Butterfly Transform: An Efficient FFT Based Neural Architecture Design

Keivan Alizadeh vahid, Anish Prabhu, Ali Farhadi, Mohammad Rastegari
University of Washington,
`keivan@cs.washington.edu`

Abstract

In this paper, we show that extending the butterfly operations from the FFT algorithm to a general Butterfly Transform (BFT) can be beneficial in building an efficient block structure for CNN designs. Pointwise convolutions, which we refer to as channel fusions, are the main computational bottleneck in the state-of-the-art efficient CNNs (e.g. MobileNets [15, 38, 14]). We introduce a set of criterion for channel fusion, and prove that BFT yields an asymptotically optimal FLOP count with respect to these criteria. By replacing pointwise convolutions with BFT, we reduce the computational complexity of these layers from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ with respect to the number of channels. Our experimental evaluations show that our method results in significant accuracy gains across a wide range of network architectures, especially at low FLOP ranges. For example, BFT results in up to a 6.75% absolute Top-1 improvement for MobileNetV1[15], 4.4% for ShuffleNet V2[28] and 5.4% for MobileNetV3[14] on ImageNet under a similar number of FLOPS. Notably, ShuffleNet-V2+BFT outperforms state-of-the-art architecture search methods MNasNet[43], FBNet [46] and MobilenetV3[14] in the low FLOP regime.

1. Introduction

Devising Convolutional Neural Networks (CNN) that can run efficiently on resource-constrained edge devices has become an important research area. There is a continued push to put increasingly more capabilities on-device for personal privacy, latency, and scale-ability of solutions. On these constrained devices, there is often extremely high demand for a limited amount of resources, including computation and memory, as well as power constraints to increase battery life. Along with this trend, there has also been greater ubiquity of custom chip-sets, Field Programmable Gate Arrays (FPGAs), and low-end processors that can be used to run CNNs, rather than traditional GPUs.

A common design choice is to reduce the FLOPs and parameters of a network by factorizing convolutional layers [15, 38, 28, 50] into a depth-wise separable convolution
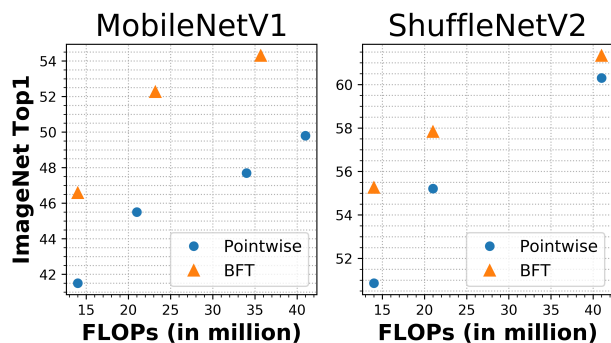

Figure 1: Replacing pointwise convolutions with BFT in state-of-the-art architectures results in significant accuracy gains in resource constrained settings.

that consists of two components: (1) *spatial fusion*, where each spatial channel is convolved independently by a depth-wise convolution, and (2) *channel fusion*, where all the spatial channels are linearly combined by 1×1 convolutions, known as pointwise convolutions. Inspecting the computational profile of these networks at inference time reveals that the computational burden of the spatial fusion is relatively negligible compared to that of the channel fusion[15]. In this paper we focus on designing an efficient replacement for these pointwise convolutions.

We propose a set of principles to design a replacement for pointwise convolutions motivated by both efficiency and accuracy. The proposed principles are as follows: (1) *full connectivity from every input to all outputs*: to allow outputs to use all available information, (2) *large information bottleneck*: to increase representational power throughout the network, (3) *low operation count*: to reduce the computational cost, (4) *operation symmetry*: to allow operations to be stacked into dense matrix multiplications. In Section 3, we formally define these principles, and mathematically prove a lower-bound of $O(n \log n)$ operations to satisfy these principles. We propose a novel, lightweight convolutional building block based on the Butterfly Transform (BFT). We prove that BFT yields an asymptotically optimal FLOP count under

these principles.

We show that BFT can be used as a drop-in replacement for pointwise convolutions in several state-of-the-art efficient CNNs. This significantly reduces the computational bottleneck for these networks. For example, replacing pointwise convolutions with BFT decreases the computational bottleneck of MobileNetV1 from 95% to 60%, as shown in Figure 3. We empirically demonstrate that using BFT leads to significant increases in accuracy in constrained settings, including up to a 6.75% absolute Top-1 gain for MobileNetV1, 4.4% for ShuffleNet V2 and 5.4% for MobileNetV3 on the ImageNet[7] dataset. There have been several efforts on using butterfly operations in neural networks [20, 6, 33] but, to the best of our knowledge, our method outperforms all other structured matrix methods (Table 2b) for replacing pointwise convolutions as well as state-of-the-art Neural Architecture Search (Table 2a) by a large margin at low FLOP ranges.

2. Related Work

Deep neural networks suffer from intensive computations. Several approaches have been proposed to address efficient training and inference in deep neural networks.

Efficient CNN Architecture Designs: Recent successes in visual recognition tasks, including object classification, detection, and segmentation, can be attributed to exploration of different CNN designs [23, 39, 13, 21, 42, 17]. To make these network designs more efficient, some methods have factorized convolutions into different steps, enforcing distinct focuses on spatial and channel fusion [15, 38]. Further, other approaches extended the factorization schema with sparse structure either in channel fusion [28, 50] or spatial fusion [30]. [16] forced more connections between the layers of the network but reduced the computation by designing smaller layers. Our method follows the same direction of designing a sparse structure on channel fusion that enables lower computation with a minimal loss in accuracy.

