Learning Fast Algorithms for Linear Transforms Using Butterfly Factorizations

Tri Dao [1] **Albert Gu** [1] **Matthew Eichhorn** [2] **Atri Rudra** [2] **Christopher Ré** [1]

Abstract

Fast linear transforms are ubiquitous in machine learning, including the discrete Fourier transform, discrete cosine transform, and other structured transformations such as convolutions. All of these transforms can be represented by dense matrix-vector multiplication, yet each has a specialized and highly efficient (subquadratic) algorithm. We ask to what extent hand-crafting these algorithms and implementations is necessary, what structural priors they encode, and how much knowledge is required to automatically learn a fast algorithm for a provided structured transform. Motivated by a characterization of matrices with fast matrix-vector multiplication as factoring into products of sparse matrices, we introduce a parameterization of divide-and-conquer methods that is capable of representing a large class of transforms. This generic formulation can automatically learn an efficient algorithm for many important transforms; for example, it recovers the $O(N \log N)$ Cooley-Tukey FFT algorithm to machine precision, for dimensions N up to 1024. Furthermore, our method can be incorporated as a lightweight replacement of generic matrices in machine learning pipelines to learn efficient and compressible transformations. On a standard task of compressing a single hidden-layer network, our method exceeds the classification accuracy of unconstrained matrices on CIFAR-10 by 3.9 points—the first time a structured approach has done so—with 4X faster inference speed and 40X fewer parameters.

1. Introduction

Structured linear transformations, such as the discrete Fourier transform (DFT), discrete cosine transform (DCT), and Hadamard transform, are a workhorse of machine learning, with applications ranging from data preprocessing, feature generation, and kernel approximation, to image and language modeling (convolutions). To date, these transformations rely on carefully designed algorithms, such as the famous fast Fourier transform (FFT) algorithm, and on specialized implementations (e.g., FFTW and cuFFT). Moreover, each specific transform requires hand-crafted implementations for every platform (e.g., Tensorflow and PyTorch lack the fast Hadamard transform), and it can be difficult to know when they are useful. Ideally, these barriers would be addressed by automatically learning the most effective transform for a given task and dataset, along with an efficient implementation of it. Such a method should be capable of recovering a range of fast transforms with high accuracy and realistic sizes given limited prior knowledge. It is also preferably composed of differentiable primitives and basic operations common to linear algebra/machine learning libraries, that allow it to run on any platform and be integrated into modern ML frameworks such as PyTorch/Tensorflow. More fundamentally, this problem ties into the foundational question of understanding the minimal prior knowledge needed to learn high-speed systems, in the spirit of modern trends toward relaxing manually imposed structure (i.e., AutoML). Recent progress in this vein of learning computational primitives includes addition/multiplication gates (Trask et al., 2018) and the Strassen 2×2 matrix multiplication algorithm (Tschannen et al., 2018).

We propose a method that addresses this problem for a class of important transforms that includes the aforementioned examples. A key challenge lies in defining or parameterizing the space of transforms and corresponding fast algorithms, which requires using a minimal amount of prior knowledge that captures important and interesting transforms while remaining learnable and efficient. Egner & Püschel (2001; 2004) previously posed this question and a novel combinatorial approach, but their solution only addresses a limited set of transforms (primarily DFT) and only on limited problem sizes. In particular, these approaches search through an exponentially large discrete space using a symbolic form of the matrix (Egner & Püschel, 2001; 2004) and recover the solution only up to dimensions 8×8. We instead draw two key lessons from the work of De Sa et al. (2018), who characterize matrices with efficient matrix-vector multiplication algorithms as being factorizable into products of sparse

[1]Department of Computer Science, Stanford University, USA [2]Department of Computer Science and Engineering, University at Buffalo, SUNY, USA. Correspondence to: Tri Dao <trid@cs.stanford.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

matrices.[1] Thus, the task of learning algorithms can be reduced to finding appropriate sparse matrix product representations of the transforms. They further show that divide-and-conquer schemes lead to fast multiplication algorithms for a surprisingly general set of structured matrices. Motivated by the broad applicability of this recursive structure, we propose a particular factorization using sequences of special block diagonal matrices, called *butterfly matrices*. Specific instances of butterfly structure have been used before—for example as a random orthogonal preconditioner (Parker, 1995) or in matrix approximation (Li et al., 2015)—but we use a relaxed representation that captures a larger class of structures and can learn from data. These form a class of structured matrices with $O(N)$ parameters and automatic fast multiplication in $O(N \log N)$ operations.

We empirically validate our method in two ways. First, we consider a specification of a transform (e.g., N input-output pairs) and attempt to factorize it. We successfully recover a fast algorithm up to machine precision for several important transforms such as the DFT, Hadamard, DCT, and convolution for realistic sizes (dimensions up to $N = 1024$), while standard sparse and low-rank baselines cannot (Section 4.1). Beyond recovering famous transforms, we additionally incorporate this method in end-to-end ML pipelines to learn fast and compressible latent transformations (Section 4.2). On the benchmark single hidden layer network, this parameterization exceeds the classification accuracy of a baseline fully connected layer on several datasets—such as by 3.9 points on CIFAR-10 while using 40X fewer parameters—which is to our knowledge the first time a structured model has outperformed the unconstrained model for this task on a realistic dataset (Thomas et al., 2018). We also find that the addition of a lightweight butterfly layer improves the accuracy of a modern ResNet architecture by 0.43 points.

Finally, our method is simple with an easily implementable fast algorithm. We compare the training and inference speed of our implementation to specialized implementations of discrete transforms (Section 4.3). Our generic representation comes within 3-5X of implementations for specific transforms such as the DFT and DCT, while still being capable of learning a rich class of more general transforms.

2. Related Work

Fast transforms are crucial and ubiquitous in the machine learning pipelines, from data preprocessing, feature generation, and dimensionality reduction to compressing models. For example, the DFT and DCT form the basis of the mel-frequency cepstral coefficients (MFCCs), a standard feature representation for speech recognition (Jurafsky & Martin, 2014). State-of-the-art kernel approximation methods leverage circulant matrices (i.e., convolution) (Yu et al., 2015) and the DFT and Hadamard transform (Le et al., 2013; Yu et al., 2016) for fast projection. Structured matrices, which are matrix representations of fast transforms, play a crucial role in designing fast neural network layers with few parameters (Sindhwani et al., 2015; Ding et al., 2017).

Given their importance, there have been significant efforts in finding more and more general classes of fast transforms. Traditional classes of structured matrices such as the Toeplitz, Hankel, Vandermonde, and Cauchy matrices are ubiquitous in engineering and signal processing (Pan, 2001), and more recently have found use in deep learning. These were generalized under the seminal notion of low displacement rank (LDR) introduced by Kailath et al. (1979), These, along with more general LDR as well as other families of transforms related to the DFT and DCT, were further significantly generalized under a single class by De Sa et al. (2018). Notably, almost all of the structured matrix classes mentioned here exhibit a form of recursive structure.

Since the product of sparse matrices immediately has a fast multiplication algorithm, the problem of sparse matrix factorization has been tackled in many settings. Sparse PCA (Zou et al., 2006) and dictionary learning (Mairal et al., 2009) factor a matrix into two components, one of which is sparse. Sparse matrix factorization with more than two factors has also been considered, for example in the setting where the true matrix is the product of random sparse matrices (Neyshabur & Panigrahy, 2013), or in the context of learning multi-layer sparse approximations (Le Magoarou & Gribonval, 2015; 2016). Our approach differs from these in that we focus on the recursive structure of the transforms, leading to sparse *and* structured transforms, and avoiding the discreteness problem inherent to learning sparsity.

Since most distinct transforms typically require significant work both to design fast algorithms and to efficiently implement them on different platforms, there have been attempts to automatically learn these fast algorithms. The field of algebraic signal processing (Puschel & Moura, 2008) uses methods from representation theory of groups and algebras to automatically generate fast algorithms from the symbolic form of the transform matrix. However, these methods require search over a combinatorially-large discrete space, limiting their approaches to small matrices of size up to 8×8 (Egner & Püschel, 2004; Voronenko & Puschel, 2009). Attempts to learn general algorithms such as matching (Mena et al., 2018), sorting (Grover et al., 2019), and traveling salesman (Bello et al., 2016) using differentiable architectures face a similar challenge of having to effectively explore a large discrete space. Thus, they only work for problems of size at most 100. By contrast, our approach simplifies the discreteness of the problem into learning a simpler set of permutations, allowing us to recover fast

[1] This characterization was equivalently known in the language of arithmetic circuits (Bürgisser et al., 2013).

algorithms for realistic dimensions.

Independently, there has been growing interest in compressed deep learning models, motivated by the goal of adapting them to resource-constrained environments. A common approach for learning compressed models involves replacing the unconstrained weight matrices with a class of structured matrices and learning directly on the parametrization of that class. The most effective methods use matrix classes that are explicitly related to Fourier transforms (Sindhwani et al., 2015), or employ highly specialized and complicated recursive algorithms (Thomas et al., 2018). As our method also implicitly defines a highly compressible subclass of matrices with linear parameter count and efficient multiplication, it can be used as a drop-in replacement for matrices in such end-to-end ML models.

