Efficient Compact Bilinear Pooling via Kronecker Product

Tan Yu, Yunfeng Cai, Ping Li

Cognitive Computing Lab
Baidu Research
10900 NE 8th St. Bellevue, Washington 98004, USA
No.10 Xibeiwang East Road, Beijing 100193, China
{tanyu01, caiyunfeng, liping11}@baidu.com

Abstract

Bilinear pooling has achieved excellent performance in fine-grained recognition tasks. Nevertheless, high-dimensional bilinear features suffer from over-fitting and inefficiency. To alleviate these issues, compact bilinear pooling (CBP) methods were developed to generate low-dimensional features. Although the low-dimensional features from existing CBP methods enable high efficiency in subsequent classification, CBP methods themselves are inefficient. Thus, the inefficiency issue of the bilinear pooling is still unsolved. In this work, we propose an efficient compact bilinear pooling method to solve the inefficiency problem inherited in bilinear pooling thoroughly. It decomposes the huge-scale projection matrix into a two-level Kronecker product of several small-scale matrices. By exploiting the "vec trick" and the tensor modal product, we can obtain the compact bilinear feature through the decomposed projection matrices in a speedy manner. Systematic experiments on four public benchmarks using two backbones demonstrate the efficiency and effectiveness of the proposed method in fine-grained recognition.

Introduction

Visual feature pooling is a vital part for image representation learning. Compared with widely-used sum-pooling and max-pooling (Krizhevsky, Sutskever, and Hinton 2012; He et al. 2016), bilinear pooling (Tenenbaum and Freeman 2000) has achieved competitive performance in fine-grained recognition (Lin, RoyChowdhury, and Maji 2015; Gao et al. 2016; Wang et al. 2016; Li et al. 2017a). Given an image, through a series of convolution layers, a set of d-dimensional local convolutional features $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ are generated. Each local feature represents a local region in the image. Bilinear pooling aggregates local convolutional features into a bilinear matrix through $\mathbf{M} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{d\times d}$. Then $\mathbf{M}$ is unfolded into a vector $\mathbf{b} = \mathrm{vec}(\mathbf{M})$, which is further fed into a fully-connected layer for classification.

Straightforwardly, bilinear pooling encodes the correlation between every two channels, that is, the second-order statistics. It possesses powerful representation capability to describe the fine-grained details in the image. Nevertheless, the bilinear feature $\mathbf{b} \in \mathbb{R}^{d^2}$ is high-dimensional, which limits its effectiveness and efficiency. From the perspective of effectiveness, when using a standard fully-connected layer as a classifier, it contains d^2c parameters where c is the number of classes. Assuming $d = 512$ and $c = 200$, it will need around 52 million parameters, making the classifier prone to over-fitting. It is especially serious in the few-shot learning scenarios. The over-fitting caused by the huge number of parameters will significantly affect the recognition accuracy of the classifier. On the efficiency side, the huge number of parameters in the classifier makes the computation and memory cost expensive. Particularly, in the image retrieval scenario, the high-dimension features will take huge costs when calculating the similarity between features.

A straightforward method for feature dimension reduction is linear projection. Given the bilinear feature $\mathbf{b} \in \mathbb{R}^{d^2}$, we can learn a projection matrix $\mathbf{P} \in \mathbb{R}^{D\times d^2}$ ($D \ll d^2$) for dimension reduction. Then the low-dimension feature is obtained by $\widehat{\mathbf{b}} = \mathbf{P}\mathbf{b}$. However, since d^2 is large, making the projection $\mathbf{P}\mathbf{b}$ extremely slow ($2d^2D$ floating-point operations). Meanwhile, $\mathbf{P}$ additionally brings huge number of parameters (d^2D), making the over-fitting more serious.

To obtain a low-dimensional bilinear feature, compact bilinear pooling (CBP) (Gao et al. 2016) relies on Tensor Sketch (TS) (Pham and Pagh 2013) and Random Maclaurin (RM) (Kar and Karnick 2012). Both TS and RM are based on random projections. Basically, the low-dimensional features from TS and RM statistically satisfy the property of bilinear features. Specifically, the inner-product between low-dimensional features from TS and RM is an unbiased estimation of the inner-product between two bilinear features. Nevertheless, to achieve a small estimation error, the random projection matrix should be large-scale, making the random projection slow. Recently, RM is revisited by MLB (Kim et al. 2017) which achieves excellent performance in cross-modal understanding and also revisited by HBP (Yu et al. 2018) which fuses features from different layers. However, the inefficiency problem of RM is still unsolved.

In this work, we seek to address the inefficiency problem in compact bilinear pooling methods. Recall that the bilinear feature is obtained by unfolding the bilinear matrix $\mathbf{b} = \mathrm{vec}(\mathbf{X}\mathbf{X}^\top)$, thus the linear projection $\mathbf{P}\mathbf{b}$ is equivalent to computing $\mathbf{P}\mathrm{vec}(\mathbf{X}\mathbf{X}^\top)$. Meanwhile, a well known vec trick gives a property that $\mathrm{vec}(\mathbf{B}\mathbf{Y}\mathbf{A}^\top) = (\mathbf{A}\otimes\mathbf{B})\mathrm{vec}(\mathbf{Y})$. To exploit the nice property of vec trick, we construct the projection matrix by a Kronecker product of two matrices,

Method	# of para.	FLOPs	Latency
RM	9.4×10^6	1.5×10^{10}	87ms
TKPF (ours)	192	3.9×10^6	3ms

Table 1: Comparisons with random Maclaurin (RM) used in CBP (Gao et al. 2016), HBP (Yu et al. 2018) and MLB (Kim et al. 2017) based on VGG16 backbone.

i.e., $\mathbf{P} = \mathbf{A} \otimes \mathbf{B}$. By using the vec trick, the linear projection $\mathbf{Pb}$ is equivalent to computing $\mathrm{vec}(\mathbf{BXX}^\top \mathbf{A}^\top)$. Since $\mathbf{A}$ and $\mathbf{B}$ are in far smaller scale than the original projection matrix $\mathbf{P}$, the computation and memory cost is significantly reduced. We term the factorization on the original projection matrix $\mathbf{P}$ as the first-level Kronecker product factorization. To further improve the efficiency, we factorize $\mathbf{A}$ as well as $\mathbf{B}$ into another Kronecker product of smaller-scale matrices. We term the factorization for $\mathbf{A}$ and $\mathbf{B}$ as second-level Kronecker product factorization. We will show that the second-level factorization makes tensor modal product feasible, further reducing the computation and memory cost.