Structured Matrices: There have been many methods which attempt to reduce the computation in CNNs, [44, 24, 8, 19] by exploiting the fact that CNNs are often extremely over-parameterized. These models learn a CNN or fully connected layer by enforcing a linear transformation structure during the training process which has less parameters and computation than the original linear transform. Different kinds of structured matrices have been studied for compressing deep neural networks, including circulant matrices[9], toeplitz-like matrices[40], low rank matrices[37], and fourier-related matrices[32]. These structured matrices have been used for approximating kernels or replacing fully connected layers. UGConv [51] has considered replacing one of the pointwise convolutions in the ShuffleNet structure with unitary group convolutions, while our Butterfly Transform is able to replace all of the pointwise convolutions. The butterfly structure has been studied for a long time in linear algebra [34, 26] and neural network models [31]. Recently, it has received more attention from researchers who have used it in RNNs [20], kernel approximation[33, 29, 4] and fully connected layers[6]. We have generalized butterfly structures to replace pointwise convolutions, and have significantly outperformed all known structured matrix methods for this task, as shown in Table 2b.

Network pruning: This line of work focuses on reducing the substantial redundant parameters in CNNs by pruning out either neurons or weights [11, 12, 45, 2]. Our method is different from these type methods in the way that we enforce a predefined sparse channel structure to begin with and we do not change the structure of the network during the training.

Quantization: Another approach to improve the efficiency of the deep networks is low-bit representation of network weights and neurons using quantization [41, 35, 47, 5, 52, 18, 1]. These approaches use fewer bits (instead of 32-bit high-precision floating points) to represent weights and neurons for the standard training procedure of a network. In the case of extremely low bitwidth (1-bit) [35] had to modify the training procedure to find the discrete binary values for the weights and the neurons in the network. Our method is orthogonal to this line of work and these method are complementary to our network.

Neural architecture search: Recently, neural search methods, including reinforcement learning and genetic algorithms, have been proposed to automatically construct network architectures [53, 48, 36, 54, 43, 27]. Recent search-based methods [43, 3, 46, 14] use Inverted Residual Blocks [38] as a basic search block for automatic network design. The main computational bottleneck in most of the search based method is in the channel fusion and our butterfly structure does not exist in any of the predefined blocks of these methods. Our efficient channel fusion can be augmented with these models to further improve the efficiency of these networks. Our experiments shows that our proposed butterfly structure outperforms recent architecture search based models on small network design.

3. Model

In this section, we outline the details of the proposed model. As discussed above, the main computational bottleneck in current efficient neural architecture design is in the channel fusion step, which is implemented with a pointwise convolution layer. The input to this layer is a tensor $\mathbf{X}$ of size $n_{\text{in}} \times h \times w$, where n is the number of channels
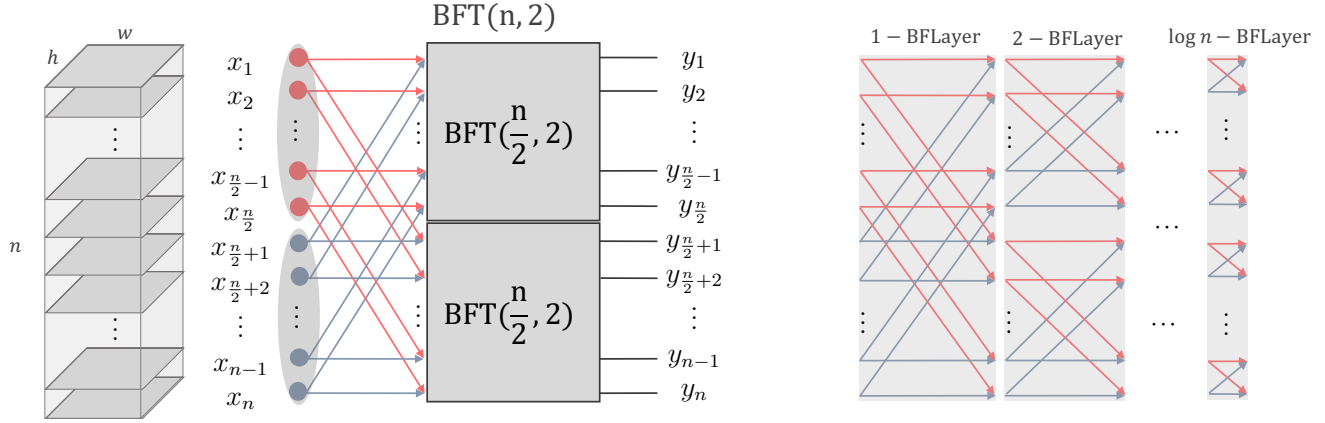
Figure 2: **BFT Architecture:** This figure illustrates the graph structure of the proposed Butterfly Transform. The left figure shows the recursive procedure of the BFT that is applied to an input tensor and the right figure shows the expanded version of the recursive procedure as $\log n$ Butterfly Layers in the network.

and w, h are the width and height respectively. The size of the weight tensor $\mathbf{W}$ is $n_{\text{out}} \times n_{\text{in}} \times 1 \times 1$ and the output tensor $\mathbf{Y}$ is $n_{\text{out}} \times h \times w$. For the sake of simplicity, we assume $n = n_{\text{in}} = n_{\text{out}}$. The complexity of a pointwise convolution layer is $\mathcal{O}(n^2wh)$, and this is mainly influenced by the number of channels n. We propose to use *Butterfly Transform* as a layer, which has $\mathcal{O}((n\log n)wh)$ complexity. This design is inspired by the Fast Fourier Transform (FFT) algorithm, which has been widely used in computational engines for a variety of applications and there exist many optimized hardware/software designs for the key operations of this algorithm, which are applicable to our method. In the following subsections we explain the problem formulation and the structure of our butterfly transform.