3. Recovering Fast Transforms

We now set up and describe our approach. We first reiterate the connection between fast algorithms and sparse matrix factorization, and briefly outline a quintessential divide-and-conquer algorithm (the FFT) as motivation.

We then elaborate the details of our method for learning particular recursive algorithms, including a core permutation-learning step that enables it to capture a wider range of structures. We also discuss the expressive power of these matrices, including which transforms they capture perfectly, and define a hierarchy of matrix classes built on butterflies that can theoretically capture richer recursive structures.

3.1. Preliminaries

Sparse factorizations One method of constructing matrices with obvious fast matrix-vector multiplication is as a product of sparse matrices, so that multiplication by an arbitrary vector will have cost proportional to the total number of nonzeros of the matrices in the product.

Surprisingly, the converse is also true. The notion of *sparse product width* (SPW) (De Sa et al., 2018), which roughly corresponds to the total sparsity of a factorization of a matrix, turns out to be equivalent to the length of the shortest linear straight-line program describing a matrix (up to a constant). Hence, it is an optimal descriptor of the algorithmic complexity of matrix-vector multiplication on these types of models (Bürgisser et al., 2013).

Given the general correspondence between sparse factorization and fast algorithms, we consider specific types of discrete transforms and their recursive factorizations. This is a prototype for our parameterization of fast recursive algorithms in Section 3.2.

Case study: DFT The Discrete Fourier Transform (DFT) transforms a complex input vector $x = [x_0, \dots, x_{N-1}]$ into a complex output vector $X = [X_0, \dots, X_{N-1}]$ by expressing the input in the basis of the complex exponentials:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad k = 0, \dots, N-1, N = 2^m.$$

Let $\omega_N := e^{2\pi i/N}$ denote a primitive N-th root of unity. The DFT can be expressed as matrix multiplication by the *DFT matrix* $F_N \in \mathbb{C}^{N\times N}$, where $(F_N)_{kn} = \omega_N^{-kn}$. The DFT of size N can be reduced to two DFTs of size $N/2$ on the even indices and the odd indices:

$$F_N x = \begin{bmatrix} F_{N/2} x_{\text{even}} + \Omega_{N/2} F_{N/2} x_{\text{odd}} \\ F_{N/2} x_{\text{even}} - \Omega_{N/2} F_{N/2} x_{\text{odd}} \end{bmatrix},$$

where $x_{\text{even}} = [x_0, x_2, \dots, x_{N-2}]$, $x_{\text{odd}} = [x_1, x_3, \dots, x_{N-1}]$, and $\Omega_{N/2}$ is the diagonal matrix with entries $1, \omega_N^{-1}, \dots, \omega_N^{-(N/2-1)}$. This recursive structure yields the efficient recursive Cooley-Tukey Fast Fourier Transform (FFT) algorithm. This computation can be written as a matrix factorization

$$F_N = \begin{bmatrix} I_{N/2} & \Omega_{N/2} \\ I_{N/2} & -\Omega_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} \begin{bmatrix} \text{Sort the even} \\ \text{and odd indices} \end{bmatrix},$$

where $I_{N/2}$ is the identity matrix, and the last factor is the permutation matrix P_N that separates the even and odd indices (e.g., mapping $[0, 1, 2, 3]$ to $[0, 2, 1, 3]$) (see Figure 2). Unrolling the recursion, we obtain:

$$\begin{aligned} F_N &= B_N \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} P_N \\ &= B_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/4} & 0 & 0 & 0 \\ 0 & F_{N/4} & 0 & 0 \\ 0 & 0 & F_{N/4} & 0 \\ 0 & 0 & 0 & F_{N/4} \end{bmatrix} \\ &\quad \begin{bmatrix} P_{N/2} & 0 \\ 0 & P_{N/2} \end{bmatrix} P_N = \cdots \qquad (1) \\ &= \left(B_N \dots \begin{bmatrix} B_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & B_2 \end{bmatrix} \right) \left(\begin{bmatrix} P_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & P_2 \end{bmatrix} \dots P_N \right). \end{aligned}$$

The product of all the $B_{N/2^k}$ matrices on the left is called a *butterfly matrix*, and each factor $B_{N/2^k}$ is a 2×2 block matrix of diagonal matrices called a *butterfly factor*. Figure 1 illustrates the sparsity pattern of the structured butterfly factors. One can also combine the product of permutation matrices on the right to obtain a single permutation called the *bit-reversal permutation*, which sorts the indices by the reverse of their binary representation (e.g. $[0, \dots, 7] \to [0, 4, 2, 6, 1, 5, 3, 7]$).

Other transforms have similar recursive structure but differ in the entries of $B_{N/2^k}$, and in the permutation. For example, the DCT involves separating the even and the odd indices, and then reversing the second half (e.g., $[0, 1, 2, 3] \to [0, 2, 1, 3] \to [0, 2, 3, 1]$).

Appendix A provides some examples of how important transforms, such as the DFT, DCT, Hadamard, and convolutions, can factor as similar products of sparse matrices.

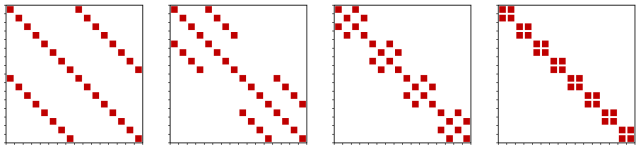


Figure 1. Butterfly matrix for $N = 16$. From left to right: single copy of B_{16}, blocks of B_8, blocks of B_4, blocks of B_2.

3.2. Recovering Fast Transform Algorithms

Many previous works attempt to compress generic matrices by sparsifying them. We note that allowing for products of matrices with a total sparsity budget is strictly more expressive than a single matrix with that sparsity, while retaining the same compression and computation complexity. Therefore one can hope to recover all fast algorithms by learning over the set of matrix products with a total sparsity budget. However, this is infeasible to learn due to the discreteness of the sparsity constraint (Section 1,2). We instead use a class of matrices built as products of specific factors that captures the recursive nature of many fast algorithms.

A butterfly parametrization Let $x = [x_0, \dots, x_{N-1}]$ be an input vector.[2] Let $\mathcal{T}_N$ be a linear transform of size N with matrix representation $T_N \in \mathbb{F}^{N\times N}$, where $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$. A general recursive structure is to separate the input vector into two halves by some permutation, apply the transform on each half, and combine the result in a linear manner by scaling by an diagonal matrix and adding the results. Written as a matrix factorization:

$$T_N = \begin{bmatrix} D_1 & D_2 \\ D_3 & D_4 \end{bmatrix} \begin{bmatrix} T_{N/2} & 0_{N/2\times N/2} \\ 0_{N/2\times N/2} & T_{N/2} \end{bmatrix} P_N,$$

where P_N is some permutation matrix and $D_1, \dots, D_4 \in \mathbb{F}^{N/2}$ are diagonal matrices. Inspired by the factors of the FFT, we call the matrix $\begin{bmatrix} D_1 & D_2 \\ D_3 & D_4 \end{bmatrix}$ a butterfly factor, denoted by B_N. Unrolling the recursion as in equation (1) gives the factorization $T_N = B^{(N)}P^{(N)}$, where $B^{(N)}$ is a butterfly matrix and $P^{(N)}$ is a permutation that can be written as the product of $\log_2(N)$ simpler block permutations. We also consider composing this module, hence learn either

$$T_N = B^{(N)}P^{(N)} \qquad T_N = B_2^{(N)}P_2^{(N)}B_1^{(N)}P_1^{(N)}, \quad (2)$$

which we term the BP and the BPBP parametrization respectively. One dimensional convolutions (i.e. circulant matrices) are notably captured by BPBP, since they can be computed via an FFT, a component-wise product, then an inverse FFT (see Appendix A).

[2]For simplicity, we assume that N is a power of 2. Otherwise, the input can be padded with zeros.

Learning a recursive permutation The butterfly blocks in the BP or BPBP parametrization have a fixed sparsity pattern and their parameters can be directly optimized. However, the transforms we are interested in capturing frequently require different permutations as part of the "divide" step, which form a set of discrete objects that we must consider. We will restrict to learning over permutations that have a simple structure often encountered in these algorithms: we assume that the distribution factors into $\log_2 N$ steps following the $\log_2 N$ recursive layers. At each step in the recursion, the permutation $P_{N/2^k}$ is allowed to either keep the first half and second half intact or separate the even and the odd indices (e.g., $[0,1,2,3] \to [0,2,1,3]$). Then, it can choose to reverse the first half (e.g., $[0,1] \to [1,0]$) and can choose to reverse the second half (e.g., $[2,3] \to [3,2]$). Thus at each step, there are 3 binary choices and hence 8 possible permutations. These are illustrated in Figure 2, where P_N^a denotes the permutation matrix on N elements that separates the even and odd elements, P_N^b denotes the permutation matrix that reverses the first half, and P_N^c denotes the permutation matrix that reverses the second half.