In a nutshell, we factorize the huge-scale projection matrix $\mathbf{P}$ into a two-level Kronecker product, making the compact bilinear pooling efficient. We term our method as two-level Kronecker product factorization (TKPF). Table 1 shows the number of parameters, floating-point operations (FLOPs), and the factual latency in the inference phase of RM and the proposed TKPF. Based on TKPF, we built an efficient compact bilinear pooling network. We conduct comprehensive experiments on four public benchmarks using two backbones, VGG16 and ResNet50. Our model achieves competitive classification accuracy in fine-grained recognition as existing compact bilinear pooling methods with considerably higher efficiency in computation and memory.

Related Work

BCNN (Lin, RoyChowdhury, and Maji 2015) and O^2P (Ionescu, Vantzos, and Sminchisescu 2015) are pioneering works for applying bilinear pooling in neural networks. The following works improve the bilinear network in two directions: 1) improve the feature's effectiveness; 2) reduce dimension for higher efficiency.

Effectiveness improvement. Ker-RP (Wang et al. 2015; Zhang et al. 2021) enhances the representation capability of bilinear feature by exploiting kernels. G2DeNet (Wang, Li, and Zhang 2017), FASON (Dai, Ng, and Davis 2017) and MoNet (Gou et al. 2018) enrich the representation by taking the first-order statistics into consideration beside the bilinear pooling, the second-order statistics. In parallel, KP (Cui et al. 2017), HOP (Koniusz et al. 2017) and HOK (Cherian, Koniusz, and Gould 2017) exploit higher-order pooling beside the second-order pooling. HBP (Yu et al. 2018) enhances the representing capability by exploiting the features from multiple layers. Improved bilinear pooling (IBP) (Lin and Maji 2017) and MPN-COV (Li et al. 2017a) conduct the root normalization on the bilinear matrix to suppress burstiness. PN (Koniusz, Zhang, and Porikli 2018) also investigates the influence of matrix normalization on the performance of bilinear features. GP (Wei et al. 2018) conducts normalization by making magnitudes of orthogonal components in the bilinear feature identical. Benefited from the normalization, the performance of bilinear features gets improved considerably. Since the original matrix normalization requires SVD, which does not support parallelism well, it is inefficient for GPU computing. To enhance the efficiency in GPU, IBP (Lin and Maji 2017) adopts Newton-Schulz (NS) iteration (Higham 2008) to obtain the approximated matrix square root. It only relies on matrix-matrix multiplications and thus is suitable for GPU computing. iSQRT-COV (Li et al. 2018) implements NS in both forward and backward propagation, making the matrix normalization efficient both in inference and training. RUN (Yu, Cai, and Li 2020) further speeds up matrix normalization by utilizing the power method. Lin *et al.* (Lin, Maji, and Koniusz 2018) adopt democratic aggregation (Jégou and Zisserman 2014) to weight the second-order local features to balance their contributions to the final feature.

Dimension reduction. The high dimension of the bilinear feature limits its efficiency for classification and makes it prone to over-fitting. To speed up the classification and suppress over-fitting, FBN (Li et al. 2017b) reduces the dimension of the weight matrix in the classifier by a product of low-rank matrices. In parallel, some methods focus on dimension reduction for the bilinear feature. BCNN (Lin, RoyChowdhury, and Maji 2015) adopts PCA to learn the projection matrix for reducing the dimension of local features. To be specific, it uses the projection matrix learned from PCA as initialization of a 1×1 convolution layer and then trains the network in an end-to-end manner. The PCA baseline is also utilized in LRBP (Kong and Fowlkes 2017) and iSQRT-COV (Li et al. 2018) for dimension reduction on the local feature. In parallel, CBP discovers the connection between bilinear pooling and the polynomial kernel. It adopts the methods for kernel approximation for reducing the dimension of bilinear features. To be specific, CBP attempts two methods, tensor sketch (Pham and Pagh 2013) and random Maclaurin (RM) (Kar and Karnick 2012), which achieve better performance than the PCA baseline. Among them, RM relies on two random projection matrices. It conducts two random projections for each local feature, followed by a Hadamard multiplication, which is quite suitable for GPU. RM is revisited in MLB (Kim et al. 2017) for cross-modal understanding and HBP (Yu et al. 2018) for fusing features in different layers. Nevertheless, to achieve as good performance as the full bilinear features, the projection matrices used in RM are normally large-scale, limiting its efficiency. SRM (Yu, Li, and Li 2021) improves RM through devising a shifted operation. Wang *et al.* (Wang et al. 2021) extends the 1×1 convolution layer used in iSQRT-COV to two consecutive convolution layers, achieving higher accuracy.

Preliminary

We denote local features by $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$. Bilinear pooling obtains the bilinear matrix by

$$\mathbf{B} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{d \times d}.$$

Then $\mathbf{B}$ is unfolded into a vector $\mathbf{b} = \mathrm{vec}(\mathbf{B}) \in \mathbb{R}^{d^2}$ as the image representation for classification or retrieval.

Kronecker product. Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{p\times q}$. The Kronecker product $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{pm\times qn}$ is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix}. \tag{1}$$

Note that, $\mathbf{A} \otimes \mathbf{B}$ contains $pqmn$ entries whereas $\mathbf{A}$ and $\mathbf{B}$ only contain $mn + pq$ entries, which is far more less than $pqmn$. Furthermore, let $\mathbf{M} \in \mathbb{R}^{q\times n}$. It holds that

$$(\mathbf{A} \otimes \mathbf{B})\text{vec}(\mathbf{M}) \equiv \text{vec}(\mathbf{B}\mathbf{M}\mathbf{A}^\top). \tag{2}$$

Hereafter, Eq. (2) is referred to as the "**vec trick**". The left-hand side of Eq. (2) is the product of an mq-by-np matrix and a vector. Generally speaking, the matrix-vector multiplication between an mq-by-np matrix and a vector takes $2mnpq$ floating-point operations (FLOPs). When the matrix has a Kronecker product structure, using the vec trick, it only takes $2mp(n + q)$ FLOPs, which is far less than $2mnpq$. To summarise, if a large scale matrix has a Kronecker-product structure, we can store it in memory efficiently, and the associated matrix-vector multiplication is fast.

Modal unfolding. Let $\boldsymbol{\mathcal{X}}$ be a tensor of size $d_1 \times \cdots \times d_N$ and $D = d_1 \cdots d_N$, then the mode-n unfolding of $\boldsymbol{\mathcal{X}}$, denoted by $\boldsymbol{\mathcal{X}}_{(n)}$, is a d_n-by-D/d_n matrix, whose columns are the mode-n fibers of $\boldsymbol{\mathcal{X}}$.

