing a high enough rank so that the decomposition is exact at each step, we prove that a c-core BTT with sufficiently large ranks $\{r_t\}_{t=1}^{c}$ can represent any $d \times d$ dense matrix exactly. Moreover, the general expression for an upper-bound on r_t to ensure exact decomposition can be deduced as $r_t \leq \min(\#i_1 \times \ldots \times \#i_t, \#j_{t+1} \times r_{t+1}) \leq d^{\min(t, c-t)/c}$: i.e. $r_1 \leq d^{1/c}, r_2 \leq d^{2/c}, \ldots, r_{c/2} \leq \sqrt{d}, \ldots, r_{c-1} \leq d^{2/c}, r_c \leq d^{1/c}$. By contrast, TT has a worse bound of $r_1 \leq d^{2/c}, r_2 \leq d^{4/c}, \ldots, r_{c/2} \leq d, \ldots, r_{c-1} \leq d^{4/c}, r_c \leq d^{2/c}$ (Oseledets, 2011).

A practical takeaway is that we can monotonically improve the expressivity of BTT by increasing r_t until the bound is reached, and we should never use ranks larger than the bound since it creates unnecessary redundancy in the parameterization.

E. Scaling Laws Experiment Details

We provide code for reproducing our experiments here.

E.1. Model architectures

MLP. Following Bachmann et al. (2023), we use MLPs consisting of residual blocks of the form

$$\mathbf{h}_{\ell+1} = \mathbf{h}_\ell + \mathbf{W}^{(2)}_\ell g\Big(\mathbf{W}^{(1)}_\ell \mathrm{LN}\left(\mathbf{h}_\ell\right)\Big), \quad \mathbf{W}^{(1)}_\ell \in \mathbb{R}^{4d \times d}, \quad \mathbf{W}^{(2)}_\ell \in \mathbb{R}^{d \times 4d}, \tag{18}$$

where $g\left(\cdot\right)$ denotes the GELU activation (Hendrycks & Gimpel, 2016) and $\mathrm{LN}\left(\cdot\right)$ stands for layer normalization (Ba et al., 2016). In addition, there is an input embedding layer and a classification layer. We refer to d as the width of the model. We use models with 3 residual blocks and scale them up by increasing d.

ViT. We use standard ViTs (Dosovitskiy et al., 2020), but with $1/d-$scaled rather $1/\sqrt{d}-$scaled attention as prescribed by μP (Yang et al., 2021) and Query-Key Normalization (Henry et al., 2020; Wortsman et al., 2023) for improved stability. We refer to the embedding dimension, commonly denoted d_{model}, as the width d of the model. We use models with 3 transformer blocks and scale them up by increasing d.

E.2. Hyperparameters

Training hyperparameters. We use random crop, random flip, and MixUp ($\alpha = 0.8$) data augmentations, and label smoothing of 0.3. We train all MLP models for 500 epochs with batch size 1024, and all ViT models for 200 epochs with batch size 256. At the end of training, the models are close to but not exactly at convergence because fitting the training set is challenging due to strong augmentations and label smoothing. We do not use early stopping as it is not necessary.

We use structure-aware learning rates and initialization described in Section 3.2, with a cosine learning rate decay to 0. We set the constant in $\Theta(\cdot)$ as 1 for the initialization standard deviations, with the exception that the last linear layer inside every residual block of the MLP and ViT is zero-initialized, as mentioned in Section 3.2. For a structured layer, zero-initialization is only applied to its last dense component so its output is zero at initialization but all the parameters receive non-zero gradients after the first step. Following (Yang et al., 2021), we also zero-initialize the classification layer and the query projection $\mathbf{W}_Q$ in transformers. We found zero-initialization generally improves performance.

We use a base learning rate of $\eta_0 = 3e-3$ for a dense MLP at $d_0 = 64$, and $\eta_0 = 1e-3$ for a dense ViT at $d_0 = 64$. For MLPs, we scale the learning rate of the input layer by a factor of 0.1 since the input image dimension is much larger than d_0. This small multiplier prevents the first layer feature updates from having much larger scales than the other layers (Yang et al., 2023a), which we found improves performance.