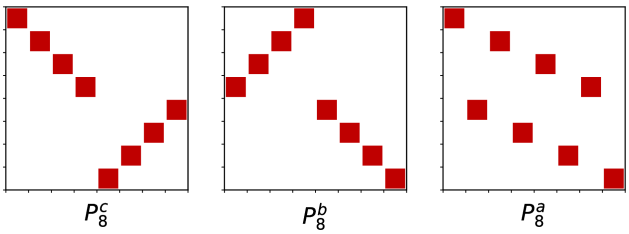


Figure 2. Three binary choices for constructing the permutation used at every step of the recursive process. One of 8 possible permutations can be constructed by multiplying a subset of these matrices in the presented order.

Instead of searching over $8^{\log_2 N}$ discrete permutations, we parameterize the permutation $P^{(N)}$ as a categorical distribution of these $8^{\log_2 N}$ permutations. The permutation $P_{N/2^k}$ at step k is thus chosen as a convex combination of the 8 possible choices:

$$P_{N/2^k} = p_{cba}P^c_{N/2^k}P^b_{N/2^k}P^a_{N/2^k} + p_{cb}P^c_{N/2^k}P^b_{N/2^k} + \dots.$$

This can be learned by representing this probability distribution $\{p_{cba}, p_{cb}, \dots\}$ for example via logits and the softmax.

We make the further simplification that the probabilities p_{cba} factor into the three components; conceptually, that the choices of choosing $P^c_{N/2^k}, P^b_{N/2^k}, P^a_{N/2^k}$ to be part of the product are independent of each other. This results in the representation $P_{N/2^k} = \prod_{s=c,b,a}(p_s P^s_{N/2^k} + (1-p_s)I)$.

Thus we learn $P_{N/2^k}$ by optimizing over 3 logits ℓ_a, ℓ_b, ℓ_c and setting $p_s = \sigma(\ell_s)$, where σ is the sigmoid function.

To encourage the distribution over permutations to be peaked, one can add entropy regularization (Grandvalet & Bengio, 2005) or semantic loss (Xu et al., 2018). However, we found that these tricks are not necessary. For example, the learned transforms in Section 4.1 typically put weight at least 0.99 on a permutation.

Initialization As the BP or BPBP construction is a product of many matrices, proper initialization is crucial to avoid exponential blowup in the size of the entries or condition numbers (i.e., the exploding/vanishing gradient problem (Pascanu et al., 2013)). We aim to initialize each butterfly factor to be close to unitary or orthogonal, so that the magnitude of the inputs and outputs to the transform are preserved. This is easy since each of the factors $B_N, \ldots, B_2$ has exactly two nonzeros in each row and column; for example in the real case, initializing each entry of B_k as $\mathcal{N}(0, 1/2)$ guarantees $\mathbb{E}B_k^* B_k = I_N$.

Comparison to related methods Some previous works have examined similar butterfly matrices in numerical algebra or machine learning (Parker, 1995; Jing et al., 2017; Munkhoeva et al., 2018), mainly motivated by trying to parametrize cheap orthogonal matrices. Our parametrization, motivated by the goal of learning recursive transforms, differs in several ways from all previous works: 1. We explicitly model and learn a permutation matrix P. 2. Our relaxation does not enforce the matrix to be orthogonal. 3. Our butterfly factors are ordered so that closer elements interact first (Figure 1), whereas some works (e.g. (Munkhoeva et al., 2018)) reverse the order. 4. Every work has a different weight-tying scheme; ours ties the blocks in each butterfly factor, leading to fewer parameters and a tighter recursive interpretation than for example (Jing et al., 2017).

3.3. Expressivity and the butterfly hierarchy

The butterfly matrix B has a total of $4N$ learnable parameters (the butterfly factors B_N, $B_N/2$, ..., B_2 have $2N$, N, ..., 4 entries respectively). The overall permutation P has $3\log_2 N$ learnable parameters; we can also tie the logits of the $\log_2 N$ probabilistic permutations—reflecting the fact that for some algorithms the reduction from size N to $N/2$ is self-similar to the reduction from size $N/2^k$ to $N/2^{k+1}$—reducing this to just 3 parameters.

We can define a natural hierarchy of matrix classes built on the BP primitive. This hierarchy covers a spectrum ranging from extremely structured matrices with a linear number of parameters, to the entire space of square matrices.

Definition 1. For any dimension N, let $(\text{BP})_r^k$ $(k, r \in \mathbb{N})$ denote the classes of matrices that can be expressed as

$$S\left(\prod_{i=1}^{k} B_i P_i\right) S^T,$$

where each $B_i P_i \in \mathbb{F}^{rN \times rN}$ is a BP module as in equation (2), and $S \in \mathbb{F}^{N \times rN} = \begin{bmatrix} I_N & 0 & \ldots & 0 \end{bmatrix}$ (that is, S and S^T select the upper left $N \times N$ entries of the BP product matrix). The subscript r is understood to be 1 if omitted.

Note that the BP and BPBP classes are equivalent to $(\text{BP})^1$ and $(\text{BP})^2$ respectively. We remark that B and P are both capable of being the identity, and thus $(\text{BP})^k \subseteq (\text{BP})^{k+1}$.

The BP hierarchy is expressive enough to theoretically represent many important transforms with low depth, as well as all matrices with linear depth:

Proposition 1. *$(BP)^1$ captures the fast Fourier transform, the fast Hadamard transform, and their inverses exactly. $(BP)^2$ captures the DCT, DST, and convolution exactly. All $N \times N$ matrices are contained in $(BP)_2^{4N+10}$.*

Proposition 1 is shown in Appendix B. We suggest some additional conjectures about the expressiveness of the BP hierarchy in Appendix D.

Even though the BP parameterization is expressive, it still retains the learnability characteristic of compressed parameterizations. In fact, neural networks comprising layers of BP and BPBP matrices still have VC dimension that is almost linear in the number of parameters (Appendix B), similar to networks with fully-connected layers (Bartlett et al., 1999; Harvey et al., 2017) and LDR (Thomas et al., 2018), which implies a corresponding sample complexity bound.

4. Empirical Evaluation

We evaluate the proposed approach to verify that our butterfly parameterization can both recover fast transforms and be integrated as an effective component in ML pipelines[3]. In Section 4.1, we confirm that it automatically learns the fast algorithms for many discrete transforms commonly used in signal processing and machine learning. Section 4.2 further shows that it can be a useful component to increase the performance of deep learning models while ensuring fast multiplication and few parameters by design.

4.1. Discrete Transforms

Below we list several important classes of structured matrices. Some of them are directly captured by our parametriza-

[3]Code to reproduce experiments and plots is available at https://github.com/HazyResearch/learning-circuits

tion and we expect that they can be recovered close to perfectly, thus providing a $O(N \log N)$ algorithm that closely approximates the naive $O(N^2)$ matrix multiplication. Others are not perfectly captured by the BPBP class but still have recursive structure; for these, we expect that our method reconstructs them better than standard matrix compression methods (sparse, low-rank, and combinations) can.

Transforms We describe the matrices we evaluate on and their applications; a standard reference is Proakis (2001). Their explicit formulas are in Appendix A, Table 3.

1. Discrete Fourier transform (DFT): arguably the most important computational tool in signal processing, the FFT is one of the top 10 algorithms of the 20th century (Dongarra & Sullivan, 2000).
2. Discrete cosine transform (DCT): it expresses the input vector in the basis of cosine functions. It finds use in lossy compression of audio (MP3) and images (JPEG), in speech processing, and in numerical methods of solving partial differential equations (PDEs).
3. Discrete sine transform (DST): similar to the DCT, it expresses the input vector as a linear combination of sine functions. It is widely employed in spectral methods to solve PDEs.
4. Convolution: widely used in statistics, image processing, computer vision, and natural language processing.
5. Hadamard transform: commonly used in quantum information processing algorithms, and in ML as a fast random projection or kernel approximation method.
6. Discrete Hartley transform: similar to the DFT, but it transforms real inputs to real outputs. It was designed as a more efficient option than the DFT for real data.

Methods We assume that the transform $\mathcal{T}$ is fully-specified, e.g., from N linearly independent input-output pairs from which the matrix representation $T_N \in \mathbb{F}^{N \times N}$ can be computed.

To recover a fast algorithm of the transform, we wish to approximate T_N with the product of one or more blocks of butterfly and permutation products, by minimizing the Frobenius norm of the difference:

$$\text{minimize} \frac{1}{N^2} \left\| T_N - B^{(N)} P^{(N)} \right\|_F^2 . \tag{3}$$

By design, this factorization yields a fast $O(N \log N)$ algorithm for the transform.

We also compare to standard baselines for matrix factorization, maintaining the same total sparsity budget (i.e. computation cost of a multiplication) for each:

1. Sparse: this is the same as choosing the largest s entries where s is the sparsity budget.
2. Low-rank: the sparsity budget is used in the parameters of the low-rank factors, which can be found with a truncated SVD.
3. Sparse + low-rank: $\|T_N - S - L\|^2$ is minimized, where S is sparse and L is low-rank, by solving a convex problem.[4] This is commonly known as robust PCA (Candès et al., 2011).