Tensor modal product. Let $\boldsymbol{\mathcal{X}}$ be a tensor of size $d_1 \times \cdots \times d_N$, $\mathbf{U}$ be a matrix of size $d_u \times d_n$. The mode-n product of $\boldsymbol{\mathcal{X}}$ and $\mathbf{U}$ is defined by $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U}$, where

$$\boldsymbol{\mathcal{Y}}_{i_1,\cdots,i_{n-1},j,i_{n+1},\cdots,i_N} = \sum_{i_n=1}^{d_n} \boldsymbol{\mathcal{X}}_{i_1,\cdots,i_N} \mathbf{U}_{j,i_n}.$$

Using modal unfolding, the above equation is equivalent to

$$\boldsymbol{\mathcal{Y}}_{(n)} = \mathbf{U}\boldsymbol{\mathcal{X}}_{(n)} \in \mathbb{R}^{d_u \times D/d_n},$$

where $D = d_1 \dots d_N$. Therefore, computing $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U}$ takes $2Dd_u$ FLOPs.

Multilinear product and its unfolding. Given a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{d_1\times\cdots\times d_N}$ and $\mathbf{U}_n \in \mathbb{R}^{k_n\times d_n}$ ($n \in [2, N]$), the multilinear product between $\boldsymbol{\mathcal{X}}$ and $\{\mathbf{U}_n\}_{n=2}^N$ is defined as

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_2 \mathbf{U}_2 \times_3 \mathbf{U}_3 \cdots \times_N \mathbf{U}_N. \tag{3}$$

The mode-n unfolding of $\boldsymbol{\mathcal{Y}}$ satisfies the below property:

$$\boldsymbol{\mathcal{Y}}_{(1)}^\top = (\mathbf{U}_N \otimes \mathbf{U}_{N-1} \otimes \cdots \otimes \mathbf{U}_2)\boldsymbol{\mathcal{X}}_{(1)}^\top. \tag{4}$$

The readers can refer to (Van Loan and Golub 2012, Chapter 12) for the derivation.

Tensorization of Matrices. Given a matrix $\mathbf{M} \in \mathbb{R}^{N\times d}$. If d can be factorized into $d = d_1 d_2$, we can fold $\mathbf{M}$ into a third-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{N\times d_1\times d_2}$. If d can be factorized into $d = d_1 d_2 d_3$, we can fold $\mathbf{M}$ into a fourth-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{N\times d_1\times d_2\times d_3}$. In general, if d can be factorized into $d = d_1 d_2 \cdots d_R$, we can fold $\mathbf{M}$ into a $(R + 1)$th-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{N\times d_1\times\cdots\times d_R}$. See Figure 1 for an illustration.

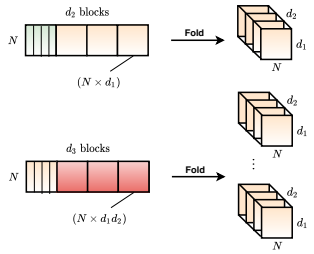

Figure 1: Tensorization of block matrices, upper: a third order tensor, lower: a fourth order tensor.

Two-level Kronecker Product Factorization

First-level Kronecker Product Factorization

As we mentioned, the high-dimensional bilinear feature $\text{vec}(\mathbf{X}\mathbf{X}^\top) \in \mathbb{R}^{d^2}$ tends to make the classifier suffer from over-fitting and inefficiency. Using a projection matrix $\mathbf{P} \in \mathbb{R}^{D\times d^2}$ with $D \ll d^2$, we can reduce the dimension by

$$\mathbf{b} = \mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top). \tag{5}$$

However, $\mathbf{P}$ is large-scale. Thus the projection $\mathbf{P}\mathbf{b}$ taking $2d^2D$ FLOPs is extremely slow. Besides, the number of parameters in $\mathbf{P}$, which is Dd^2, is also extremely large, making the over-fitting problem more serious. To alleviate the over-fitting problem and improve the efficiency, one simple yet effective way is to impose a Kronecker-product structure on the projection matrix $\mathbf{P}$. To be specific, we constrain the projection matrix in the form of Kronecker product:

$$\mathbf{P} = \mathbf{A} \otimes \mathbf{B}, \tag{6}$$

where $\mathbf{A} \in \mathbb{R}^{a\times d}$, $\mathbf{B} \in \mathbb{R}^{b\times d}$ and $ab = D$. In this case, the number of parameters in $\{\mathbf{A}, \mathbf{B}\}$ is only $(a+b)d$, which is significantly smaller than that in $\mathbf{P}$, abd^2. Meanwhile, by using the vec trick, computing $(\mathbf{A} \otimes \mathbf{B})\text{vec}(\mathbf{X}\mathbf{X}^\top)$ is significantly more efficient than computing $\mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top)$. To be specific, using the vec trick, we have

$$(\mathbf{A} \otimes \mathbf{B})\text{vec}(\mathbf{X}\mathbf{X}^\top) = \text{vec}[\mathbf{B}\mathbf{X}(\mathbf{A}\mathbf{X})^\top],$$

where $\text{vec}[\mathbf{B}\mathbf{X}(\mathbf{A}\mathbf{X})^\top]$ can be computed very efficiently through computing two projections $\mathbf{S} = \mathbf{B}\mathbf{X}$ and $\mathbf{T} = \mathbf{A}\mathbf{X}$ followed by a matrix-matrix multiplication $\mathbf{S}\mathbf{T}^\top$. Since $\mathbf{B}$, $\mathbf{A}$ are small-scale matrices, computing $\mathbf{B}\mathbf{X}$, $\mathbf{A}\mathbf{X}$ and $\mathbf{S}\mathbf{T}^\top$ takes only $2N[(a + b)d + D]$ FLOPs, which is considerably faster than computing the $\mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top)$ with $2d^2D$ FLOPs.

To enhance the representing capability, we factorize $\mathbf{P}$ into a summation of multiple Kronecker products:

$$\mathbf{P} = \sum_{q=1}^{Q} \mathbf{A}^{(q)} \otimes \mathbf{B}^{(q)}. \tag{7}$$

Using the vec trick, $\mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top)$ is efficiently computed by

$$\mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top) = \sum_{q=1}^{Q} \text{vec}[\mathbf{B}^{(q)}\mathbf{X}(\mathbf{A}^{(q)}\mathbf{X})^\top]. \tag{8}$$

Second-level Kronecker Product Factorization

The first-level Kronecker product factorization converts a large scale projection matrix into smaller-scale projection matrices $\mathbf{A}^{(q)}$ and $\mathbf{B}^{(q)}$. Meanwhile, it converts a heavy projection $\mathbf{P}\text{vec}(\mathbf{X}\mathbf{X}^\top)$ into projections $\mathbf{A}^{(q)}\mathbf{X}$ and $\mathbf{B}^{(q)}\mathbf{X}$ and earns higher efficiency. Without ambiguity and for ease of notations, we omit the superscript in $\mathbf{A}^{(q)}$ and $\mathbf{B}^{(q)}$. To further improve the computation and memory efficiency, we seek to further decompose $\mathbf{A}$ as well as $\mathbf{B}$ as a Kronecker product of smaller-scale matrices.