Structure-specific hyperparameters. We provide hyperparameters such as ranks we use for each structure and any other design choices we make.

- Low-rank: we set the ranks of low-rank matrices to $\sqrt{\min(d_{\text{in}}, d_{\text{out}})}$ for MLP and $0.1 \times \min(d_{\text{in}}, d_{\text{out}})$ for ViT. The first choice leads to $\mathcal{O}\big(d^{3/2}\big)$ scaling of compute and parameters, same as Kronecker, 2-core BTT, and 2-core TT, but the second choice works significantly better for ViTs. We round the rank to its nearest integer when necessary. We initialize $\mathbf{V} \in \mathbb{R}^{r \times d}$ of the low-rank layer as $V_{ij} \sim \mathcal{N}(0, \sqrt{1/d_{\text{in}}})$, rather than $V_{ij} \sim \mathcal{N}(0, \sqrt{1/(rd_{\text{in}})})$. While the latter is required for having the desired spectral norm at initialization according to Section 3.2, when we choose a rank of $\sqrt{\min(d_{\text{in}}, d_{\text{out}})}$, it is not compatible with our zero-initialization scheme as it led to vanishing gradients for both $\mathbf{U}$ and $\mathbf{V}$ as the width gets large.
- Kronecker: for any dimension d that is not a perfect square, we factorize it so that the factors are as close as possible. For example, for a 20×30 matrix, we use the factorization $\mathbf{L} \otimes \mathbf{R}$ where $\mathbf{L} \in \mathbb{R}^{4\times 5}$ and $\mathbf{R} \in \mathbb{R}^{5\times 6}$.
- TT: we use two cores with TT-rank of 16 for MLPs and 8 for ViTs. We deal with non-perfect-square dimensions same as in Kronecker.
- Monarch: unless otherwise specified, we use $\mathbf{L}$ and $\mathbf{R}$ with 4 blocks, following the ViT and GPT-2 experiments in Dao et al. (2022).
- BTT: we use BTT with various ranks and deal with non-perfect-square dimensions same as in Kronecker.

F. Results for BTT with $c > 2$

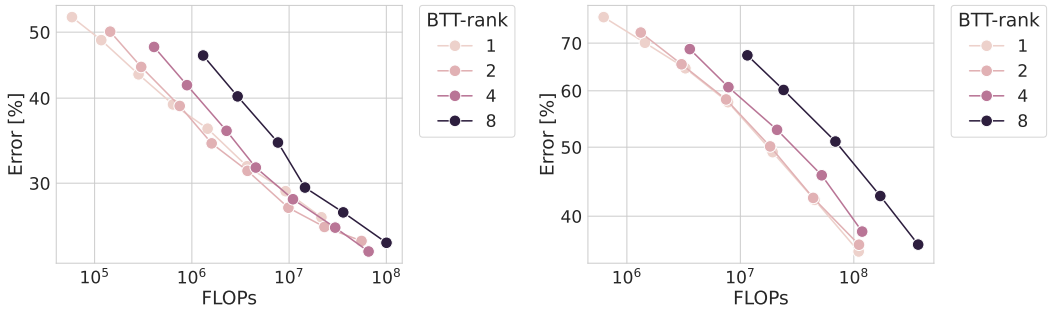
In Figure 5, we showed scaling compute per dimension ξ as $\xi = 2d^{1/2}$ using BTT with $c = 2$ and $r = 1$ leads to better scaling laws than other choices of r that increases ξ to $2r^{1/2}$. The gap between different choices of r closes as the models are scaled up in width, e.g. $d \gg r$. In Figure 11, we show a similar trend for $c = 3$, where higher values of r perform worse when controlling for FLOPs, though the gap tends to vanish as the width is scaled up. Each connected line shows the performance of BTT with a fixed r while d is increased.