Experimental procedure We use the Adam optimizer (Kingma & Welling, 2014) to minimize the Frobenius norm of the error, and use Hyperband (Li et al., 2017) to automatically tune the hyperparameters (learning rates, random seed for initialization). The runs are stopped early if the average per entry difference (aka RMSE) $\sqrt{\frac{1}{N^2} \left\| T_N - B^{(N)} P^{(N)} \right\|_F^2}$ is low enough: we consider RMSE below 1e-4 (corresponding to the objective in (3) below 1e-8, while we use 32-bit floats with machine epsilon around 6e-8) to mean that we successfully recover the fast algorithms for the transforms to machine precision. For consistency, we consider the unitary or orthogonal scaling of these transforms such that they have norm on the order of 1.0. For the DCT and DST, we add another simple permutation for extra learnability. All transforms considered learn over BP except for convolution which uses BPBP. All methods are optimized over complex entries.

Since the forward mapping of our butterfly parameterization is differentiable with respect to the entries of the butterfly matrices and the logits of the permutations, gradients are easily obtained with the help of an auto-differentiation framework. We provide our code in PyTorch.

Quality Figure 3 visualizes the lowest error found by Hyperband for various matrix dimensions and several methods. Full numerical results are provided in Appendix C. As shown, we successfully recover the fast algorithms for these transforms up to $N = 512$ for convolution and $N = 1024$ for other transforms. For example, the matrix factorization procedure recovers the bit-reversal permutation applied at the beginning of the Cooley-Tukey fast Fourier transform. It also discovers many other unconventional permutations that also lead to exact factorization of the FFT.

We note that there are other transforms not captured by our parameterization. Orthogonal polynomial transforms, such as the discrete Legendre transform (DLT), are known only to have fast $O(N \log^2 N)$ algorithms. They follow a slightly more general divide-and-conquer decomposition that we elaborate on in Appendix A.6. As expected, we find that the butterfly parameterization does not perfectly capture the DLT, but does recover it slightly better than the baselines.

[4] Although there is an extra addition, this can also be written as a sparse product of 3 matrices by adding auxiliary identity blocks.

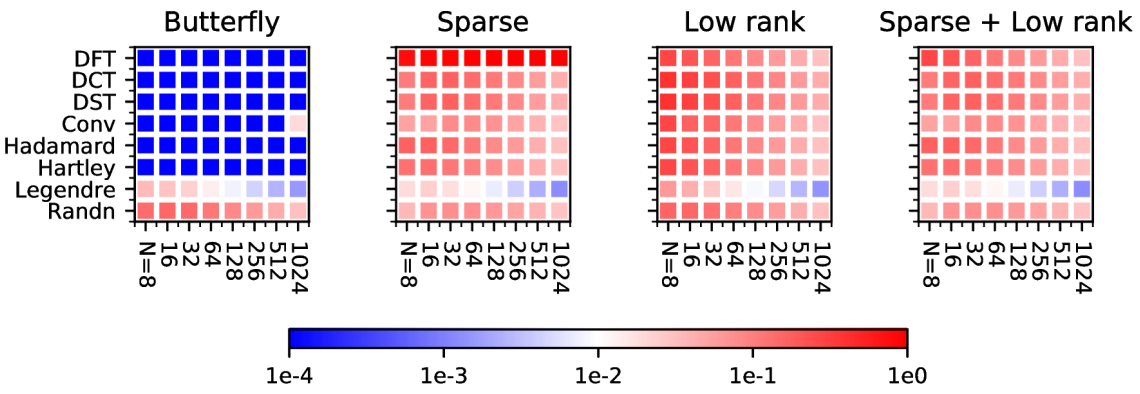


Figure 3. RMSE of learning fast algorithms for common transforms, with early stopping when RMSE is below 1e-4. (Blue is better and red is worse.) Our butterfly parameterization can recover common transforms up to $N = 1024$ and convolutions up to $N = 512$. Explicit formulas for each transform are listed in Appendix A, Table 3.

Figure 3 also includes a baseline row factoring a matrix of appropriately scaled i.i.d. Gaussian entries, to indicate typical errors for factoring an unstructured matrix.

4.2. Neural Network Compression

Many structured matrix approaches have been proposed to replace fully-connected (FC) layers of neural networks, to speed up training and inference, and to reduce the memory consumption. These structured matrices are cleverly designed by combining commonly used fast transforms. For example, Fastfood (Le et al., 2013) and Deep Fried Convnets (Yang et al., 2015) compose the fast Hadamard transform and fast Fourier transforms, and Sindhwani et al. (2015) use Toeplitz-like matrices that can be written as a sequence of 2 or 4 FFTs. However, the design choice for these light-weight replacement layers is restricted by the set of known and implementable transforms.

On the first benchmark task of compressing a single hidden layer model, the real version of BPBP has better classification accuracy than a fully-connected layer on all datasets tested, and uses more than 56X fewer parameters (Table 1); the complex version performs even better with a slight parameter increase. The previous best methods fail to achieve this on the more challenging CIFAR-10 dataset at the same parameter budget (Thomas et al., 2018). We further demonstrate that this layer is effective as a lightweight addition to a larger-scale ResNet architecture.

Fully-connected Previous work showed that structured matrix approaches based on the low displacement rank framework, including Toeplitz-like (Sindhwani et al., 2015), LDR-SD and LDR-TD matrices (Thomas et al., 2018), compare very favorably to other compression approaches. Following previous experimental settings (Chen et al., 2015; Sindhwani et al., 2015; Thomas et al., 2018), we compare our proposed classes to several baselines using dense structured matrices to compress the hidden layer of a single hidden layer neural network. Competing methods include simple low-rank factorizations (Denil et al., 2013), circulant matrices (equivalent to 1-dimensional convolutions) (Cheng et al., 2015), the adaptive Fastfood transform (Yang et al., 2015), and low displacement rank methods (Sindhwani et al., 2015; Thomas et al., 2018) which implicitly define a structured matrix through a displacement equation and admit specialized fast divide-and-conquer algorithms (De Sa et al., 2018). Our implementation is built on top of the publicly available implementation of Thomas et al. (2018) with the same hyperparameters, and we report their numbers for the competing baseline methods directly. We test on the three main datasets from Thomas et al. (2018): two challenging variants of MNIST—one with randomly rotated images and random background, the other with correlated background noise—and the standard CIFAR-10 dataset.

Table 1 reports results for variants of our butterfly parametrization, compared to the unstructured matrix baseline and other structured matrix approaches. Notably, the butterfly methods achieve higher classification accuracy than the fully-connected layer on all datasets and are highly competitive with the other approaches.

We note that improvements over unconstrained matrices can arise from lower generalization error due to fewer parameters (relating to VC bounds, Proposition 2), or better inductive bias encoded by the structured class. For example, convolutions are important in image tasks due to encoding shift equivariance, and Thomas et al. (2018) hypothesize that their structured classes improve over FC layers through imposing approximate equivariance to more general transformations. Since our BP parametrization can represent arbitrary convolutions, it can encode these important priors.

Table 1. Test accuracy when replacing the hidden layer with structured classes. For the BPBP methods, the permutations P have been fixed to the bit-reversal permutation. The butterfly parameterization achieves higher accuracy than the unstructured layer on all datasets.

Method	**MNIST-bg-rot**	**MNIST-noise**	**CIFAR-10**	*Compression factor*
Unstructured	44.08	65.15	46.03	*1*
BPBP (complex, fixed permutation)	**46.26**	77.00	**49.93**	*39.4*
BPBP (real, fixed permutation)	46.16	75.00	48.69	*56.9*
LDR-TD (Thomas et al., 2018)	45.81	**78.45**	45.33	*56.9*
Toeplitz-like (Sindhwani et al., 2015)	42.67	75.75	41.78	*56.9*
Fastfood (Yang et al., 2015)	38.13	63.55	39.64	*78.7*
Circulant (Cheng et al., 2015)	34.46	65.35	34.28	*93.0*
Low-rank (Denil et al., 2013)	35.67	52.25	32.28	*56.9*

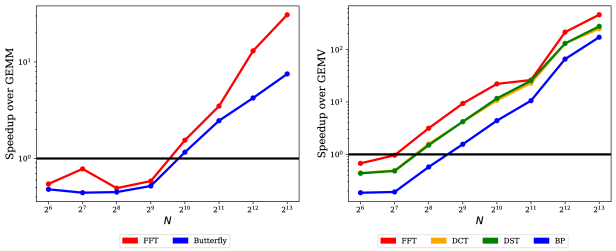
ResNet In addition to the standard single hidden layer benchmarks, we test the effect of using butterfly layers in a standard ResNet18 (He et al., 2016) implementation on the CIFAR-10 dataset. This architecture is normally fully convolutional, ending with a FC layer of dimensions 512×10 before the softmax. However, we experiment with adding an additional FC or structured layer right before this final FC layer. Table 2 shows that the ResNet18 architecture can benefit from an additional fully connected layer, and using a BPBP layer instead improves performance even more while adding a negligible (0.07% increase) number of parameters to the original model.

Table 2. Classification accuracy for the ResNet18 architecture with different layers inserted before the final FC/softmax layer.