A straightforward factorization for $\mathbf{A}$ as well as $\mathbf{B}$ is

$$\mathbf{A} = \widehat{\mathbf{A}}_1 \otimes \widehat{\mathbf{A}}_2,\ \mathbf{B} = \widehat{\mathbf{B}}_1 \otimes \widehat{\mathbf{B}}_2. \tag{9}$$

However, when we plug Eq. (9) in Eq. (6), it leads to a fourth order polynomial of parameters, which is not friendly to optimizer and tends to make the training unstable. Thus, we factorize $\mathbf{A}$ and $\mathbf{B}$ in the following form:

$$\mathbf{A} = \mathbf{I}_r \otimes \widehat{\mathbf{A}},\ \mathbf{B} = \widehat{\mathbf{B}} \otimes \mathbf{I}_r, \tag{10}$$

where $\mathbf{I}_r$ is the identity matrix with rank r, and $\mathbf{I}_r$ is parameter-free. To make it clear, we term that in Eq. (6) as the first-level Kronecker product factorization, and term that in Eq. (10) as the second-level Kronecker product factorization. Since $\mathbf{A} \in \mathbb{R}^{a\times d}$ and $\mathbf{B} \in \mathbb{R}^{b\times d}$, based on the definition of Kronecker product, it is straightforward to get

$$\widehat{\mathbf{A}} \in \mathbb{R}^{\frac{a}{r}\times\frac{d}{r}},\ \widehat{\mathbf{B}} \in \mathbb{R}^{\frac{b}{r}\times\frac{d}{r}}. \tag{11}$$

In this case, the number of parameters in $\widehat{\mathbf{A}}$ is only ad/r^2, which is only $1/r^2$ of number of parameters in original $\mathbf{A}$. That is, the second-level Kronecker product factorization further brings a r^2 times reduction on the number of parameters. When we choose a large value for r, say $r = a$, the number of parameters in $\widehat{\mathbf{A}}$ is only $ad/a^2 = d/a$, which is much smaller than the number of parameters in $\mathbf{A}$, da. Below we further show the efficiency improvement brought by the second-level Kronecker product factorization.

Algorithm 1: Tensor Modal Product

1: **Input:** r, $\mathbf{X} \in \mathbb{R}^{d\times N}$, $\widehat{\mathbf{A}} \in \mathbb{R}^{\frac{a}{r}\times\frac{d}{r}}$.
2: **Output:** $\mathbf{T} = [\mathbf{I}_r \otimes \widehat{\mathbf{A}}]\mathbf{X}$.
3: Reshape $\mathbf{X}^\top$ into a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{N\times\frac{d}{r}\times r}$.
4: Perform modal product $\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{X}} \times_2 \widehat{\mathbf{A}} \times_3 \mathbf{I}_r$.
5: Unfold the tensor $\boldsymbol{\mathcal{T}}$ along mode-1, and set $\mathbf{T} = \boldsymbol{\mathcal{T}}_{(1)}^\top$.

With the help of tensorization and tensor modal product, using the special structure of $\mathbf{A}$ and $\mathbf{B}$ in Eq. (10), we can compute $\mathbf{AX}$ and $\mathbf{BX}$ in a fast manner. We show the details of computing $\mathbf{AX}$ in Algorithm 1 and $\mathbf{BX}$ can be computed in the same manner. Using the equivalence between Eq. (3) and (4), we know that $\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{X}} \times_2 \widehat{\mathbf{A}} \times_3 \mathbf{I}_r$ is equivalent to $\boldsymbol{\mathcal{T}}_{(1)}^\top = (\mathbf{I}_r \otimes \widehat{\mathbf{A}})\boldsymbol{\mathcal{X}}_{(1)}^\top = \mathbf{AX}$, based on which, we can claim that on output of Algorithm 1, $\mathbf{T} = \boldsymbol{\mathcal{T}}_{(1)}^\top = \mathbf{AX}$.

Since $\mathbf{I}_r$ is an identity matrix, we can omit it in modal product and directly compute $\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{X}} \times_2 \widehat{\mathbf{A}}$. It takes $2adN/r$ FLOPs. When $r = a$, $\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{X}} \times_2 \widehat{\mathbf{A}}$ takes only $2dN$ FLOPs, which is far cheaper than computing $\mathbf{AX}$ with $2adN$ FLOPs. The computational complexity analysis for $\mathbf{BX}$ based on $\mathbf{B} = \widehat{\mathbf{B}} \otimes \mathbf{I}_r$ is similar, and hence we omit it here. Plugging Eq. (10) into Eq. (7), we can obtain

$$\widehat{\mathbf{P}} = \sum_{q=1}^{Q} (\mathbf{I}_r \otimes \widehat{\mathbf{A}}^{(q)}) \otimes (\widehat{\mathbf{B}}^{(q)} \otimes \mathbf{I}_r). \tag{12}$$

That is, the projection matrix is decomposed into a summation of a two-level Kronecker product of small-scale matrices. Figure 2 visualizes the pipeline of the proposed two-level Kronecker product factorization.

Alternative Structure

Second-level Kronecker product summation. Recall from Eq. (7) that, in the first-level Kronecker product factorization, we use a summation of multiple items. Similarly, in the second-level Kronecker product factorization, we can also extend the original single-entry Kronecker product to a summation of multiple Kronecker products:

$$\mathbf{A} = \sum_{p=1}^{P} \mathbf{I}_r \otimes \widehat{\mathbf{A}}^{(p)},\ \ \mathbf{B} = \sum_{p=1}^{P} \widehat{\mathbf{B}}^{(p)} \otimes \mathbf{I}_r, \tag{13}$$

Our experiments show that the second-level Kronecker product summation cannot improve the accuracy, and thus we still adopt the single-entry form in Eq. (10).

Symmetric structure. In Eq. (10), $\mathbf{A}$ and $\mathbf{B}$ adopt different factorization forms. We term it as asymmetric structure. An alternative choice is a symmetric structure:

$$\mathbf{A} = \mathbf{I}_r \otimes \widehat{\mathbf{A}},\ \mathbf{B} = \mathbf{I}_r \otimes \widehat{\mathbf{B}}. \tag{14}$$

Our experiments show that the asymmetric structure achieves a better performance than its symmetric counterpart. By default, we adopt the asymmetric structure.