3.1. Pointwise Convolution as Matrix-Vector Products

A pointwise convolution can be defined as a function $\mathcal{P}$ as follows:

$$\mathbf{Y} = \mathcal{P}(\mathbf{X}; \mathbf{W}) \tag{1}$$

This can be written as a matrix product by reshaping the input tensor $\mathbf{X}$ to a 2-D matrix $\hat{\mathbf{X}}$ with size $n \times (hw)$ (each column vector in the $\hat{\mathbf{X}}$ corresponds to a spatial vector $\mathbf{X}[:, i, j]$) and reshaping the weight tensor to a 2-D matrix $\hat{\mathbf{W}}$ with size $n \times n$,

$$\hat{\mathbf{Y}} = \hat{\mathbf{W}}\hat{\mathbf{X}} \tag{2}$$

where $\hat{\mathbf{Y}}$ is the matrix representation of the output tensor $\mathbf{Y}$. This can be seen as a linear transformation of the vectors in the columns of $\hat{\mathbf{X}}$ using $\hat{\mathbf{W}}$ as a transformation matrix. The linear transformation is a matrix-vector product and its complexity is $\mathcal{O}(n^2)$. By enforcing structure on this transformation matrix, one can reduce the complexity of the transformation. However, to be effective as a channel fusion

transform, it is critical that this transformation respects the desirable characteristics detailed below.

Fusion network design principles: 1) *full connectivity from every input to all outputs*: This condition allows every single output to have access to all available information in the inputs. 2) *large information bottleneck*: The bottleneck size is defined as the minimum number of nodes in the network that if removed, the information flow from input channels to output channels would be completely cut off (i.e. there would be no path from any input channel to any output channel). The representational power of the network is bound by the bottleneck size. To ensure that information is not lost while passed through the channel fusion, we set the minimum bottleneck size to n. 3) *low operation count*: The fewer operations, or equivalently edges in the graph, that there are, the less computation the fusion will take. Therefore we want to reduce the number of edges. 4) *operation symmetry*: By enforcing that there is an equal out-degree in each layer, the operations can be stacked into dense matrix multiplications, which is in practice much faster for inference than sparse computation.

Claim: A multi-layer network with these properties has at least $\mathcal{O}(n \log n)$ edges.

Proof: Suppose there exist n_i nodes in i^{th} layer. Removing all the nodes in one layer will disconnect inputs from outputs. Since the maximum possible bottleneck size is n, therefore $n_i \geq n$. Now suppose that out degree of each node at layer i is d_i. Number of nodes in layer i, which are reachable from an input channel is $\prod_{j=0}^{i-1} d_j$. Because of the every-to-all connectivity, all of the n nodes in the output layer are reachable. Therefore $\prod_{j=0}^{m-1} d_j \geq n$. This implies that $\sum_{j=0}^{m-1} \log_2(d_j) \geq \log_2(n)$. The total number of edges will be:

$$\sum_{j=0}^{m-1} n_j d_j \ \geq \ n \sum_{j=0}^{m-1} d_j \ \geq \ n \sum_{j=0}^{m-1} \log_2(d_j) \ \geq \ n \log_2 n \blacksquare$$

In the following section we present a network structure that satisfies all the design principles for fusion network.

3.2. Butterfly Transform (BFT)

As mentioned above we can reduce the complexity of a matrix-vector product by enforcing structure on the matrix. There are several ways to enforce structure on the matrix. Here we first explain how the channel fusion is done through BFT and then show a family of the structured matrix equivalent to this fusion leads to a $\mathcal{O}(n \log n)$ complexity of operations and parameters while maintaining accuracy.

Channel Fusion through BFT: We want to fuse information among all channels. We do it in sequential layers. In the first layer we partition channels to k parts with size $\frac{n}{k}$ each, $\mathbf{x}_1, .., \mathbf{x}_k$. We also partition output channels of this first layer to k parts with $\frac{n}{k}$ size each, $\mathbf{y}_1, .., \mathbf{y}_k$. We connect elements of $\mathbf{x}_i$ to $\mathbf{y}_j$ with $\frac{n}{k}$ parallel edges $\mathbf{D}_{ij}$. After combining information this way, each $\mathbf{y}_i$ contains the information from all channels, then we recursively fuse information of each $\mathbf{y}_i$ in the next layers.

Butterfly Matrix: In terms of matrices $\mathbf{B}^{(n,k)}$ is a butterfly matrix of order n and base k where $\mathbf{B}^{(n,k)} \in \mathbb{R}^{n \times n}$ is equivalent to fusion process described earlier.

$$\mathbf{B}^{(n,k)} = \begin{pmatrix} \mathbf{M}_1^{(\frac{n}{k},k)}\mathbf{D}_{11} & \dots & \mathbf{M}_1^{(\frac{n}{k},k)}\mathbf{D}_{1k} \\ \vdots & \ddots & \vdots \\ \mathbf{M}_k^{(\frac{n}{k},k)}\mathbf{D}_{k1} & \dots & \mathbf{M}_k^{(\frac{n}{k},k)}\mathbf{D}_{kk} \end{pmatrix} \quad (3)$$

Where $\mathbf{M}_i^{(\frac{n}{k},k)}$ is a butterfly matrices of order $\frac{n}{k}$ and base k and $\mathbf{D}_{ij}$ is an arbitrary diagonal $\frac{n}{k} \times \frac{n}{k}$ matrix. The matrix-vector product between a butterfly matrix $\mathbf{B}^{(n,k)}$ and a vector $\mathbf{x} \in \mathbb{R}^n$ is :

$$\mathbf{B}^{(n,k)}\mathbf{x} = \begin{pmatrix} \mathbf{M}_1^{(\frac{n}{k},k)}\mathbf{D}_{11} & \dots & \mathbf{M}_1^{(\frac{n}{k},k)}\mathbf{D}_{1k} \\ \vdots & \ddots & \vdots \\ \mathbf{M}_k^{(\frac{n}{k},k)}\mathbf{D}_{k1} & \dots & \mathbf{M}_k^{(\frac{n}{k},k)}\mathbf{D}_{kk} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_k \end{pmatrix}$$
$$(4)$$