Last layer	None	FC	BPBP
Accuracy	93.58 ± 0.15	93.89 ± 0.19	**94.01** ± 0.09

4.3. Training and Inference Speed Comparison

By design, the BP parameterization yields a fast algorithm of complexity $O(N \log N)$, no matter which transform it learns. Moreover, given the parameters of the BP model, it is easy to implement this fast algorithm (this can be done in 5 lines of Python, and our code provides a function to do this automatically). The BP parameterization captures many common transforms (Section 4.1), and its implementation makes no transform-specific optimizations. Nevertheless, our simple implementation is surprisingly competitive with hand-tuned kernels both for training and for inference (after the parameters of the BP model are learned and we wish to evaluate BPx for new input x). In Figure 4, we compare the speed of the BP fast multiplication against specialized implementation of common transforms such as the FFT, DCT, and DST (all have complexity $O(N \log N)$), using dense matrix-vector multiply (GEMV, complexity $O(N^2)$) as a baseline. For training with realistic input sizes $N = 1024$ and batch size 256 on GPU, the training time (forward and backward) of butterfly matrix is 15% faster than dense matrix multiply (GEMM from cuBLAS) and within 40% of FFT (from cuFFT). For inference on CPU, the BP fast multiplication can be one or two orders of magnitude faster than GEMV, is within a factor of 5 of the FFT, and is within a factor of 3 of the DCT and the DST, across a range of input sizes. The GEMM/GEMV and the FFT are two of the most heavily tuned numerical routines.



(a) Training (b) Inference

Figure 4. Speedup of FFT and Butterfly against dense matrix-matrix multiply (GEMM) for training, and FFT, DCT, DST, and BP against dense matrix-vector multiply (GEMV) for inference. Butterfly's performance is constant with respect to any of the possible transforms it can learn, in contrast to the highly tuned implementations for specific transforms.

5. Conclusion

We address the problem of automatically learning fast algorithms for a class of important linear transforms, through a parameterization of recursive algorithms via butterfly factorizations. We validate our method by learning transforms including the DFT, DCT, Hadamard transform, and convolutions up to machine precision and dimension $N = 1024$. Finally, we show that the same method yields consistent performance improvements and substantial compression and speed increases as a component of end-to-end ML models.

Acknowledgments

We thank Maximilian Lam for his help with early experiments.

We gratefully acknowledge the support of DARPA under Nos. FA87501720095 (D3M) and FA86501827865 (SDH), NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF1763315 (Beyond Sparsity) and CCF1563078 (Volume to Velocity), ONR under No. N000141712266 (Unifying Weak Supervision), the Moore Foundation, NXP, Xilinx, LETI-CEA, Intel, Google, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, the Okawa Foundation, and American Family Insurance, Google Cloud, Swiss Re, and members of the Stanford DAWN project: Intel, Microsoft, Teradata, Facebook, Google, Ant Financial, NEC, SAP, VMWare, and Infosys. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA, NIH, ONR, or the U.S. Government. Matthew Eichhorn and Atri Rudra's research is supported by NSF grant CCF-1763481.

References

Bartlett, P. L., Maiorov, V., and Meir, R. Almost linear VC dimension bounds for piecewise polynomial networks. In *Advances in Neural Information Processing Systems*, pp. 190–196, 1999.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. 2016.

Bürgisser, P., Clausen, M., and Shokrollahi, M. A. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 2013.

Candès, E. J., Li, X., Ma, Y., and Wright, J. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3): 11, 2011.

Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. Compressing neural networks with the hashing trick. In Bach, F. and Blei, D. (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2285–2294, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/chenc15.html.

Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S.-F. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2857–2865, 2015.

De Sa, C., Gu, A., Puttagunta, R., Ré, C., and Rudra, A. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1060–1079. SIAM, 2018.

Denil, M., Shakibi, B., Dinh, L., De Freitas, N., et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pp. 2148–2156, 2013.

Ding, C., Liao, S., Wang, Y., Li, Z., Liu, N., Zhuo, Y., Wang, C., Qian, X., Bai, Y., Yuan, G., et al. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 395–408. ACM, 2017.

Dongarra, J. and Sullivan, F. Guest editors' introduction: The top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, 2000.

Driscoll, J. R., Healy, Jr., D. M., and Rockmore, D. N. Fast discrete polynomial transforms with applications to data analysis for distance transitive graphs. *SIAM J. Comput.*, 26(4):1066–1099, August 1997. ISSN 0097-5397. doi: 10.1137/S0097539792240121. URL http://dx.doi.org/10.1137/S0097539792240121.

Egner, S. and Püschel, M. Automatic generation of fast discrete signal transforms. *IEEE Transactions on Signal Processing*, 49(9):1992–2002, 2001.

Egner, S. and Püschel, M. Symmetry-based matrix factorization. *Journal of Symbolic Computation*, 37(2):157–186, 2004.

Grandvalet, Y. and Bengio, Y. Semi-supervised learning by entropy minimization. In Saul, L. K., Weiss, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems 17*, pp. 529–536. MIT Press, 2005.

Grover, A., Wang, E., Zweig, A., and Ermon, S. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1eSS3CcKX.

Harvey, N., Liaw, C., and Mehrabian, A. Nearly-tight VC-dimension bounds for piecewise linear neural networks. In Kale, S. and Shamir, O. (eds.), *Proceedings of the 2017 Conference on Learning Theory*, volume 65 of *Proceedings of Machine Learning Research*, pp. 1064–1068, Amsterdam, Netherlands, 07–10 Jul

2017. PMLR. URL http://proceedings.mlr.press/v65/harvey17a.html.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Jing, L., Shen, Y., Dubcek, T., Peurifoy, J., Skirlo, S., LeCun, Y., Tegmark, M., and Soljačić, M. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1733–1741. JMLR. org, 2017.

Jurafsky, D. and Martin, J. H. *Speech and language processing*, volume 3. Pearson London, 2014.

Kailath, T., Kung, S.-Y., and Morf, M. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications*, 68(2):395–407, 1979.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, April 2014.

Le, Q., Sarlos, T., and Smola, A. Fastfood - computing Hilbert space expansions in loglinear time. In Dasgupta, S. and McAllester, D. (eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 244–252, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL http://proceedings.mlr.press/v28/le13.html.

Le Magoarou, L. and Gribonval, R. Chasing butterflies: In search of efficient dictionaries. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3287–3291, April 2015. doi: 10.1109/ICASSP.2015.7178579.

Le Magoarou, L. and Gribonval, R. Flexible multilayer sparse approximations of matrices and applications. *IEEE Journal of Selected Topics in Signal Processing*, 10(4): 688–700, June 2016. ISSN 1932-4553. doi: 10.1109/JSTSP.2016.2543461.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

Li, Y., Yang, H., Martin, E. R., Ho, K. L., and Ying, L. Butterfly factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015.

Mairal, J., Ponce, J., Sapiro, G., Zisserman, A., and Bach, F. R. Supervised dictionary learning. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems 21*, pp. 1033–1040. Curran Associates, Inc., 2009. URL http://papers.nips.cc/paper/3448-supervised-dictionary-learning.pdf.

Makhoul, J. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(1):27–34, February 1980. ISSN 0096-3518. doi: 10.1109/TASSP.1980.1163351.

Mena, G., Belanger, D., Linderman, S., and Snoek, J. Learning latent permutations with Gumbel-Sinkhorn networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=Byt3oJ-0W.

Munkhoeva, M., Kapushev, Y., Burnaev, E., and Oseledets, I. Quadrature-based features for kernel approximation. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 9165–9174. Curran Associates, Inc., 2018.

Neyshabur, B. and Panigrahy, R. Sparse matrix factorization. *arXiv preprint arXiv:1311.3315*, 2013.

Pan, V. Y. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 0-8176-4240-4.

Parker, D. S. Random butterfly transformations with applications in computational linear algebra. 1995.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pp. 1310–1318, 2013.

Proakis, J. G. *Digital signal processing: principles algorithms and applications*. Pearson Education India, 2001.

Puschel, M. and Moura, J. M. Algebraic signal processing theory. *IEEE Transactions on Signal Processing*, 56(8): 3572–3585, 2008.

Sindhwani, V., Sainath, T., and Kumar, S. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*, pp. 3088–3096, 2015.

Szegö, G. *Orthogonal Polynomials*. Number v. 23 in American Mathematical Society colloquium publications. American Mathematical Society, 1967. ISBN 9780821889527. URL https://books.google.com/books?id=3hcW8HBh7gsC.

Thomas, A., Gu, A., Dao, T., Rudra, A., and Ré, C. Learning compressed transforms with low displacement rank. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 9066–9078. Curran Associates, Inc., 2018.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., and Blunsom, P. Neural arithmetic logic units. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 8046–8055. Curran Associates, Inc., 2018.

Tschannen, M., Khanna, A., and Anandkumar, A. StrassenNets: Deep learning with a multiplication budget. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4985–4994. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/tschannen18a.html.

Voronenko, Y. and Puschel, M. Algebraic signal processing theory: Cooley–Tukey type algorithms for real DFTs. *IEEE Transactions on Signal Processing*, 57(1):205–222, 2009.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Van den Broeck, G. A semantic loss function for deep learning with symbolic knowledge. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5502–5511. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/xu18h.html.

Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., and Wang, Z. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1476–1483, 2015.

Ye, K. and Lim, L.-H. Every matrix is a product of toeplitz matrices. *Foundations of Computational Mathematics*, 16(3):577–598, Jun 2016. ISSN 1615-3383. doi: 10.1007/s10208-015-9254-z. URL https://doi.org/10.1007/s10208-015-9254-z.