Relation with Existing Methods

PCA baseline is exploited in BCNN (Lin, RoyChowdhury, and Maji 2015) and iSQRT-COV (Li et al. 2018) for dimension reduction for bilinear features. It adopts a 1×1 convolutional layer to reduce the dimension of the local feature from

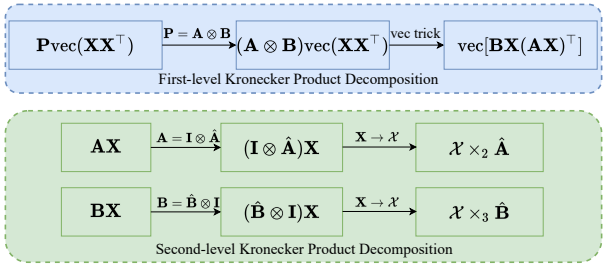

Figure 2: The pipeline of our two-level Kronecker product factorization. In the first level, the efficiency is boosted through exploiting the vec trick. In the second level, the efficiency is enhanced through using tensor modal product.

method	formulation	# of parameters		FLOPs	
		symbolic	numeric	symbolic	numeric
PCA	$\mathbf{b} = \mathbf{W}_{\text{PCA}}\mathbf{X}(\mathbf{W}_{\text{PCA}}\mathbf{X})^\top$	dm	3.3×10^4	$2(md + D)N$	5.8×10^7
RM	$(\mathbf{W}_1\mathbf{X} \odot \mathbf{W}_2\mathbf{X})\mathbf{1}_N$	$2dD$	4.2×10^6	$2(2dD + D)N$	6.6×10^9
TKPF	$\sum_{q=1}^{Q}\{[(\mathbf{I}_r \otimes \widehat{\mathbf{A}}^{(q)})\mathbf{X}][(\widehat{\mathbf{B}}^{(q)} \otimes \mathbf{I}_r)\mathbf{X}]^\top\}$	$\left(\frac{ad}{r^2} + \frac{bd}{r^2}\right)Q$	32	$2\left(\frac{ad}{r} + \frac{bd}{r} + D\right)NQ$	1.6×10^7

Table 2: Comparisons between PCA, Random Maclaurin (RM) and our two-level Kronecker product factorization (TKPF). $\mathbf{W}_{\text{PCA}} \in \mathbb{R}^{d\times m}$, $d = 512$ and $m = 64$. $\mathbf{W}_1/\mathbf{W}_2 \in \mathbb{R}^{d\times D}$ and $D = m^2 = 4096$. $\mathbf{1}_N$ is the N-dimension column vector with all elements as 1. $\mathbf{X} \in \mathbb{R}^{d\times N}$ and $N = 28^2 = 784$. $Q = 2$. $\widehat{\mathbf{A}}^{(q)}/\widehat{\mathbf{B}}^{(q)} \in \mathbb{R}^{a/r\times d/r}$ where $r = 64$ and $a = b = 64$.

d to m. Then the bilinear pooling is conducted on the compact local features and generates m^2-dimensional feature. That is, given N local features $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, PCA baseline generates the compact local features by

$$\widehat{\mathbf{X}} = \mathbf{W}\mathbf{X},\ \mathbf{W} \in \mathbb{R}^{m\times d}. \tag{15}$$

Then the compact bilinear feature is obtained by

$$\widehat{\mathbf{b}} = \text{vec}(\widehat{\mathbf{X}}\widehat{\mathbf{X}}^\top) \in \mathbb{R}^{m^2}. \tag{16}$$

Random Maclaurin (RM) is exploited in CBP (Gao et al. 2016) for compact bilinear pooling. It is revisited in MLB (Kim et al. 2017) for cross-modal understanding and HBP (Yu et al. 2018) for fusing features from different layers. RM utilizes two projection matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ and generates the compact feature:

$$\widehat{\mathbf{b}} = \sum_{i=1}^{N}(\mathbf{W}_1\mathbf{x}_i) \odot (\mathbf{W}_2\mathbf{x}_i),\ \mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{D\times d}, \tag{17}$$

where $\odot$ denotes the Hadamard product. In RM, the number of parameters is $2dD$ and the FLOPs is $4NdD$. To achieve a good performance, D is normally set as a large number, *e.g.*, 10^4. The huge D makes the projections $\mathbf{W}_1\mathbf{x}_i$ and $\mathbf{W}_2\mathbf{x}_i$ for each local feature quite slow. Table 2 compares PCA baseline, random Maclaurin (RM), and the proposed TKPF. Compared with RM, our TKPF significantly reduces the number of parameters and FLOPs.

Efficient Compact Bilinear Network

Based on TKPF, we built an efficient compact bilinear network. Figure 3 visualizes its architecture. Given an image, a set of local features $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \in \mathbb{R}^{d\times N}$ are extracted by a convolutional neural network. Then the compact bilinear feature $\widehat{\mathbf{b}}$ is obtained by TKPF. After that, a fully-connected layer generates the class scores. Following RUN (Yu, Cai, and Li 2020), we conduct matrix normalization on local features $\mathbf{X}$. We also conduct element-wise signed square-root and ℓ_2 normalization on $\widehat{\mathbf{b}}$.

Experiments

Datasets. We conduct experiments on four public benchmarks for fine-grained recognition including FGVC-Aircraft (AIR) (Maji et al. 2013), CUB-200-2011 (CUB) (Wah et al. 2011), MIT scene dataset (Quattoni and Torralba 2009), and Describable Texture Dataset (DTD) (Cimpoi et al. 2014).

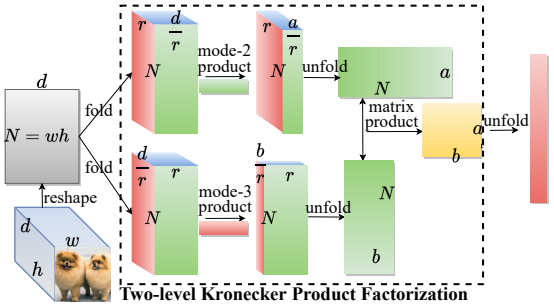

Figure 3: The structure of our network.

Backbone. Following CBP (Gao et al. 2016), we use VGG16 (Simonyan and Zisserman 2015) as the backbone, by default. Each image is resized into 448×448, and generates a $28 \times 28 \times 512$ feature map from conv5-3 layer. The number of local features $N = 28^2 = 784$, and the local feature dimension, d, is 512. We also conduct experiments using ResNet50 (He et al. 2016) as the backbone. It generates $14 \times 14 \times 2048$ feature map. The number of local features $N = 14^2 = 196$, and the local feature dimension, d, is 2048.

Settings. We adopt a two-phase training. In the first phase, we only update parameters in TKPF and classifier layers. In the second phase, we fine-tune parameters of all layers. It is implemented in PaddlePaddle platform developed by Baidu.

Ablation study

We conduct ablation studies on TKPF. By default, we adopt VGG16 as the backbone on ablation studies.

a	AIR	CUB	MIT	DTD	# of pa.	FLOPs
32	90.6	84.0	76.4	61.8	64	6.4M
64	91.5	85.8	79.7	67.3	128	19.3M
96	91.4	86.0	80.5	68.2	192	38.5M
128	90.8	85.8	80.0	68.0	256	64.2M

Table 3: Influence of a. M denotes million.