where $\mathbf{x}_i \in \mathbb{R}^{\frac{n}{k}}$ is a subsection of $\mathbf{x}$ that is achieved by breaking $\mathbf{x}$ into k equal sized vector. Therefore, the product can be simplified by factoring out $\mathbf{M}$ as follow:

$$\mathbf{B}^{(n,k)}\mathbf{x} = \begin{pmatrix} \mathbf{M}_1^{(\frac{n}{k},k)} \sum_{j=1}^{k} \mathbf{D}_{1j}\mathbf{x}_j \\ \vdots \\ \mathbf{M}_i^{(\frac{n}{k},k)} \sum_{j=1}^{k} \mathbf{D}_{ij}\mathbf{x}_j \\ \vdots \\ \mathbf{M}_k^{(\frac{n}{k},k)} \sum_{j=1}^{k} \mathbf{D}_{kj}\mathbf{x}_j \end{pmatrix} = \begin{pmatrix} \mathbf{M}_1^{(\frac{n}{k},k)}\mathbf{y}_1 \\ \vdots \\ \mathbf{M}_i^{(\frac{n}{k},k)}\mathbf{y}_i \\ \vdots \\ \mathbf{M}_k^{(\frac{n}{k},k)}\mathbf{y}_k \end{pmatrix}$$
$$(5)$$

where $\mathbf{y}_i = \sum_{j=1}^{k} \mathbf{D}_{ij}\mathbf{x}_j$. Note that $\mathbf{M}_i^{(\frac{n}{k},k)}\mathbf{y}_i$ is a smaller product between a butterfly matrix of order $\frac{n}{k}$ and a vector of size $\frac{n}{k}$ therefore, we can use divide-and-conquer to recursively calculate the product $\mathbf{B}^{(n,k)}\mathbf{x}$. If we consider $T(n,k)$ as the computational complexity of the product between a (n,k) butterfly matrix and an n-D vector. From equation 5, the product can be calculated by k products of butterfly matrices of order $\frac{n}{k}$ which its complexity is $kT(n/k,k)$. The complexity of calculating $\mathbf{y}_i$ for all $i \in \{1, \dots, k\}$ is $\mathcal{O}(kn)$ therefore:

$$T(n,k) = kT(n/k,k) + \mathcal{O}(kn) \quad (6)$$

$$T(n,k) = \mathcal{O}(k(n \log_k n)) \quad (7)$$

With a smaller choice of $k(2 \leq k \leq n)$ we can achieve a lower complexity. Algorithm 1 illustrates the recursive procedure of a butterfly transform when $k = 2$.

Algorithm 1: Recursive Butterfly Transform

---
1 **Function** `ButterflyTransform`(*W, X, n*):
    /* algorithm as a recursive function */
    Data: W
    Weights containing $2n \log(n)$ numbers
    Data: X
    An input containing n numbers
2   **if** *n == 1* **then**
3   **return** *[X]* ;
4   Make $D_{11}, D_{12}, D_{21}, D_{22}$ using first $2n$ numbers of W;
5   Split rest $2n(\log(n) - 1)$ numbers to two sequences W_1, W_2 with length $n(\log(n) - 1)$ each.;
6   Split X to X_1, X_2;
7     $y_1 \longleftarrow D_{11}X_1 + D_{12}X_2$;
8     $y_2 \longleftarrow D_{21}X_1 + D_{22}X_2$;
9     $My_1 \longleftarrow$ `ButterflyTransform`$(W_1, y_1, n-1)$;
10     $My_2 \longleftarrow$ `ButterflyTransform`$(W_2, y_2, n-1)$;
11   **return** *Concat*(My_1, My_2);

3.3. Butterfly Neural Network

The procedure explained in Algorithm 1 can be represented by a butterfly graph similar to the FFT's graph. The butterfly network structure has been used for function representation [25] and fast factorization for approximating linear transformation [6]. We adopt this graph as an architecture design for the layers of a neural network. Figure 2 illustrates the architecture of a butterfly network of base $k = 2$ applied on an input tensor of size $n \times h \times w$. The left figure shows how the recursive structure of the BFT as a network. The right figure shows the constructed multi-layer network which has $\log n$ Butterfly Layers (BFLayer). Note that the complexity of each Butterfly Layer is $\mathcal{O}(n)$ ($2n$ operations), therefore, the total complexity of the BFT architecture will be $\mathcal{O}(n \log n)$.

Each Butterfly layer can be augmented by batch norm and nonlinearity functions (*e.g.* ReLU, Sigmoid). In Section 4.2 we study the effect of using different choices of these functions. We found that both batch norm and nonlinear functions (ReLU and Sigmoid) are not effective within BFLayers. Batch norm is not effective
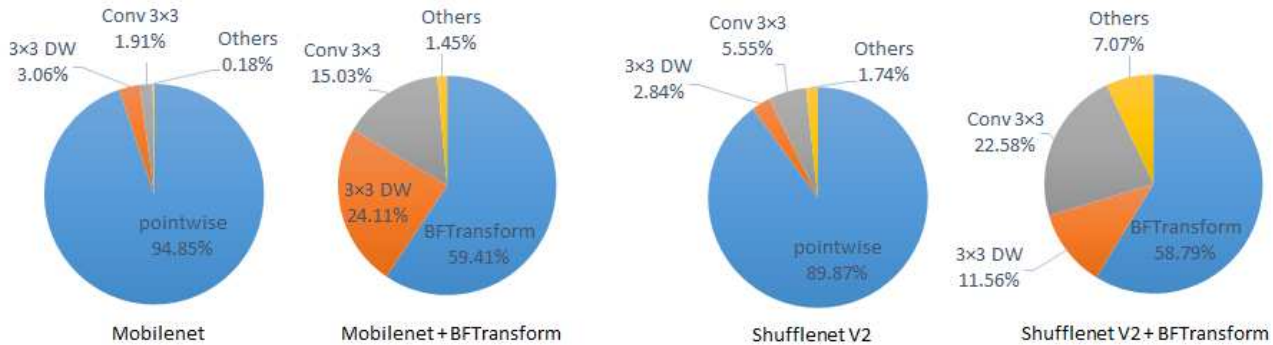
Figure 3: **Distribution of FLOPs:** This figure shows that replacing the pointwise convolution with BFT reduces the size of the computational bottleneck.