Yu, F. X., Kumar, S., Rowley, H. A., and Chang, S. Compact nonlinear maps and circulant extensions. *CoRR*, abs/1503.03893, 2015.

Yu, F. X. X., Suresh, A. T., Choromanski, K. M., Holtmann-Rice, D. N., and Kumar, S. Orthogonal random features. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 1975–1983. Curran Associates, Inc., 2016.

Zou, H., Hastie, T., and Tibshirani, R. Sparse principal component analysis. *Journal of computational and graphical statistics*, 15(2):265–286, 2006.

Table 3. Formulas for transforms considered in Section 4.1 and Figure 3.

Transform	Formula
DFT	$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N}nk}$
DCT	$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right]$
DST	$X_k = \sum_{n=0}^{N-1} x_n \sin\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)(k+1)\right]$
Convolution	$X_k = \sum_{n=0}^{N-1} x_n g_{k-n}$
Hadamard	$H_1 = 1$, $H_m = \frac{1}{\sqrt{2}}\begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix}$
Hartley	$X_k = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi}{N}nk\right) + \sin\left(\frac{2\pi}{N}nk\right)\right]$
Legendre	$X_k = \sum_{n=0}^{N-1} x_n L_k(2n/N-1)$, $L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1)^k$
Randn	$(T_N)_{ij} \sim \mathcal{N}(1, \frac{1}{N})$

A. Matrix Factorizations of Linear Transforms

Table 3 summarizes the transforms considered in Section 4.1. In general, they transform a (real or complex) vector $x = [x_0, \ldots, x_{N-1}]$ into another (real or complex) vector $X = [X_0, \ldots, X_{N-1}]$ by expressing the input signal in terms of another set of basis.

A.1. Discrete Cosine Transform (DCT) Matrix

The DCT of a vector $x \in \mathbb{R}^N$ is defined as

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right], \qquad k = 0, \ldots, N-1.$$

As described in Makhoul (1980), the DCT of x can be written in terms of the FFT of order N. To do this, we permute x into v by separating the even and odd indices and reversing the odd indices (e.g. $[0,1,2,3] \to [0,2,3,1]$), taking the FFT of v to obtain V, and multiplying each V_k ($k = 0, \ldots, N-1$) by $2e^{-\frac{i\pi k}{2N}}$ and taking the real part to get X_k.

Written in terms of matrix factorization:

$$DCT_N = \Re\,\mathbf{diag}\left(2e^{-\frac{i\pi k}{2N}}\right) F_N P',$$

where $\Re$ takes the real part and P' is a permutation matrix (the permutation done at the beginning of the DCT). Recall that F_N has the form

$$F_N = B_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_2 \end{bmatrix} P,$$

where P is the bit-reversal permutation matrix. $\mathbf{diag}\left(2e^{-\frac{i\pi k}{2N}}\right)$ can be combined with B_N to form another butterfly factor B'_N. Thus the DCT has this factorization:

$$DCT_N = \Re\, B'_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_2 \end{bmatrix} PP'.$$

This is a BP^2 factorization (with the additional final step of computing the real part) with the left BP performing the FFT and final scaling, the right butterfly matrix as the identity, and the right permutation matrix as the permutation at the beginning of the DCT.

A.2. Discrete Sine Transform (DST) Matrix

The DST of a vector $x \in \mathbb{R}^N$ is defined as

$$X_k = \sum_{n=0}^{N-1} x_n \sin\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)(k+1)\right], \qquad k = 0, \ldots, N-1.$$

Just as with the DCT, we express the DST of x in terms of the FFT of order N. First, we permute x into v by separating the even and odd indices and reversing the odd indices (e.g. $[0,1,2,3] \to [0,2,3,1]$). However, since sine is an odd function, we must negate those elements in the second half of v. Next, we take the FFT of v to obtain V. Finally multiply each V_k ($k = 0, \ldots, N-1$) by $2ie^{-\frac{i\pi k}{2N}}$ and take the real part to get X_k.

Written in terms of matrix factorization:

$$DST_N = \Re\,\mathbf{diag}\left(2ie^{-\frac{i\pi k}{2N}}\right) F_N D P',$$

where $\Re$ takes the real part, D is the matrix $\begin{bmatrix} I_{N/2} & 0 \\ 0 & -I_{N/2} \end{bmatrix}$ and P' is a permutation matrix (the permutation done at the beginning of the DST). Recall that F_N has the form

$$F_N = B_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_2 \end{bmatrix} P,$$

where P is the bit-reversal permutation matrix. We may combine $\mathbf{diag}\left(2ie^{-\frac{i\pi k}{2N}}\right)$ with b_N to obtain a new butterfly factor, which we call B'_N. Thus the DST has this factorization:

$$DST_N = \Re\, B'_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_2 \end{bmatrix} PDP'.$$

Note that any diagonal matrix (e.g. D) is trivially representable as a butterfly matrix. Hence, this factorization of the DST is a BP^2 factorization (with the additional final step of computing the real part) with the left BP performing the FFT and final scaling, the right butterfly matrix as D, and the right permutation matrix as the permutation at the beginning of the DST.

A.3. Hadamard Matrix

The Hadamard matrix (for powers of 2) is defined recursively as $H_1 = 1$, and $H_N = \begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix}$. Thus we have the recursive factorization:

$$H_N = \begin{bmatrix} I_{N/2} & I_{N/2} \\ I_{N/2} & -I_{N/2} \end{bmatrix} \begin{bmatrix} H_{N/2} & 0 \\ 0 & H_{N/2} \end{bmatrix},$$

which is a BP factorization with each butterfly factor, $B_{N/2^k} = \begin{bmatrix} I_{N/2^{k+1}} & I_{N/2^{k+1}} \\ I_{N/2^{k+1}} & -I_{N/2^{k+1}} \end{bmatrix}$ and with permutation matrix $P^{(N)} = I_N$. Here, the entries of the butterfly factors may be real, instead of complex.

A.4. Convolution

Here we apply the decomposition of FFT to see if we can learn the decomposition of fast convolution.

Suppose we have a fixed vector $h \in \mathbb{C}^N$ and the linear map we're interested in is $x \mapsto h * x$. We can write this convolution with h explicitly as a *circulant* matrix:

$$A = \begin{bmatrix} h_0 & h_{N-1} & \cdots & h_2 & h_1 \\ h_1 & h_0 & h_{N-1} & & h_2 \\ \vdots & h_1 & h_0 & \ddots & \vdots \\ h_{N-2} & & \ddots & \ddots & h_{N-1} \\ h_{N-1} & h_{N-2} & \cdots & h_1 & h_0 \end{bmatrix}.$$

We can compute convolution by the DFT:

$$Ax = F_N^{-1}((F_N h) \odot (F_N x)),$$

where F_N^{-1} denotes the inverse Fourier transform matrix where $(F_N^{-1}) = \frac{1}{N}\omega_N^{ij}$ and $\odot$ denotes elementwise multiplication. Since h is just some fixed vector, elementwise multiplication with $F_N h$ is just multiplication by some fixed diagonal matrix D. Then

$$Ax = F_N^{-1} D F_N x.$$

Note that the inverse Fourier transform has the same algorithm, and thus the same factorization, as the Fourier transform (with different twiddle factors, ω_N^{ij} instead of ω_N^{-ij}). Hence, we can express

$$A = \frac{1}{N}\tilde{B}_N \begin{bmatrix} \tilde{B}_{N/2} & 0 \\ 0 & \tilde{B}_{N/2} \end{bmatrix} \cdots \begin{bmatrix} \tilde{B}_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \tilde{B}_2 \end{bmatrix} PDB_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & B_2 \end{bmatrix} P,$$

where P is the bit-reversal permutation. We may fold the $\frac{1}{N}$ into $\tilde{B}_N$ to obtain a new butterfly factor $\tilde{B}'_N$, and we may similarly fold the diagonal matrix D into B_N to obtain a new butterfly factor B'_N. Hence, our final factorization of convolution / the circulant matrix is :

$$A = \tilde{B}'_N \begin{bmatrix} \tilde{B}_{N/2} & 0 \\ 0 & \tilde{B}_{N/2} \end{bmatrix} \cdots \begin{bmatrix} \tilde{B}_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \tilde{B}_2 \end{bmatrix} PB'_N \begin{bmatrix} B_{N/2} & 0 \\ 0 & B_{N/2} \end{bmatrix} \cdots \begin{bmatrix} B_2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & B_2 \end{bmatrix} P,$$

which is a $(\text{BP})^2$ factorization.