In Figure 12, we show the performance of BTT with $r = 1$ and $c \in \{2, 3, 4\}$. Further reducing the scaling of ξ to $3d^{1/3}$ or $4d^{1/4}$ brings no or negligible improvement to performance when controlling for FLOPs.

In summary, choosing $c = 2$ and $r = 1$ leads to near-optimal performance for BTT on these tasks. In this case, BTT is equivalent to Monarch with $\sqrt{d}$ blocks.

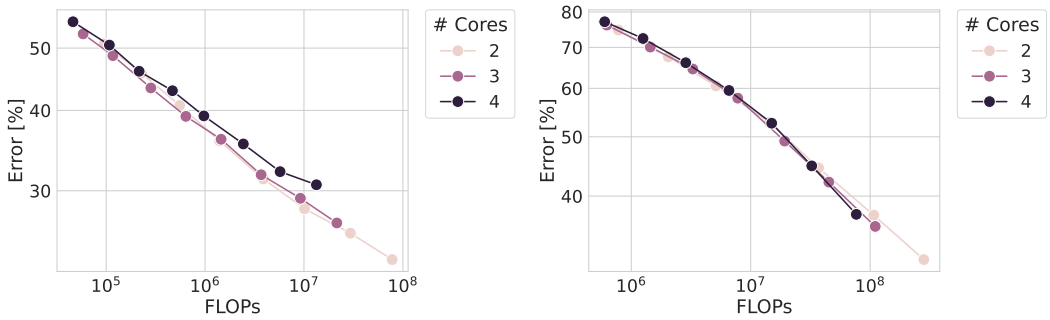
G. Transformer experiments

We provide code for reproducing our experiments here.



(a) MLP CIFAR-10 Train (b) ViT CIFAR-100 Train

Figure 11. **Lower BTT-ranks have better compute-efficiency for BTT with $c = 3$ cores.** Controlling for FLOPs, increasing the rank often degrades performance, though it reduces memory cost as the width is smaller.



(a) MLP CIFAR-10 Train (b) ViT CIFAR-100 Train

Figure 12. **BTT with $c = 2$ cores achieves near-optimal compute-efficiency.** Controlling for FLOPs, increasing c beyond 2 leads to no or negligible improvement in performance, while incurring higher memory costs as the models are wider.

G.1. ViT on ImageNet

We train with a global batch size of 3072 for 300 epochs with random crops, horizontal flip, random augmentations (`rand-m9-mstd0.5-inc1` from the `timm` library (Wightman, 2019)), and Mixup of 0.2. The model has 12 transformer blocks, with width d_{model} ranging from 80 to 384 for dense. We use BTT with rank 1 or 2 and Monarch with 4 or 16 blocks. All but the classification head is replaced with structured matrices. We use the AdamW optimizer and set the base learning rate to $2e-3$ for the smallest dense model, which is transferred to other models via μP and our structured-aware learning rate scaling. We apply a cosine learning rate decay to 0. The AdamW weight decay is set to 0.05 for all models and is scaled automatically with width by being multiplied by the learning rate (Yang et al., 2021). The architecture is identical to the one in Appendix E.1.

G.2. GPT-2 on OpenWebText

We train with a global batch size of 480 and a context length of 512 for 600,000 steps. We report the performance of the following models, all having 12 transformer blocks:

- Structure = Dense, $d_{\text{model}} = 384$, $n_{\text{head}} = 6$, $d_{\text{head}} = 64$
- Structure = Dense, $d_{\text{model}} = 512$, $n_{\text{head}} = 12$, $d_{\text{head}} = 64$
- Structure = Dense, $d_{\text{model}} = 768$, $n_{\text{head}} = 12$, $d_{\text{head}} = 64$ (GPT-2 Small (Radford et al., 2019))
- Structure = BTT ($r = 4$), $d_{\text{model}} = 1024$, $n_{\text{head}} = 6$, $d_{\text{head}} = 64$
- Structure = BTT ($r = 4$), $d_{\text{model}} = 1536$, $n_{\text{head}} = 6$, $d_{\text{head}} = 64$
- Structure = BTT ($r = 4$), $d_{\text{model}} = 2048$, $n_{\text{head}} = 6$, $d_{\text{head}} = 64$
- Structure = BTT ($r = 4$), $d_{\text{model}} = 2560$, $n_{\text{head}} = 12$, $d_{\text{head}} = 64$