mainly because its complexity is the same as the BFLayer $\mathcal{O}(n)$, therefore, it doubles the computation of the entire transform. We use batch norm only at the end of the transform. The non-linear activation $ReLU$ and $Sigmoid$ zero out almost half of the values in each BFLayer, thus multiplication of these values throughout the forward propagation destroys all the information. The BFLayers can be internally connected with residual connections in different ways. In our experiments, we found that the best residual connections are the one that connect the input of the first BFLayer to the output of the last BFLayer. The base of the BFT affects the shape and the number of FLOPs. We have empirically found that base $k = 4$ achieves the highest accuracy while having the same number FLOPs as the base $k = 2$ as shown in Figure 5c.

Butterfly network satisfies all the fusion network design principles. There exist exactly one path between every input channel to all the output channels, the degree of each node in the graph is exactly k, the bottleneck size is n, and the number of edges are $\mathcal{O}(n \log n)$.

We use the BFT architecture as a replacement of the pointwise convolution layer (1×1 convs) in different CNN architectures including MobileNetV1[15], ShuffleNetV2[28] and MobileNetV3[14]. Our experimental results shows that under the same number of FLOPs, the efficiency gain by BFT is more effective in terms of accuracy compared to the original model with smaller channel rate. We show consistent accuracy improvement across several architecture settings.

Fusing channels using BFT, instead of pointwise convolution reduces the size of the computational bottleneck by a large-margin. Figure 3 illustrate the percentage of the number of operations by each block type throughout a forward pass in the network. Note that when BFT is applied, the percentage of the depth-wise convolutions increases by $8\times$.

4. Experiments

In this section, we demonstrate the performance of the proposed BFT on large-scale image classification tasks. To showcase the strength of our method in designing very small networks, we compare performance of Butterfly Transform with pointwise convolutions in three state-of-the-art efficient architectures: (1)

MobileNetV1, (2) ShuffleNetV2, and (3) MobileNetV3. We compare our results with other type of structured matrices that have $\mathcal{O}(n \log n)$ computation (e.g. low-rank transform and circulant transform). We also show that our method outperforms state-of-the-art architecture search methods at low FLOP ranges.

4.1. Image Classification

4.1.1 Implementation and Dataset Details:

Following standard practice, we evaluate the performance of Butterfly Transforms on the ImageNet dataset, at different levels of complexity, ranging from 14 MFLOPS to 150 MFLOPs. ImageNet classification dataset contains 1.2M training samples and 50K validation samples, uniformly distributed across 1000 classes.

For each architecture, we substitute pointwise convolutions with Butterfly Transforms. To keep the FLOP count similar between BFT and pointwise convolutions, we adjust the channel numbers in the base architectures (MobileNetV1, ShuffleNetV2, and MobileNetV3). For all architectures, we optimize our network by minimizing cross-entropy loss using SGD. Specific learning rate regimes are used for each architecture which can be found in the Appendix. Since BFT is sensitive to weight decay, we found that using little or no weight decay provides much better accuracy. We experimentally found (Figure 5c) that butterfly base $k = 4$ performs the best. We also used a custom weight initialization for the internal weights of the Butterfly Transform which we outline below. More information and intuition on these hyper-parameters can be found in our ablation studies (Section 4.2).

Weight initialization: Proper weight initialization is critical for convergence of neural networks, and if done improperly can lead to instability in training, and poor performance. This is especially true for Butterfly Transforms due to the amplifying effect of the multiplications within the layer, which can create extremely large or small values. A common technique for initializing pointwise convolutions is to initialize weights uniformly from the range $(-x, x)$ where $x = \sqrt{\frac{6}{n_{in}+n_{out}}}$, which is referred to as Xavier initialization [10]. We cannot simply apply this initialization to butterfly layers, since we are changing the internal structure.

(a)

Flops	ShuffleNetV2	ShuffleNetV2+**BFT**	Gain
14 M	50.86 (14 M)*	55.26 (14 M)	**4.40**
21 M	55.21 (21 M)*	57.83 (21 M)	**2.62**
40 M	59.70(41 M)*	61.33 (41 M)	**1.63**
	60.30 (41 M)		**1.03**

(b)

Flops	MobileNetV3	MobileNetV3+**BFT**	Gain
10-15 M	49.8 (13 M)	55.21 (15 M)	**5.41**

(c)

Flops	MobileNet	MobileNet+**BFT**	Gain
14 M	41.50 (14 M)	46.58 (14 M)	**5.08**
20 M	45.50 (21 M)	52.26 (23 M)	**6.76**
40 M	47.70 (34 M)	54.30 (35 M)	**6.60**
	50.60 (41 M)		**3.70**
50 M	56.30 (49 M)	57.56 (51 M)	**1.26**
		58.35 (52 M)	**2.05**
110 M	61.70 (110 M)	63.03 (112 M)	**1.33**
150 M	63.30 (150 M)	64.32 (150 M)	**1.02**

Table 1: These tables compare the accuracy of ShuffleNetV2, MobileNetV1 and MobileNetV3 when using standard pointwise convolution vs using BFTs

We denote each entry $B_{u,v}^{(n,k)}$ as the multiplication of all the edges in path from node u to v. We propose initializing the weights of the butterfly layers from a range $(-y, y)$, such that the multiplication of all edges along paths, or equivalently values in $B^{(n,k)}$, are initialized close to the range $(-x, x)$. To do this, we solve for a y which makes the expectation of the absolute value of elements of $B^{(n,k)}$ equal to the expectation of the absolute value of the weights with standard Xavier initialization, which is $x/2$. Let $e_1, .., e_{log(n)}$ be edges on the path p from input node u to output node v. We have the following:

$$E[|B_{u,v}^{(n,k)}|] = E[|\prod_{i=1}^{log(n)} e_i|] = \frac{x}{2} \qquad (8)$$

