Butterfly Transform: An Efficient FFT Based Neural Architecture Design

Keivan Alizadeh-Vahid, Anish Prabhu, Ali Farhadi, Mohammad Rastegari
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 $O(n^2)$ to $O(n \log n)$ 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. Code at https://github.com/keivanalizadeh/ButterflyTransform

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 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 $1 \times 1$ 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.

Figure 1: Replacing pointwise convolutions with BFT in state-of-the-art architectures results in significant accuracy gains in resource constrained settings.
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 5. 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 ImageNet2 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 2a) 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 [2], 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][37][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][45][4] 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 $X$ of size $n_m \times h \times w$, where $n$ is the number of channels and $w, h$ are the width and height respectively. The size of
the weight tensor $W$ is $n_{\text{out}} \times n_{\text{in}} \times 1 \times 1$ and the output tensor $Y$ 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 $O(n^2wh)$, and this is mainly influenced by the number of channels $n$. We propose to use Butterfly Transform as a layer, which has $O((n \log n)wh)$ 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 $P$ as follows:

$$Y = P(X; W)$$

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

$$\tilde{Y} = \tilde{W}\tilde{X}$$

where $\tilde{Y}$ is the matrix representation of the output tensor $Y$. This can be seen as a linear transformation of the vectors in the columns of $\tilde{X}$ using $\tilde{W}$ as a transformation matrix. The linear transformation is a matrix-vector product and its complexity is $O(n^2)$. 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 $O(n \log n)$ edges.

Proof: Suppose there exist $n_i$ nodes in $i^{\text{th}}$ 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:

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

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 $O(n \log n)$ 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, $x_1, \ldots, x_k$. We also partition output channels of this first layer to $k$ parts with $\frac{n}{k}$ size each, $y_1, \ldots, y_k$. We connect elements of $x_i$ to $y_j$ with $\frac{n}{k}$ parallel edges $D_{ij}$. After combining information this way, each $y_i$ contains the information from all channels, then we recursively fuse information of each $y_i$ in the next layers.

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

$$B^{(n,k)} = \begin{pmatrix}
M_1^{(\frac{n}{k}, k)}D_{11} & \cdots & M_1^{(\frac{n}{k}, k)}D_{1k} \\
\vdots & \ddots & \vdots \\
M_k^{(\frac{n}{k}, k)}D_{k1} & \cdots & M_k^{(\frac{n}{k}, k)}D_{kk}
\end{pmatrix}$$

(3)

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

$$B^{(n,k)}x = \begin{pmatrix}
M_1^{(\frac{n}{k}, k)}D_{11} & \cdots & M_1^{(\frac{n}{k}, k)}D_{1k} \\
\vdots & \ddots & \vdots \\
M_k^{(\frac{n}{k}, k)}D_{k1} & \cdots & M_k^{(\frac{n}{k}, k)}D_{kk}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
\vdots \\
x_k
\end{pmatrix}$$

(4)

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

$$B^{(n,k)}x = \begin{pmatrix}
M_1^{(\frac{n}{k}, k)}\sum_{j=1}^{k} D_{1j}x_j \\
\vdots \\
M_k^{(\frac{n}{k}, k)}\sum_{j=1}^{k} D_{kj}x_j
\end{pmatrix}
\begin{pmatrix}
y_1 \\
\vdots \\
y_k
\end{pmatrix}$$

(5)

where $y_i = \sum_{j=1}^{k} D_{ij}x_j$. Note that $M_i^{(\frac{n}{k}, k)}y_i$ 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 $B^{(n,k)}x$. 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 $y_i$ for all $i \in \{1, \ldots, k\}$ is $O(kn)$ therefore:

$$T(n, k) = kT(n/k, k) + O(kn)$$

(6)

$$T(n, k) = O(k(n \log_k n))$$

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

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 and fast factorization for approximating linear transformation. 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 $n \times k \times w$. 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 $O(n)$ (2n operations), therefore, the total complexity of the BFT architecture will be $O(n \log n)$. 
Each Butterfly layer can be augmented by batch norm and non-linearity 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 mainly because its complexity is the same as the BFLayer $O(n)$, 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.

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 $O(n \log n)$ 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.