We use BTT with rank 4 in every linear layer, including the language modeling head. We set n_{head} to be smaller than the usual $d_{\text{model}}/d_{\text{head}}$ for the BTT models since otherwise we would spend too much compute in the attention layers relative to the FFN layers. We use the Adam optimizer and set the base learning rate to $6e-4$ for the dense model at $d_{\text{model}} = 768$, which is transferred to other models via μP and our structured-aware learning rate scaling.

H. Structure-Aware Learning Rate for Other Optimizers

The structure-aware learning rate scaling described in Section 3 applies to Adam or AdamW. However, we can derive appropriate scaling rules for other optimizers such as SGD. In Section 3.3, we obtain our structure-aware learning rate scaling rule in three steps: 1) decompose the matrix-vector multiplication (MVM) of a structured matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ as a sequence of batched MVMs involving only dense matrices $\{\mathbf{G}_i\}_{i=1}^{k}$, 2) identify the input and output dimensions d_{in}^i and d_{out}^i of these dense matrices, 3) apply μP to each of these dense matrices to scale their learning rates based on d_{in}^i and d_{out}^i. Steps 1 and 2 are optimizer-agnostic. While step 3 is optimizer-dependent, it only requires knowing how to set μP learning rates for regular dense matrices, which has been analyzed in prior works for various optimizers, including SGD, Adam, and SignSGD (Yang & Littwin, 2023; Yang et al., 2023a). For example, instead of having the learning rate η_i of $\mathbf{G}_i$ be $\Theta(1/d_{\text{in}}^i)$, which is correct for Adam, SGD would require $\eta_i = \Theta(d_{\text{out}}^i/d_{\text{in}}^i)$ (Yang et al., 2023a). Therefore, the structure-aware learning rate multiplier relative to a dense $\mathbf{W}$ should now be $\kappa_i = \Theta\left(\frac{d_{\text{out}}^i/d_{\text{in}}^i}{d_{\text{out}}/d_{\text{in}}}\right)$ instead of $\Theta(d_{\text{in}}/d_{\text{in}}^i)$, which is correct for Adam.

I. Limitations and Future Work

We provide a summary of the limitations of this work, and exciting directions for future work:

- Due to affordability constraints, we conducted our evaluation primarily with relatively small-scale models and datasets. Extending our evaluation to much larger-scale models and datasets is an important future direction.
- The scaling laws we study differ from the compute-optimal scaling laws more relevant for large-scale training, which require optimally trading off between training larger models and training for more iterations. We only varied model size while keeping training iterations constant. Similarly, we did not optimize between scaling width v.s. depth, which allowed us to conveniently transfer learning rate through μP[2].
- Our comparisons are based on FLOPs rather than runtimes. While the structures we consider have asymptotically the same MVM runtimes as dense matrices per FLOP (Appendix B), they introduce non-trivial runtime overhead for small matrix sizes, e.g. $\mathcal{O}(10^3)$. Developing highly optimized implementations will be important to realize the benefits of structured matrices in practice.
- Despite our efforts to avoid over-fitting to image data (shuffling pixels for the MLP experiment), our findings that structured matrices can significantly outperform dense matrices may still be highly dataset-dependent, as BTT offers a less significant improvement in language modeling compared to in image classification.
- Our findings are empirical. Theoretically understanding when and why structured matrices can have better scaling laws than dense matrices, depending on model and data characteristics, will enable a prescriptive selection of structure rather than via trial and error alone.

[2] See Yang et al. (2023b) for a depth extension of μP and why it doesn't work for transformers in principle.