We initialize each e_i in range $(-y, y)$ where

$$(\frac{y}{2})^{log(n)} = \frac{x}{2} \implies y = x^{\frac{1}{log(n)}} * 2^{\frac{log(n)-1}{log(n)}}. \qquad (9)$$

4.1.2 MobileNetV1 + BFT

To add BFT to MobileNeV1, for all MobileNetV1 blocks, which consist of a depthwise layer followed by a pointwise layer, we replace the pointwise convolution with our Butterfly Transform, as shown in Figure 4. We would like to emphasize that this means we replace *all* pointwise convolution in MobileNetV1, with BFT. In Table 1, we show that we outperform a spectrum of MobileNetV1s from about 14M to 150M FLOPs with a spectrum of MobileNetV1s+BFT within the same FLOP range. Our experiments with MobileNetV1+BFT include all combinations of width-multiplier 1.00 and 2.00, as well as input resolutions 128, 160, 192, and 224. We also add a width-multiplier 1.00 with input resolution 96 to cover the low FLOP range (14M). A full table of results can be found in the Appendix.

In Table 1c we showcase that using BFT outperforms traditional MobileNets across the entire spectrum, but is especially effective in the low FLOP range. For example using BFT results in an increase of 6.75% in top-1 accuracy at 23 MFLOPs. Note that MobileNetV1 + BFT at 23 MFLOPs has much higher accuracy than MobileNetV1
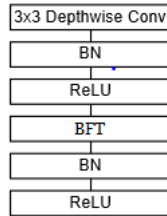
```
3x3 Depthwise Conv
BN
ReLU
BFT
BN
ReLU
```

Figure 4: **MobileNetV1+BFT Block**

at 41 MFLOPs, which means it can get higher accuracy with almost half of the FLOPs. This was achieved without changing the architecture at all, other than simply replacing pointwise convolutions, which means there are likely further gains by designing architectures with BFT in mind.

4.1.3 ShuffleNetV2 + BFT

We modify the ShuffleNet block to add BFT to ShuffleNetv2. In Table 1a we show results for ShuffleNetV2+BFT, versus the original ShuffleNetV2. We have interpolated the number of output channels to build ShuffleNetV2-1.25+BFT, to be comparable in FLOPs with a ShuffleNetV2-0.5. We have compared these two methods for different input resolutions (128, 160, 224) which results in FLOPs ranging from 14M to 41M. ShuffleNetV2-1.25+BFT achieves about 1.6% better accuracy than our implementation of ShuffleNetV2-0.5 which uses pointwise convolutions. It achieves 1% better accuracy than the reported numbers for ShuffleNetV2 [28] at 41 MFLOPs.

4.1.4 MobileNetV3 + BFT

We follow a procedure which is very similar to that of MobileNetV1+BFT, and simply replace all pointwise convolutions with Butterfly Transforms. We trained a MobileNetV3+BFT Small with a network-width of 0.5 and an input resolution 224, which achieves 55.21% Top-1 accuracy. This model outperforms MobileNetV3 Small network-width of 0.35 and input resolution 224 at a similar FLOP range by about 5.4% Top-1, as shown in 1b. Due to resource constraints, we only trained one variant of MobileNetV3+BFT.

4.1.5 Comparison with Neural Architecture Search

Including BFT in ShuffleNetV2 allows us to achieve higher accuracy than state-of-the-art architecture search methods, MNasNet[43], FBNet [46], and MobileNetV3 [14] on an extremely low resource setting ($\sim$ 14M FLOPs). These architecture search methods search a space of predefined building blocks, where the most efficient block for channel fusion is the pointwise convolution. In Table 2a, we show that by simply replacing pointwise convolutions in ShuffleNetv2, we are able to outperform state-of-the-art architecture search methods in terms of Top-1 accuracy on ImageNet. We hope that this leads to future work where BFT is in-

(a) **BFT vs. Architecture Search**			(b) **BFT vs. Other Structured Matrix Approaches**	

Model	Accuracy
ShuffleNetV2+**BFT** (14 M)	**55.26**
MobileNetV3Small-224-0.5+**BFT** (15 M)	**55.21**
FBNet-96-0.35-1 (12.9 M)	50.2
FBNet-96-0.35-2 (13.7 M)	51.9
MNasNet (12.7 M)	49.3
MobileNetV3Small-224-0.35 (13 M)	49.8
MobileNetV3Small-128-1.0 (12 M)	51.7

Model	Accuracy
MobilenetV1+**BFT** (35 M)	**54.3**
MobilenetV1 (42 M)	50.6
MobilenetV1+Circulant* (42 M)	35.68
MobilenetV1+low-rank* (37 M)	43.78
MobilenetV1+BPBP (35 M)	49.65
MobilenetV1+Toeplitz* (37 M)	40.09
MobilenetV1+FastFood* (37 M)	39.22

Table 2: These tables compare BFT with other efficient network design approaches. In Table (a), we show that ShuffleNetV2 + BFT outperforms state-of-the-art neural architecture search methods (MNasNet [43], FBNet[46], MobilenetV3[14]). In Table (b), we show that BFT achieves significantly higher accuracy than other structured matrix approaches which can be used for channel fusion. The * denotes that this is our implementation.

cluded as one of the building blocks in architecture searches, since it provides an extremely low FLOP method for channel fusion.

4.1.6 Comparison with Structured Matrices

To further illustrate the benefits of Butterfly Transforms, we compare them with other structured matrix methods which can be used to reduce the computational complexity of pointwise convolutions. In Table 2b we show that BFT significantly outperforms all these other methods at a similar FLOP range. For comparability, we have extended all the other methods to be used as replacements for pointwise convolutions, if necessary. We then replaced all pointwise convolutions in MobileNetV1 for each of the methods and report Top-1 validation accuracy on ImageNet. Here we summarize these other methods:

Circulant block: In this block, the matrix that represents the pointwise convolution is a circulant matrix. In a circulant matrix rows are cyclically shifted versions of one another [9]. The product of this circulant matrix by a column can be efficiently computed in $\mathcal{O}(n \log(n))$ using the Fast Fourier Transform (FFT).