Influence of a and b. Recall that $\widehat{\mathbf{A}} \in \mathbb{R}^{a/r\times d/r}$ and $\widehat{\mathbf{B}} \in \mathbb{R}^{b/r\times d/r}$. That is, $\mathbf{I}_r \otimes \widehat{\mathbf{A}} \in \mathbb{R}^{a\times d}$ and $\widehat{\mathbf{B}}^{(q)} \otimes \mathbf{I}_r \in \mathbb{R}^{b\times d}$. The dimension of the compact bilinear feature is $D = ab$. By default, we set $a = b$ and thus $D = a^2$. We evaluate the influence of a and b on the performance of the proposed TKPF. We fix $r = 32$ and vary a and b among $\{32, 64, 96, 128\}$. Thus D varies among $\{32^2, 64^2, 96^2, 128^2\}$. As shown in

	VGG16						ResNet50					
	AIR	CUB	MIT	DTD	# of para.	FLOPs	AIR	CUB	MIT	DTD	# of para.	FLOPs
PCA	90.8	84.1	77.6	65.4	49K	92M	90.4	85.2	81.9	69.8	262K	109M
PC	91.1	82.7	75.3	60.9	156K	244M	90.3	80.0	80.2	66.1	557K	225M
RM	91.2	85.6	80.3	67.0	9.4M	15B	92.0	85.9	83.2	70.4	67.1M	26.3B
TKPF	91.4	86.0	80.5	68.2	192	38.5M	92.1	85.7	84.1	71.4	4096	38.5M

Table 4: Comparisons with PCA in Lin, RoyChowdhury, and Maji (2015); Li et al. (2018), progressive convolutions (PC) in Wang et al. (2021), and random Maclaurin (RM) in Gao et al. (2016); Kim et al. (2017); Yu et al. (2018).

Table 3, the accuracy increases as a increases from 32 to 96. When D further increases from 96 to 128, the accuracy becomes slightly worse. This might be due to over-fitting. By default, we set $a = b = 96$, that is, $D = 96^2$.

Influence of r. The identity matrix $\mathbf{I}_r$ is parameter-free. When a and b are fixed, a larger-scale $\mathbf{I}_r$ leads to smaller-scale $\widehat{\mathbf{A}}^{(q)}$ and $\widehat{\mathbf{B}}^{(q)}$, and thus reduces the number of parameters as well as computation cost. We evaluate the influence of the dimension of identity matrices, r, on the recognition accuracy and the efficiency. In the experiments, we fix $a = b = 96$ and $D = 96^2$. As shown in Table 5, as r increases from 4 to 32, the number of parameters and FLOPs are reduced, and the accuracy only slightly changes. We set $r = 32$ by default when using VGG16 backbone.

r	AIR	CUB	MIT	DTD	# of pa.	FLOPs
4	90.8	85.5	79.4	67.4	13K	106M
8	91.1	85.6	79.6	67.4	3.1K	67.4M
16	90.9	85.5	79.9	67.3	768	48.2M
32	91.4	86.0	80.5	68.2	192	38.5M

Table 5: Influence of r. M denotes million.

Influence of Q. Recall from Eq. (7) that we use a summation of Q Kronecker products to enhance the representation capability. We evaluate the influence of Q on TKPF. We fix $a = b = 96$ and $r = 32$. We vary Q among $\{1, 2, 4\}$. As shown in Table 6, as Q increases from 1 to 2, the recognition accuracy gets improved, and meanwhile, FLOPs are doubled. As Q increases from 2 to 4, the recognition accuracy saturates, but FLOPs are further doubled. Considering both effectiveness and efficiency, we set $Q = 2$ by default.

Q	AIR	CUB	MIT	DTD	# of pa.	FLOPs
1	90.9	85.8	79.6	67.1	96	19.3M
2	91.4	86.0	80.5	68.2	192	38.5M
4	91.2	85.9	79.7	68.3	384	77.1M

Table 6: Influence of Q. M denotes million.

Comparisons with other compact bilinear methods

VGG16 backbone. We compare with PCA used in BCNN (Lin, RoyChowdhury, and Maji 2015), random Maclaurin (RM) in CBP (Gao et al. 2016) and progressive convolutions (PC) (Wang et al. 2021). To make a fair comparison, we fix the dimension of compact bilinear features from different methods, $D = 96^2$. Thus, we set $\mathbf{W}_{\mathrm{PCA}} \in \mathbb{R}^{96\times 512}$ in PCA method, and set $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{96^2\times 512}$ in RM method. In PC method, we set the kernel size of the first convolution layer as 512×256 and set that of the second layer as 256×96. Meanwhile, in TKPF, we set $Q = 2$, $a = b = 96$ and $r = 32$. As shown in Table 4, TKPF achieves higher classification accuracy than PCA and PC. Despite that CP achieves an excellent performance on large-scale datasets (Wang et al. 2021), it is not as good as TKPF on our testing datasets. It might be due to that PC brings more parameters, making the over-fitting more serious. Compared with RM, TKPF achieves a comparable accuracy on AIR, CUB, and MIT and a higher accuracy on DTD with significantly fewer parameters and FLOPs.

Method	VGG16		ResNet50	
	BP	total	BP	total
RM	87ms	410ms	161ms	577ms
TKPF	3ms	325ms	3ms	420ms

Table 7: The factual time cost. Latency in bilinear pooling layer (BP) and the whole network (total) are reported.

ResNet50 backbone. We fix the dimension of features of all methods, $D = 128^2$. The dimension of local feature from ResNet50, $d = 2048$. Thus, the projection matrix we used for PCA, $\mathbf{W}_{\mathrm{PCA}}$, is of the size 2048×128. Meanwhile, the projection matrices in RM, $\mathbf{W}_1/\mathbf{W}_2$ is of the size 2048×128^2. In our TKPF, we set $Q = 2$, $a = b = 128$ and $r = 16$. As shown in Table 4, TKPF achieves higher recognition accuracy than PCA and PC. Meanwhile, we earn a comparable accuracy as RM. But we need fewer FLOPs and parameters than RM. An interesting observation is that the FLOPs of TKPF with ResNet50 is the same as VGG16.