Similarly, the skew-circulant matrix also lies in $(\text{BP})^2$:

$$A = \begin{bmatrix} h_0 & -h_{N-1} & \dots & -h_2 & -h_1 \\ h_1 & h_0 & -h_{N-1} & & -h_2 \\ \vdots & h_1 & h_0 & \ddots & \vdots \\ h_{N-2} & & \ddots & \ddots & -h_{N-1} \\ h_{N-1} & h_{N-2} & \dots & h_1 & h_0 \end{bmatrix}.$$

A.5. Toeplitz Matrices

Let T_N be the Toeplitz matrix:

$$T_N = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{-N+2} & t_{-N+1} \\ t_1 & t_0 & t_{-1} & & t_{-N+2} \\ \dots & t_1 & t_0 & \ddots & \dots \\ t_{N-2} & & \ddots & \ddots & t_{-1} \\ t_{N-1} & t_{N-2} & \dots & t_1 & t_0 \end{bmatrix}.$$

Define $\tilde{T}_N$ to be:

$$\tilde{T}_N = \begin{bmatrix} 0 & t_{N-1} & \dots & t_2 & t_1 \\ t_{-N+1} & 0 & t_{N-1} & & t_2 \\ \dots & t_{-N+1} & 0 & \ddots & \dots \\ t_{-2} & & \ddots & \ddots & t_{N-1} \\ t_{-1} & t_{-2} & \dots & t_{-N+1} & 0 \end{bmatrix}.$$

Then, $T_N = \begin{bmatrix} I_N & 0 \end{bmatrix} \begin{bmatrix} T_N & \tilde{T}_N \\ \tilde{T}_N & T_N \end{bmatrix} \begin{bmatrix} I_N \\ 0 \end{bmatrix}$. Note that the inner matrix is a $2N \times 2N$ circulant matrix that can be decomposed into a $(\text{BP})^2$ factorization as described in A.4. Therefore, our final factorization for Toeplitz matrices is contained within $(\text{BP})_2^2$.

A.6. Orthogonal Polynomial Matrices

Although the ability to represent general orthogonal polynomial matrices in terms of butterfly matrices is left as an open problem, we nonetheless present an alternate sparse factorization.

Definition 2. A family of polynomials $\{p\} = p_0(x), p_1(x), \ldots \in \mathbb{R}[x]$ is *orthogonal* over $\mathbb{R}$ if:

- $p_0(x) = c_1$
- $p_1(x) = a_1 x + b_1$
- $p_i(x) = (a_i x + b_i) p_{i-1}(x) + c_i\, p_{i-2}(x)$ for all $i \geq 2$

We say that $\{p\}$ is parameterized by real sequences $\{a_i, b_i, c_i : i \in \mathbb{N}\}$ (with c_1 and each $a_i \in \mathbb{R} \setminus \{0\}$).

Definition 3. Given a family of orthogonal polynomials $\{p\}$, we may define the *orthogonal polynomial matrix* $P_{[s:n]} \in \mathbb{R}^{(n-s)\times n}$ such that:

$$p_{s+i} = \sum_{j=0}^{n} \left(P_{[s:n]}\right)_{ij} x^j, \qquad 0 \leq i < n - s$$

For sake of clarify, we formulate the decomposition using matrices of polynomials. We note that each polynomial entry with degree bounded by d may be expanded into a $d \times 2d$ Toeplitz convolution matrix if one desires matrices of real coefficients.

For a given family of orthogonal polynomials $\{p\}$ parameterized by $\{a_j, b_j, c_j : 1 \leq j \leq n-1\}$, let $T_j \in \mathbb{R}[x]^{2\times 2}$ be a *transition matrix* defined by:

$$\begin{bmatrix} a_j x + b_j & c_j \\ 1 & 0 \end{bmatrix}.$$

For convenience of notation, let $T_0 = I$. Let $T_{[\ell,r]} \in \mathbb{R}[x]^{2\times 2}$ be a *transition product matrix* defined by:

$$T_{[\ell:r]} \equiv T_\ell \cdot T_{(\ell-1)} \ldots T_{(r+1)} \cdot T_r \equiv \begin{bmatrix} A_{[\ell:r]}(x) & B_{[\ell:r]}(x) \\ C_{[\ell:r]}(x) & D_{[\ell:r]}(x) \end{bmatrix}.$$

From these definitions, we see that for all $j \geq 0$,

$$\begin{bmatrix} p_{j+1}(x) \\ p_j(x) \end{bmatrix} = T_j \begin{bmatrix} p_j(x) \\ p_{j-1}(x) \end{bmatrix} = T_{[j:0]} \begin{bmatrix} p_1(x) \\ p_0(x) \end{bmatrix}.$$

We use this to formulate the following decomposition of the orthogonal polynomial matrix $P_{[0:n]}$.

$$P_{[0:n]} = \underbrace{\begin{bmatrix} p_0(x) \\ p_1(x) \\ \vdots \\ p_{n-1}(x) \end{bmatrix}}_{n\times 1} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 \end{bmatrix}}_{n\times 2n} \underbrace{\begin{bmatrix} T_{[0:0]} \\ T_{[1:0]} \\ \vdots \\ T_{[n-1:0]} \end{bmatrix}}_{2n\times 2} \underbrace{\begin{bmatrix} p_1(x) \\ p_0(x) \end{bmatrix}}_{2\times 1}. \tag{4}$$

The first "stretched" identity matrix serves the function of selecting every other entry from the vector of $2n$ polynomials to its right. We focus our attention on the middle matrix. Noting that $T_{[\ell:r]} = T_{[\ell:m]} \cdot T_{[m-1:r]}$ for any $r \leq m \leq \ell$, we may represent this block matrix as:

$$\underbrace{\begin{bmatrix} T_{[0:0]} \\ T_{[1:0]} \\ \vdots \\ T_{[n-1:0]} \end{bmatrix}}_{2n\times 2} = \underbrace{\begin{bmatrix} T_{[0:0]} \\ \vdots \\ T_{[\frac{n}{2}-1:0]} \\ \mathbf{0}_{n\times 2} \end{bmatrix}}_{2n\times 2} + \underbrace{\begin{bmatrix} \mathbf{0}_{n\times 2} \\ T_{[\frac{n}{2}:\frac{n}{2}]} \\ \vdots \\ T_{[n-1:\frac{n}{2}]} \end{bmatrix}}_{2n\times 2} \underbrace{\begin{bmatrix} T_{[\frac{n}{2}-1:0]} \end{bmatrix}}_{2\times 2} = \underbrace{\begin{bmatrix} T_{[0:0]} & \mathbf{0}_{n\times 2} \\ \vdots & \\ T_{[\frac{n}{2}-1:0]} & \\ & T_{[\frac{n}{2}:\frac{n}{2}]} \\ \mathbf{0}_{n\times 2} & \vdots \\ & T_{[n-1:\frac{n}{2}]} \end{bmatrix}}_{2n\times 4} \underbrace{\begin{bmatrix} \mathbf{I}_{2\times 2} \\ T_{[\frac{n}{2}-1:0]} \end{bmatrix}}_{4\times 2}. \tag{5}$$

Notice that the left matrix in this last expression consists of two matrices with the same structure as the first expression, but of half the size. Hence, we may repeat the same decomposition on each of the sub-matrices. In general, the decomposition becomes:

$$\underbrace{\begin{bmatrix} T_{[0:0]} \\ T_{[1:0]} \\ \vdots \\ T_{[n-1:0]} \end{bmatrix}}_{2n \times 2} = \underbrace{\begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} & \dots & \mathbf{0}_{2\times 2} \\ T_1 & \mathbf{0}_{2\times 2} & \dots & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{I}_{2\times 2} & \dots & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & T_3 & \dots & \mathbf{0}_{2\times 2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{2\times 2} & \mathbf{0}_{2\times 2} & \dots & \mathbf{I}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{0}_{2\times 2} & \dots & T_{n-1} \end{bmatrix}}_{2n \times n} \cdots \underbrace{\begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} \\ T_{[\frac{n}{4}-1:0]} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{I}_{2\times 2} \\ \mathbf{0}_{2\times 2} & T_{[\frac{3n}{4}-1:\frac{n}{2}]} \end{bmatrix}}_{8 \times 4} \underbrace{\begin{bmatrix} \mathbf{I}_{2\times 2} \\ T_{[\frac{n}{2}-1:0]} \end{bmatrix}}_{4 \times 2}. \tag{6}$$

Discrete Legendre Transform The Discrete Legendre Transform (DLT) of a vector $x \in \mathbb{R}^N$ is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n L_k \left(\frac{2n}{N-1} \right),$$

where L_k is the k'th Legendre polynomial. The Legendre polynomials are a family of orthogonal polynomials with:

$$L_0(x) = 1 \qquad L_1(x) = x \qquad L_k(x) = \left(\tfrac{2k-1}{k}\right) x L_{k-1}(x) - \left(\tfrac{k-1}{k}\right) L_{k-2}(x), \quad k \geq 2.$$

Hence, the DLT may be factored as described above.

B. Proofs

B.1. VC Dimension Bound for Neural Network with Butterfly Layers

Proposition 2. *Let $\mathcal{F}$ denote the class of neural networks with L layers, each is a butterfly layer using the BP or BPBP parameterization, with fixed permutation, W total parameters, and piecewise linear activations. Let* $\operatorname{sign}\mathcal{F}$ *denote the corresponding classification functions, i.e. $\{x \mapsto \operatorname{sign} f(x) : f \in \mathcal{F}\}$. The VC dimension of this class is*

$$\operatorname{VCdim}(\operatorname{sign}\mathcal{F}) = O(LW \log W).$$

Because the parameters within a layer interact multiplicatively, the standard VC dimension bound for fully-connected layers (Bartlett et al., 1999; Harvey et al., 2017) does not apply directly. However, a variant of the same argument applies to the case where degree of multiplicative interaction is not too high (Thomas et al., 2018, Theorem 3).