Low-rank matrix: In this block, the matrix that represents the pointwise convolution is the product of two $log(n)$ rank matrices ($W = UV^T$). Therefore the pointwise convolution can be performed by two consequent small matrix product and the total complexity is $\mathcal{O}(n \log n)$.

Toeplitz Like: Toeplitz like matrices have been introduced in [40]. They have been proven to work well on kernel approximation. We have used displacement rank $r = 1$ in our experiments.

Fastfood: This block has been introduce in [22] and used in Deep Fried ConvNets[49]. In Deep Fried Nets they replace fully connected layers with FastFood. By unifying batch, height and width dimension, we can use a fully connected layer as a pointwise convolution.

BPBP: This method uses the butterfly network structure for fast factorization for approximating linear transformation, such as Discrete Fourier Transform (DFT) and the Hadamard transform[6]. We extend BPBP to work with pointwise convolutions by using the trick explained in the Fastfood section above, and performed experiments on ImageNet.

4.2. Ablation Study

Now, we study different elements of our BFT model. As mentioned earlier, residual connections and non-linear activations can

be augmented within our BFLayers. Here we show the performance of these elements in isolation on CIFAR-10 dataset using MobileNetv1 as the base network. The only exception is the Butterfly Base experiment which was performed on ImageNet.

Residual connections: The graphs that are obtained by replacing BFTransform with pointwise convolutions are very deep. Residual connections generally help when training deep networks. We experimented with three

Model	Accuracy
No residual	79.2
Every-other-Layer	81.12
First-to-Last	**81.75**

Table 3: **Residual connections**

different ways of adding residual connections (1) *First-to-Last*, which connects the input of the first BFLayer to the output of last BFLayer, (2) *Every-other-Layer*, which connects every other BFLayer and (3) *No-residual*, where there is no residual connection. We found the First-to-last is the most effective type of residual connection as shown in Table 3.

With/Without Non-Linearity: As studied by [38] adding a non-linearity function like $ReLU$ or $Sigmoid$ to a narrow layer (with few channels) reduces the accuracy because it cuts off half of the values of an internal layer to zero. In BFT, the effect of an input channel i on an output channel o, is determined by the multiplication of all the edges on the path between i and o. Dropping any value along the path to zero will destroy all the information transferred between the two nodes. Dropping half of the values of each internal layer destroys almost all the information in the entire layer. Because of this, we don't use any activation in the internal Butterfly Layers. Figure 5b compares the the learning curves of BFT models with and without non-linear activation functions.

With/Without Weight-Decay: We found that BFT is very sensitive to the weight decay. This is because in BFT there is only one path from an input channel i to an output channel o. The effect of i on o is determined by the multiplication of all the intermediate edges along the path between i and o. Pushing all weight values toowards zero, will significantly reduce the effect of the i on o. Therefore, weight decay is very destructive in BFT. Figure 5a illustrates the learning curves with and without using weight decay on BFT.

Butterfly base: The parameter k in $B^{(n,k)}$ determines the structure of the Butterfly Transform and has a significant impact on the accuracy of the model. The internal structure of the BFT will

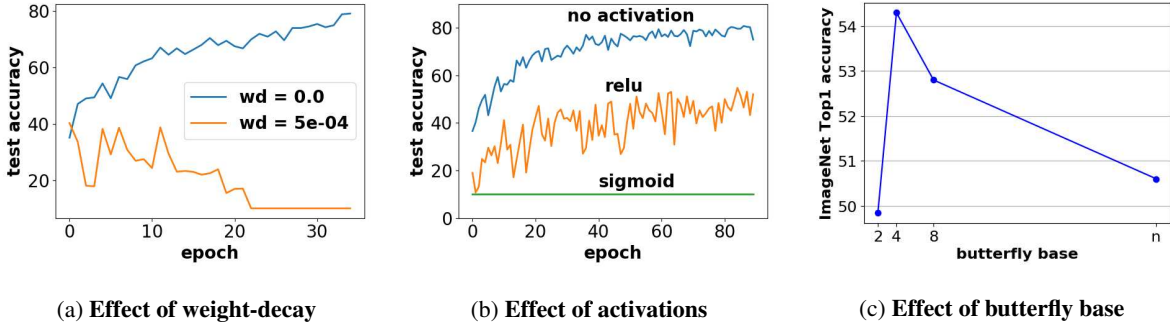| (a) **Effect of weight-decay** | (b) **Effect of activations** | (c) **Effect of butterfly base** |

Figure 5: Design choices for BFT: a) In BFT we should not enforce weight decay, because it significantly reduces the effect of input channels on output channels. b) Similarly, we should not apply the common non-linear activation functions. These functions zero out almost half of the values in the intermediate BFLayers, which leads to a catastrophic drop in the information flow from input channels to the output channels. c) Butterfly base determines the structure of BFT. Under $40M$ FLOP budget base $k = 4$ works the best.

contain $\log_k(n)$ layers. Because of this, very small values of k lead to deeper internal structures, which can be more difficult to train. Larger values of k are shallower, but have more computation, since each node in layers inside the BFT has an out-degreee of k. With large values of k, this extra computation comes at the cost of more FLOPs.