Factual time cost. In Table 7, we compare the factual time cost of TKPF in inference with RM. For both RM and TKPF, we use the same settings as Table 4. Since our TKPF only improves the efficiency of the bilinear pooling layer, it does not influence the efficiency of other layers in the network. Thus, we report the latency of the bilinear pooling (BP) layer and that of the whole network as well. The experiments are conducted on a single NVIDIA Titan X (Pascal) GPU card. We report the GPU time cost for a mini-batch of 64 images. As shown in Table 7, using VGG16 backbone, in bilinear pooling layer, our TKPF only takes 3ms, whereas RM takes 87ms. That is, it achieves a $29\times$ speed-up ratio. Meanwhile, using ResNet50 backbone, BP takes 161ms in the bilinear pooling layer. In contrast, we only take 3ms, achieving a

Method	Backbone	Dim	AIR	CUB	MIT	DTD	# of para.	FLOPs
BCNN (Lin, RoyChowdhury, and Maji 2015)	VGG16	131K	86.9	84.0	–	–	0	411M
Improved BCNN (Lin and Maji 2017)	VGG16	131K	88.5	85.8	–	–	0	411M
RUN (Yu, Cai, and Li 2020)	VGG16	131K	89.8	86.3	80.8	68.4	0	411M
DeepKSPD (Engin et al. 2018)	VGG16	131K	91.0	86.5	81.0	–	0	411M
ReDro (Rahman et al. 2020)	VGG16	33K	89.1	86.5	80.5	–	0	103M
iSQRT-COV (Li et al. 2018)	VGG16	32K	90.0	87.2	–	–	131K	308M
HBP (Yu et al. 2018)	VGG16	24.6K	90.3	87.2	–	–	25.2M	39B
CBP-RM (Gao et al. 2016)	VGG16	8.2K	–	84.0	76.2	64.5	8.4M	13B
LRBP (Kong and Fowlkes 2017)	VGG16	10K	87.3	84.2	–	65.8	51K	161M
MoNet-2 (Gou et al. 2018)	VGG16	10K	86.7	85.7	–	–	10M	16B
CBP-RM + RUN (Yu, Cai, and Li 2020)	VGG16	10K	91.0	85.7	80.5	67.3	10M	16B
TKPF (ours)	VGG16	9.2K	91.4	86.0	80.5	68.2	192	38.5M
DBTNet-50 (Zheng et al. 2019)	DBTNet	2K	91.2	87.5	–	–	–	–
iSQRT-COV (Li et al. 2018)	ResNet50	32K	90.0	88.1	–	–	131K	308M
ReDro (Rahman et al. 2020)	ResNet50	132K	85.4	86.2	84.0	–	2.1M	925M
TKPF (ours)	ResNet50	16K	92.1	85.7	84.1	71.4	4096	38.5M

Table 8: Comparison with state-of-the-art methods. M denotes million and B denotes billion.

$53\times$ speed-up ratio. Reducing 80ms-160ms latency can considerably improve the user experience in a real-time system. Since our TKPF does not influence other layers except the BP layer, the latency in other layers keeps unchanged. Thus, as shown in Table 7, the speed-up ratio in the total inference time is not as significant as that in the BP layer.

Comparisons with SOTA methods

VGG16 backbone. We first compare with full bilinear pooling methods, BCNN (Lin, RoyChowdhury, and Maji 2015), improved BCNN (Lin and Maji 2017) and RUN (Yu, Cai, and Li 2020). As shown in Table 8, the dimension of the full bilinear features is high, 131K. The high dimension limits its efficiency in subsequent classification and retrieval. Our TKPF achieves comparable accuracy with them, but the dimension of compact features from TKPF is only 9.2K.

Then we further compare with medium-scale bilinear features, ReDro (Rahman et al. 2020) and iSQRT-COV (Li et al. 2018). ReDro (Rahman et al. 2020) split 512 channels of local features into four groups and conduct bilinear pooling within each group. Thus, it reduces the dimension of bilinear features to 33K. As shown in Table 8, using more compact features with less FLOPs, our TKPF achieves comparable accuracy with ReDro. In parallel, iSQRT-COV (Li et al. 2018) reduces the dimension of local feature by a 1×1 convolution layer from 512 to 256. Consequently, it reduces the dimension of the bilinear feature from 131K to 32K. The 1×1 convolution layer is equivalent to PCA baseline in Table 4. As shown in Table 8, iSQRT-COV (Li et al. 2018) outperforms TKPF on CUB dataset. But the better performance of iSQRT-COV might be attributed to being pre-trained on a large-scale ImageNet dataset. In contrast, the proposed TKPF is only fine-tuned on the target small-scale datasets. Without pre-training, the 1×1 convolution (PCA baseline) does not perform well as our TKPF, as shown in Table 4.

After that, we compare with a group of compact bilinear pooling methods, including HBP (Yu et al. 2018), CBP-RM (Gao et al. 2016), LRBP (Kong and Fowlkes 2017), MoNet-2 (Gou et al. 2018) and CBP-RM+RUN (Yu, Cai, and Li 2020). As shown in Table 8, our TKPF achieves higher accuracy than CBP-RM (Gao et al. 2016), LRBP (Kong and Fowlkes 2017). Meanwhile, TKPF with higher efficiency achieves a comparable accuracy with MoNet-2 (Gou et al. 2018) and CBP-RM+RUN (Yu, Cai, and Li 2020). Note that HBP (Yu et al. 2018) achieves higher accuracy than TKPF. The higher accuracy of HBP is owing to fusing features from different layers.

ResNet50 backbone. As shown in Table 8, we achieve comparable accuracy with iSQRT-COV (Li et al. 2018) and ReDro (Rahman et al. 2020) using more compact bilinear features with less number of parameters and FLOPs. iSQRT-COV (Li et al. 2018) pre-trained on ImageNet dataset achieves higher accuracy than our TKPF on CUB dataset. We can also boost TKPF through pre-training on ImageNet, but it is not the focus of this paper. We also compare with DBTNet-50 (Zheng et al. 2019) which upgrades blocks in the backbone. It achieves higher accuracy than ours in CUB dataset with a more compact feature. It is worth noting that the focus of TKPF is not improving the architecture of the backbone for extracting more effective local features. Instead, we focus on the last pooling layer.

Conclusion

In this paper, we propose a fast and memory-efficient compact bilinear network. It factorizes the huge-scale projection matrix into a two-level Kronecker product of several small-scale matrices. Benefited from factorization based on Kronecker product, the number of parameters is significantly reduced, making the network memory-efficient. Meanwhile, by utilizing the vec trick and tensor modal product, the compact bilinear feature is obtained in a very fast manner. Systematic experiments on four public benchmarks using two widely-used backbones demonstrate the effectiveness and efficiency in both computation and memory of the proposed method in fine-grained recognition.

References

Cherian, A.; Koniusz, P.; and Gould, S. 2017. Higher-Order Pooling of CNN Features via Kernel Linearization for Action Recognition. In *Proceedings of the 2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 130–138. Santa Rosa, CA.

Cimpoi, M.; Maji, S.; Kokkinos, I.; Mohamed, S.; and Vedaldi, A. 2014. Describing Textures in the Wild. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3606–3613. Columbus, OH.

Cui, Y.; Zhou, F.; Wang, J.; Liu, X.; Lin, Y.; and Belongie, S. J. 2017. Kernel Pooling for Convolutional Neural Networks. In *Proceedingts of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3049–3058. Honolulu, HI.

Dai, X.; Ng, J. Y.; and Davis, L. S. 2017. FASON: First and Second Order Information Fusion Network for Texture Recognition. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6100–6108. Honolulu, HI.