We provide a short proof of the VC dimension bound for neural networks with BP or BP^2 layers based on this result.

Proof. Theorem 3 of Thomas et al. (2018) requires that the entries of the linear layer, as polynomials of the parameters, has degree at most $c_1 m_l^{c_2}$ for some universal constant $c_1, c_2 > 0$, where m_l is the size of output of the l-th layer. In our case, the BP or BPBP parameterization with fixed permutation has total degree at most $2 \log_2 n$ in the parameters of B, where n is

Table 4. RMSE of learning fast algorithms for common transforms, where we stop early when RMSE $<$ 1e-4.

Transform	N = 8	16	32	64	128	256	512	1024
DFT	3.1e-06	4.6e-06	8.7e-06	1.0e-05	2.0e-05	3.8e-05	8.0e-05	5.7e-05
DCT	4.4e-06	1.1e-05	8.6e-06	1.2e-05	2.1e-05	1.9e-05	3.1e-05	7.3e-05
DST	1.1e-06	7.5e-06	4.6e-05	5.1e-05	3.0e-05	2.1e-05	3.6e-05	4.6e-05
Convolution	4.0e-06	2.5e-05	6.4e-05	7.6e-05	5.9e-05	7.8e-05	6.3e-05	1.9e-02
Hadamard	8.8e-07	7.8e-06	1.3e-05	3.9e-05	3.5e-05	4.5e-05	6.1e-05	3.6e-05
Hartley	3.4e-06	9.0e-06	1.1e-05	1.3e-05	3.6e-05	4.3e-05	4.5e-05	3.6e-05
Legendre	3.4e-02	2.9e-02	2.4e-02	1.4e-02	7.9e-03	4.5e-03	2.6e-03	1.6e-03
Randn	1.4e-01	1.6e-01	1.4e-01	1.1e-01	8.4e-02	6.1e-02	4.4e-02	3.1e-02

the size of the layer, since each $B^{(n)}$ is a product of $\log_2 n$ matrices. It thus satisfies the condition of the theorem, and so the VC dimension is bounded to be almost linear in the number of parameters:

$$\text{VCdim}(\text{sign}\,\mathcal{F}) = O(LW \log W).$$

□

B.2. Proposition 1

Proof. 1. The inclusion of the DFT in $(\text{BP})^1$ is shown in the Case study in Section 3.1. The inverse Fourier Transform has the same structure except the twiddle factors of the form ω_N^{-ij} are replaced with ω_N^{ij} and all entries of the first butterfly factor are scaled by $\frac{1}{N}$.

2. The inclusion of the Hadamard Transform in $(\text{BP})^1$ is shown in Section A.3.

3. The inclusion of the DCT in $(\text{BP})^2$ is shown in Section A.1.

4. The inclusion of the DST in $(\text{BP})^2$ is shown in Section A.2.

5. The inclusion of the convolution in $(\text{BP})^2$ is shown in Section A.4.

6. The inclusion of all $N \times N$ matrices in $(\text{BP})_2^{4N+10}$ follows from the fact that every $N \times N$ matrix may be expressed by a product of at most $2N + 5$ Toeplitz matrices (Ye & Lim, 2016). From Section A.5, we may conclude that all Toeplitz matrices are in $(\text{BP})_2^2$. Therefore, by appending the BP modules from each Toeplitz matrix, we see that a total of $4N + 10$ BP modules are needed. By left multiplying each butterfly factor by the $2N \times 2N$ diagonal matrix with 1s in the upper half and 0s in the lower half, we ensure that the upper left $N \times N$ entries of the final product are exactly the product of the upper left $N \times N$ entries of each BP module, as required. This diagonal matrix may be absorbed into the adjacent butterfly factor. Hence, the factorization is in $(\text{BP})_2^{4N+10}$.

□

C. Experimental Details and Results

C.1. Recovering Fast Transforms

In Section 4.1, given a matrix representation of a transform, we use the BP or BPBP parameter to recover a fast algorithm to the transform. We report in Table 4 the root mean square error (RMSE) $\sqrt{\frac{1}{N^2}\left\|T_N - B^{(N)}P^{(N)}\right\|}$ for different transforms and for different values of N.

We use Hyperband (Li et al., 2017) to tune the hyperparameters, which include the learning rate (from 0.0001 to 0.5), initialization, and whether to share the logits in the permutation block $P^{(N)}$.

C.2. Fully connected network

The model is a network with a single hidden layer of dimensions $N \times N$, where N is the input dimension, followed by a fully-connected softmax layer. We build on top of the framework of Thomas et al. (2018)[5], replacing the unconstrained

[5] Available at https://github.com/HazyResearch/structured-nets

or structured matrix with our PyTorch BPBP implementation. The CIFAR-10 dataset is a grayscale version of input size 1024 since the single hidden layer architecture receives a single channel as input. With the exception of learning rate, hyperparameters such as batch size 50, validation set comprising 15% of training data, and fixed momentum at 0.9 are fixed as reported in Appendix F.1 of their paper. For the BP methods, the learning rate was tested for the values $\{0.005, 0.01, 0.02, 0.05, 0.1, 0.2\}$; parameters outside this range were found to be ineffective. For each method, Table 1 reports the test accuracy of the model with the highest validation accuracy.

C.3. Resnet

We build on top of the standard ResNet18 model from PyTorch.[6] The model is modified for CIFAR-10 by reducing the kernel size and stride for the initial convolution to 3 and 1 respectively, and removing the first max pool layer. Weight decay of $\lambda = 0.0002$ was used. The learning rate was initialized in $\{0.1, 0.2\}$, and decayed by $\{0.1, 0.2\}$ every 25 epochs for 100 epochs total. For each method, Table 2 reports the mean and standard deviation of the test accuracies for the hyperparameters with the highest average validation accuracy.

C.4. Speed Comparison

In Section 4.3, we benchmark the speed of training and inference of butterfly factorizations.

For training, we compare our CUDA implementation of the fast algorithm for butterfly matrices with dense matrix-matrix multiply (GEMM from cuBLAS) and FFT (from cuFFT). The batch size is 256, and we measure the total time of the forward and backward pass. The experiment is run on a Tesla P100 GPU with 16GB of memory.

For inference, we compare our simple Python implementation of the fast algorithm for the BP parameterization, against dense matrix-vector multiplication (GEMV), FFT, DCT, and DST. Our BP parameterization here refers to the product of a butterfly matrix $B^{(N)}$ and a fixed permutation $P^{(N)}$ (say, learned from data). We use the standard dense matrix-vector multiplication implementation in Numpy (BLAS binding), the FFT implementation from Numpy and the DCT and DST implementation from Scipy (FFTPACK binding). We compare their speed in single-threaded mode, running on a server Intel Xeon CPU E5-2690 v4 at 2.60GHz.

Results are shown in Figure 4.

D. BP Hierarchy

In Definition 1, we defined the notion of a BP hierarchy, which we believes captures a natural class of matrices. To this point, we offer the following observations, the latter left as a conjecture, about the expressiveness of this hierarchy, supplementing the inclusion results of Proposition 1.

Proposition 3. *For every fixed $c \geq 1$, there is a sufficiently large N such that there is an $N \times N$ matrix M_N that is in (BP)$^{c+1}$ but not in (BP)^c.*

Proof. Given c, fix N such that N is even and such that $c < \frac{N}{8 \log_2 N}$. For sake of contradiction, assume that every $N \times N$ matrix in (BP)$^{c+1}$ is also in (BP)^c. Let A be an arbitrary $\frac{N}{2} \times \frac{N}{2}$ matrix. Then, from Proposition 1, A is in (BP)$_2^{2N+10}$. Therefore, from Definition 1, there is some $N \times N$ matrix $M \in$ (BP)$^{2N+10}$ such that the upper-left $\frac{N}{2} \times \frac{N}{2}$ entries are A. From our assumption, we can replace the first $c+1$ BP factors in M with c (possibly different) BP factors. We can repeat this process until we are left with c (BP) factors, so M in (BP)^c. This representation for M has $c \cdot 2N \log_2 N$ parameters, which must be less than $\frac{N}{8 \log_2 N} \cdot 2N \log_2 N = \frac{N^2}{4}$ based on how we fixed N above. However, A (and therefore M) has $\frac{N^2}{4}$ arbitrary entries, contradicting that it can be represented with fewer than $\frac{N^2}{4}$ parameters. Hence, there must be some $N \times N$ matrix in (BP)$^{c+1}$ that is not in (BP)^c. □

Conjecture 1. *Let M be an $N \times N$ matrix such that for any $x \in \mathcal{F}^N$, Mx can be computed with an arithmetic circuit of size $N \operatorname{poly}\log(N)$ and depth $\operatorname{poly}\log(N)$. Then, M is in (BP)$_{O(1)}^{\operatorname{poly}\log N}$.*

We believe that we can prove an approximation of the above using known approximations of the Jacobi transform by the DCT (up to some scaling) (Szegö, 1967). It is known that such transforms have an arithmetic circuit of the kind mentioned

[6] Available at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

in the conjecture above (Driscoll et al., 1997).