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 \(B_{u,v}^{n,k}\), 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 \(B_{u,v}^{n,k}\) equal to the expectation of the absolute value of the weights with standard Xavier initialization, which is \(x/2\). Let \(e_1, \ldots, e_{\log(n)}\) be edges on the path \(p\) from input node \(u\) to output node \(v\). We have the following:

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

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

\[
\frac{\log(n)}{2} \times \frac{x}{2} \implies y = \frac{x}{\log(n)} \times 2^\left(\frac{\log(n)}{2} - 1\right). \tag{9}
\]

### 4.1.2 MobileNetV1 + BFT

To add BFT to MobileNetV1, 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 3. 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 MobileNets from about 14M to 150M FLOPs with a spectrum of MobileNetV1+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 6.7% in top-1 accuracy at 23 MFLOPs. Note that MobileNetV1+BFT at 23 MFLOPs has much higher accuracy than MobileNetV1 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 2 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 \(\mathcal{V5}\) 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 [11]. Due to resource constraints, we only trained one variant of MobileNetV3+BFT.


<table>
<thead>
<tr>
<th></th>
<th>Flips</th>
<th>MobileNetV1</th>
<th>MobileNetV1+BFT</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>14M</td>
<td>49.8 (13 M)</td>
<td>55.21 (15 M)</td>
<td>5.41</td>
<td></td>
</tr>
<tr>
<td>20M</td>
<td>55.21 (21 M)</td>
<td>6.60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25M</td>
<td>59.70 (24 M)</td>
<td>1.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40M</td>
<td>61.33 (41 M)</td>
<td>1.03</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: These tables compare the accuracy of ShuffleNetV2, MobileNetV1 and MobileNetV3 when using standard pointwise convolution vs using BFTs.
We extend BPBP to work with pointwise convolutions by using

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 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 5 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 towards zero, will significantly reduce the effect of the $i$ on $o$. 

### Table 3: Residual connections

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>No residual</td>
<td>79.2</td>
</tr>
<tr>
<td>Every-other-Layer</td>
<td>81.12</td>
</tr>
<tr>
<td>First-to-Last</td>
<td>81.75</td>
</tr>
</tbody>
</table>

**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 towards zero, will significantly reduce the effect of the $i$ on $o$. 

### Table 3: Residual connections

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>No residual</td>
<td>79.2</td>
</tr>
<tr>
<td>Every-other-Layer</td>
<td>81.12</td>
</tr>
<tr>
<td>First-to-Last</td>
<td>81.75</td>
</tr>
</tbody>
</table>

**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 5 compares the the learning curves of BFT models with and without non-linear activation functions.
Therefore, weight decay is very destructive in BFT. Figure 5 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 contain \( \log_2(n) \) 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-degree 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 \( \approx 40M \) 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 5 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

Thanks Aditya Kusupati, Carlo Del Mundo, Golnoosh Samei, Hessam Bagherinezhad, James Gabriel and Tim Dettmers for their help and valuable comments. This work is in part supported by NSF IIS 1652052, IIS 1730316, DARPA N66001-19-2-4031, 67102239 and gifts from Allen Institute for Artificial Intelligence.

References


A. Experimental details

Here we explain our experimental setup. For all architectures, we optimize our network by minimizing cross-entropy loss using SGD.

A.1. MobileNetV1+BFT

We have used weight decay of $10^{-5}$. We train for 170 epochs. We have used a constant learning rate 0.5 and decay it by $\frac{1}{10}$ at epochs 140, 160. For details on width multiplier of MobileNet and input resolution on each experiment look at Table 4.

A.2. ShuffleNetV2+BFT

We have used weight decay of $10^{-5}$. We train for 300 epochs. We start with a learning rate of 0.5 linearly decaying it to 0. All of the pointwise convolutions are replaced by BFT as shown in Figure 6 except the first pointwise convolution with input channel size of 24. For comparing under the similar number of FLOPs we have slightly changed ShuffleNet’s layer width to create ShuffleNetV2-1.25. This is the structure which is used for shuffleNetV2-1.25:

<table>
<thead>
<tr>
<th>Layer</th>
<th>output size</th>
<th>Kernel</th>
<th>Stride</th>
<th>Repeat</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>224×224</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Conv 1</td>
<td>112×112</td>
<td>3×3</td>
<td>2</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>Max pool</td>
<td>56×56</td>
<td>3×3</td>
<td>2</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>Stage 2</td>
<td>28×28</td>
<td>2×1</td>
<td>1</td>
<td>3</td>
<td>128</td>
</tr>
<tr>
<td>Stage 3</td>
<td>14×14</td>
<td>2×1</td>
<td>1</td>
<td>7</td>
<td>256</td>
</tr>
<tr>
<td>Stage 4</td>
<td>7×7</td>
<td>2×1</td>
<td>1</td>
<td>3</td>
<td>1024</td>
</tr>
<tr>
<td>Conv 5</td>
<td>7×7</td>
<td>BFT</td>
<td>1</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>Global Pool</td>
<td>1×1</td>
<td>7×7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>41</td>
</tr>
<tr>
<td>FLOPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For details on input resolution on each experiment look at Table 5.

A.3. MobileNetV3+BFT

We have used weight decay of $10^{-5}$. We train for 200 epochs. We start with a warm-up for the first 5 epochs, starting from a learning rate 0.1 and linearly increasing it to 0.5. Then we decay learning rate from 0.5 to 0.0 using a cosine scheme in the remaining 195 epochs. For details on width multiplier and input resolution on each experiment look at Table 6.

![Figure 6: ShuffleNetV2+BFT Block](image-url)
<table>
<thead>
<tr>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>128</td>
<td>14 M</td>
<td>41.50</td>
<td>1.0</td>
<td>96</td>
<td>14 M</td>
<td>46.58</td>
<td>5.08</td>
</tr>
<tr>
<td>0.25</td>
<td>160</td>
<td>21 M</td>
<td>45.50</td>
<td>1.0</td>
<td>128</td>
<td>23 M</td>
<td>52.26</td>
<td>6.76</td>
</tr>
<tr>
<td>0.25</td>
<td>192 224</td>
<td>34 M</td>
<td>47.70</td>
<td>1.0</td>
<td>160</td>
<td>35 M</td>
<td>54.30</td>
<td>6.60</td>
</tr>
<tr>
<td>0.50</td>
<td>128</td>
<td>49 M</td>
<td>56.30</td>
<td>1.0</td>
<td>192</td>
<td>51 M</td>
<td>57.56</td>
<td>1.26</td>
</tr>
<tr>
<td>0.50</td>
<td>192 110 M</td>
<td>61.70</td>
<td>2.0</td>
<td>2.0</td>
<td>192 112 M</td>
<td>63.03</td>
<td>1.33</td>
<td></td>
</tr>
<tr>
<td>0.50</td>
<td>224 150 M</td>
<td>63.30</td>
<td>2.0</td>
<td>2.0</td>
<td>224 150 M</td>
<td>64.32</td>
<td>1.02</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Comparison between Mobilenet and Mobilenet+BFT. For comparison under similar number of FLOPs we have used wider channels in Mobilenet+BFT.

<table>
<thead>
<tr>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>128</td>
<td>14 M</td>
<td>50.86*</td>
<td>1.25</td>
<td>128</td>
<td>14 M</td>
<td>55.26</td>
<td>4.4</td>
</tr>
<tr>
<td>0.50</td>
<td>160</td>
<td>21 M</td>
<td>55.21*</td>
<td>1.25</td>
<td>160</td>
<td>21 M</td>
<td>57.83</td>
<td>2.62</td>
</tr>
<tr>
<td>0.50</td>
<td>224</td>
<td>41 M</td>
<td>59.70*</td>
<td>1.25</td>
<td>224 41 M</td>
<td>60.30</td>
<td>1.63</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Table 5: Comparison between ShuffleNetV2 and ShuffleNetV2+BFT. For comparison under similar number of FLOPs we have used wider channels in ShuffleNetV2+BFT.

<table>
<thead>
<tr>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>width</th>
<th>resolution</th>
<th>flops</th>
<th>Accuracy</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small-0.35</td>
<td>224</td>
<td>13 M</td>
<td>49.8</td>
<td>Small-0.5</td>
<td>224</td>
<td>15 M</td>
<td>55.21</td>
<td>5.41</td>
</tr>
</tbody>
</table>

Table 6: Comparison between MobileNetV3 and MobileNetV3+BFT. For comparison under similar number of FLOPs we have used wider channels in MobileNetV3+BFT.