Engin, M.; Wang, L.; Zhou, L.; and Liu, X. 2018. DeepKSPD: Learning Kernel-Matrix-Based SPD Representation For Fine-Grained Image Recognition. In *Proceedings of the 15th European Conference on Computer Vision (ECCV), Part II*, 629–645. Munich, Germany.

Gao, Y.; Beijbom, O.; Zhang, N.; and Darrell, T. 2016. Compact Bilinear Pooling. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 317–326. Las Vegas, NV.

Gou, M.; Xiong, F.; Camps, O. I.; and Sznaier, M. 2018. MoNet: Moments Embedding Network. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3175–3183. Salt Lake City, UT.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. Las Vegas, NV.

Higham, N. J. 2008. *Functions of matrices - theory and computation*. SIAM.

Ionescu, C.; Vantzos, O.; and Sminchisescu, C. 2015. Matrix Backpropagation for Deep Networks with Structured Layers. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 2965–2973. Santiago, Chile.

Jégou, H.; and Zisserman, A. 2014. Triangulation Embedding and Democratic Aggregation for Image Search. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3310–3317. Columbus, OH.

Kar, P.; and Karnick, H. 2012. Random Feature Maps for Dot Product Kernels. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 22, 583–591. La Palma, Canary Islands, Spain.

Kim, J.; On, K. W.; Lim, W.; Kim, J.; Ha, J.; and Zhang, B. 2017. Hadamard Product for Low-rank Bilinear Pooling. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. Toulon, France.

Kong, S.; and Fowlkes, C. C. 2017. Low-Rank Bilinear Pooling for Fine-Grained Classification. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 7025–7034. Honolulu, HI.

Koniusz, P.; Yan, F.; Gosselin, P.; and Mikolajczyk, K. 2017. Higher-Order Occurrence Pooling for Bags-of-Words: Visual Concept Detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(2): 313–326.

Koniusz, P.; Zhang, H.; and Porikli, F. 2018. A Deeper Look at Power Normalizations. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 5774–5783. Salt Lake City, UT.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, 1106–1114. Lake Tahoe, NV.

Li, P.; Xie, J.; Wang, Q.; and Gao, Z. 2018. Towards Faster Training of Global Covariance Pooling Networks by Iterative Matrix Square Root Normalization. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 947–955. Salt Lake City, UT.

Li, P.; Xie, J.; Wang, Q.; and Zuo, W. 2017a. Is Second-Order Information Helpful for Large-Scale Visual Recognition? In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2089–2097. Venice, Italy.

Li, Y.; Wang, N.; Liu, J.; and Hou, X. 2017b. Factorized Bilinear Models for Image Recognition. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2098–2106. Venice, Italy.

Lin, T.; and Maji, S. 2017. Improved Bilinear Pooling with CNNs. In *Proceedings of the British Machine Vision Conference (BMVC)*. London, UK.

Lin, T.; Maji, S.; and Koniusz, P. 2018. Second-Order Democratic Aggregation. In *Proceedings of the 15th European Conference on Computer Vision (ECCV), Part III*, 639–656. Munich, Germany.

Lin, T.; RoyChowdhury, A.; and Maji, S. 2015. Bilinear CNN Models for Fine-Grained Visual Recognition. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 1449–1457. Santiago, Chile.

Maji, S.; Rahtu, E.; Kannala, J.; Blaschko, M.; and Vedaldi, A. 2013. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*.

Pham, N.; and Pagh, R. 2013. Fast and scalable polynomial kernels via explicit feature maps. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 239–247. Chicago, IL.

Quattoni, A.; and Torralba, A. 2009. Recognizing indoor scenes. In *Proceedings of the 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 413–420. Miami, FL.

Rahman, S.; Wang, L.; Sun, C.; and Zhou, L. 2020. RedDro: Efficiently Learning Large-Sized SPD Visual Representation. In *Proceedings of the 16th European Conference on Computer Vision (ECCV), Part XV*, 1–17. Glasgow, UK.

Simonyan, K.; and Zisserman, A. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. San Diego, CA.

Tenenbaum, J. B.; and Freeman, W. T. 2000. Separating Style and Content with Bilinear Models. *Neural Comput.*, 12(6): 1247–1283.

Van Loan, C. F.; and Golub, G. H. 2012. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 4th edition.

Wah, C.; Branson, S.; Welinder, P.; Perona, P.; and Belongie, S. 2011. The caltech-ucsd birds-200-2011 dataset.

Wang, L.; Zhang, J.; Zhou, L.; Tang, C.; and Li, W. 2015. Beyond Covariance: Feature Representation with Nonlinear Kernel Matrices. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 4570–4578. Santiago, Chile.

Wang, Q.; Li, P.; and Zhang, L. 2017. G2DeNet: Global Gaussian Distribution Embedding Network and Its Application to Visual Recognition. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6507–6516. Honolulu, HI.

Wang, Q.; Li, P.; Zuo, W.; and Zhang, L. 2016. RAID-G: Robust Estimation of Approximate Infinite Dimensional Gaussian with Application to Material Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 4433–4441. Las Vegas, NV.

Wang, Q.; Xie, J.; Zuo, W.; Zhang, L.; and Li, P. 2021. Deep CNNs Meet Global Covariance Pooling: Better Representation and Generalization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 43(8): 2582–2597.

Wei, X.; Zhang, Y.; Gong, Y.; Zhang, J.; and Zheng, N. 2018. Grassmann Pooling as Compact Homogeneous Bilinear Pooling for Fine-Grained Visual Classification. In *Proceedings of the 15th European Conference on Computer Vision (ECCV), Part III*, 365–380. Munich, Germany.

Yu, C.; Zhao, X.; Zheng, Q.; Zhang, P.; and You, X. 2018. Hierarchical Bilinear Pooling for Fine-Grained Visual Recognition. In *Proceedings of the 15th European Conference on Computer Vision (ECCV), Part XVI*, 595–610. Munich, Germany.

Yu, T.; Cai, Y.; and Li, P. 2020. Toward Faster and Simpler Matrix Normalization via Rank-1 Update. In *Proceedings of the 16th European Conference on Computer Vision (ECCV), Part XIX*, 203–219. Glasgow, UK.

Yu, T.; Li, X.; and Li, P. 2021. Fast and compact bilinear pooling by shifted random maclaurin. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 3243–3251.

Zhang, J.; Wang, L.; Zhou, L.; and Li, W. 2021. Beyond Covariance: SICE and Kernel Based Visual Feature Representation. *Int. J. Comput. Vis.*, 129(2): 300–320.

Zheng, H.; Fu, J.; Zha, Z.; and Luo, J. 2019. Learning Deep Bilinear Transformation for Fine-grained Image Representation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 4279–4288. Vancouver, Canada.