We tested the values of $k = 2, 4, 8, n$ on MobileNetV1+BFT with an input resolution of 160x160 which results in $\sim 40M$ FLOPs. When $k = n$, this is equivalent to a standard pointwise convolution. For a fair comparison, we made sure to hold FLOPs consistent across all our experiments by varying the number of channels, and tested all models with the same hyper-parameters on ImageNet. Our results in Figure 5c show that $k = 4$ significantly outperforms all other values of k. Our intuition is that this setting allows the block to be trained easily, due to its shallowness, and that more computation than this is better spent elsewhere, such as in this case increasing the number of channels. It is a likely possibility that there is a more optimal value for k, which varies throughout the model, rather than being fixed. We have also only performed this ablation study on a relatively low FLOP range ($40M$), so it might be the case that larger architectures perform better with a different value of k. There is lots of room for future exploration in this design choice.

5. Drawbacks

A weakness of our model is that there is an increase in working memory when using BFT since we must add substantially more channels to maintain the same number of FLOPs as the original network. For example, a MobileNetV1-2.0+BFT has the same number of FLOPS as a MobileNetV1-0.5, which means it will use about four times as much working memory. Please note that the intermediate BFLayers can be computed in-place so they do not increase the amount of working memory needed. Due to using wider channels, GPU training time is also increased. In our implementation, at the forward pass, we calculate $B^{(n,k)}$ from the current weights of the BFLayers, which is a bottleneck in training. Introducing a GPU implementation of butterfly operations would greatly reduce training time.

6. Conclusion and Future Work

In this paper, we demonstrated how a family of efficient transformations referred to as the Butterfly Transforms can replace pointwise convolutions in various neural architectures to reduce the computation while maintaining accuracy. We explored many design decisions for this block including residual connections, non-linearities, weight decay, the power of the BFT , and also introduce a new weight initialization, which allows us to significantly outperform all other structured matrix approaches for efficient channel fusion that we are aware of. We also provided a set of principles for fusion network design, and BFT exhibits all these properties.

As a drop-in replacement for pointwise convolutions in efficient Convolutional Neural Networks, we have shown that our method significantly increases accuracy of models, especially at the low FLOP range, and can enable new capabilities on resource constrained edge devices. It is worth noting that these neural architectures have not at all been optimized for BFT , and we hope that this work will lead to more research towards networks designed specifically with the Butterfly Transform in mind, whether through manual design or architecture search. BFT can also be extended to other domains, such as language and speech, as well as new types of architectures, such as Recurrent Neural Networks and Transformers.

We look forward to future inference implementations of Butterfly structures which will hopefully validate our hypothesis that this block can be implemented extremely efficiently, especially on embedded devices and FPGAs. Finally, one of the major challenges we faced was the large amount of time and GPU memory necessary to train BFT , and we believe there is a lot of room for optimizing training of this block as future work.

Acknowledgement

References

[1] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An architecture for ultralow power binary-weight cnn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[2] Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. LCNN: lookup-based convolutional neural network. *CoRR*, abs/1611.06473, 2016.

[3] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.

[4] Krzysztof Choromanski, Mark Rowland, Wenyu Chen, and Adrian Weller. Unifying orthogonal Monte Carlo methods. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1203–1212, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[5] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training neural networks with weights and activations constrained to+ 1 or- 1. *arXiv preprint arXiv:1602.02830*, 2016.

[6] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. *arXiv preprint arXiv:1903.05895*, 2019.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[8] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[9] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408. ACM, 2017.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[11] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[12] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[14] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019.

[15] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[16] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018.

[17] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[18] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

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

[20] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott A. Skirlo, Max Tegmark, and Marin Soljacic. Tunable efficient unitary neural networks (EUNN) and their application to RNN. *CoRR*, abs/1612.05231, 2016.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[22] Quoc Le, Tamas Sarlos, and Alex Smola. Fastfood - approximating kernel expansions in loglinear time. In *30th International Conference on Machine Learning (ICML)*, 2013.

[23] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[24] Chong Li and CJ Richard Shi. Constrained optimization based low-rank approximation of deep neural networks. In *ECCV*, 2018.

[25] Yingzhou Li, Xiuyuan Cheng, and Jianfeng Lu. Butterfly-net: Optimal function representation based on convolutional neural networks. *arXiv preprint arXiv:1805.07451*, 2018.

[26] Yingzhou Li, Haizhao Yang, Eileen R. Martin, Kenneth L. Ho, and Lexing Ying. Butterfly factorization. *Multiscale Modeling Simulation*, 13:714–732, 2015.

[27] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[28] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

[29] Michaël Mathieu and Yann LeCun. Fast approximation of rotations and hessians matrices. *CoRR*, abs/1404.7195, 2014.

[30] Sachin Mehta, Mohammad Rastegari, Linda Shapiro, and Hannaneh Hajishirzi. Espnetv2: A light-weight, power efficient, and general purpose convolutional neural network. In *CVPR*, 2019.

[31] Tatsuya Member and Kazuyoshi Member. Bidirectional learning for neural network having butterfly structure. *Systems and Computers in Japan*, 26:64 – 73, 04 1995.

[32] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. Acdc: A structured efficient linear layer. *CoRR*, abs/1511.05946, 2015.

[33] Marina Munkhoeva, Yermek Kapushev, Evgeny Burnaev, and Ivan V. Oseledets. Quadrature-based features for kernel approximation. *CoRR*, abs/1802.03832, 2018.

[34] D. Stott Parker. Random butterfly transformations with applications in computational linear algebra. Technical report, 1995.

[35] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[36] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[37] Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6655–6659, 2013.

[38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2014.

[40] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3088–3096. Curran Associates, Inc., 2015.

[41] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS*, 2014.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[43] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

[44] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *NIPS*, 2016.

[45] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. *CoRR*, abs/1906.00586, 2019.

[46] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *arXiv preprint arXiv:1812.03443*, 2018.

[47] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR*, 2016.

[48] Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1379–1388, 2017.

[49] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alexander J. Smola, Le Song, and Ziyu Wang. Deep fried convnets. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1476–1483, 2014.

[50] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[51] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Building efficient deep neural networks with unitary group convolutions. *CoRR*, abs/1811.07755, 2018.

[52] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[53] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[54] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.