FAST BILINEAR MATRIX NORMALIZATION VIA RANK-1 UPDATE

Anonymous authors
Paper under double-blind review

ABSTRACT

Bilinear pooling has achieved an impressive improvement over classical average and max pooling in many computer vision tasks. Recent studies discover that matrix normalization is vital for improving the performance of bilinear pooling since it effectively suppresses the burstiness. Nevertheless, exiting matrix normalization methods such as matrix square-root and logarithm are based on singular value decomposition (SVD), which is not well suited in the GPU platform, limiting its efficiency in training and inference. To boost the efficiency in the GPU platform, recent methods rely on Newton-Schulz (NS) iteration to approximate the matrix square-root. Despite that NS iteration is well supported by GPU, it takes $\mathcal{O}(KD^3)$ computation complexity, where D is the dimension of each local feature and K is the number of iterations, which is still expensive. Meanwhile, NS iteration is applicable only to full bilinear matrix. In contrast, a compact bilinear feature obtained from tensor sketch or random projection has broken the matrix structure, cannot be normalized by NS iteration. To overcome these limitations, we propose a rank-1 update normalization (RUN), which reduces the computational cost from $\mathcal{O}(KD^3)$ to $\mathcal{O}(KDN)$, where N is the number of local features per image. Moreover, it supports the normalization on compact bilinear features. Meanwhile, the proposed RUN is differentiable, and thus it is feasible to plug it in a convolutional neural network as a layer to support an end-to-end training. Comprehensive experiments on four public benchmarks show that, for the full bilinear pooling, the proposed RUN achieves comparable accuracies with a $330\times$ speedup over NS iteration. For the compact bilinear pooling, our RUN achieves comparable accuracies with a $5400\times$ speedup over the SVD-based normalization.

1 INTRODUCTION

In the past decade, convolutional neural network (CNN) has achieved a great success in many computer vision tasks ranging from image recognition (He et al. (2016)), object detection (Ren et al. (2015)), semantic segmentation (Long et al. (2015)) to action recognition (Simonyan & Zisserman (2014a)). Despite CNN architecture has evolved significantly, it still inherits the basic architecture from the pioneering work, AlexNet (Krizhevsky et al. (2012)). To be specific, it consists of three parts: a feature extractor, an aggregation module and a classifier, as visualized in Figure 1. The feature extractor normally consists of a series of convolution, pooling, batch normalization and non-linear rectification layers. It generates a feature map $\mathcal{F}$ of $W \times H \times D$ size, where W and H are the width and height of the feature map and D is the depth of the feature map, *i.e.*, the number of channels. To enhance the performance of a CNN, many efforts have been devoted to boosting effectiveness of the feature extractor. For instance, GoogLeNet (Szegedy et al. (2014)) proposes an Inception module, which fuses feature maps from different scales and encodes richer visual information than the vanilla CNN. ResNet (He et al. (2016)) adopts a residual architecture based identity mapping, which overcomes the performance degeneration as network goes deep. It has achieved record-breaking performance in many tasks. DenseNet (Huang et al. (2017)) extends the residual module to a densely-connected module, achieving a better performance.

The aggregation module converts the feature map $\mathcal{F}$ generated by the feature extractor into a holistic feature vector $\mathbf{f} \in \mathbb{R}^d$. The early work such as AlexNet and VGGNet (Simonyan & Zisserman (2014b)) implements the aggregate module by a fully-connected layer. To be specific, they unfold the three-dimensional feature map $\mathcal{F}$ into one dimension vector $\mathrm{vec}(\mathcal{F}) \in \mathbb{R}^{WHD}$ and obtain a

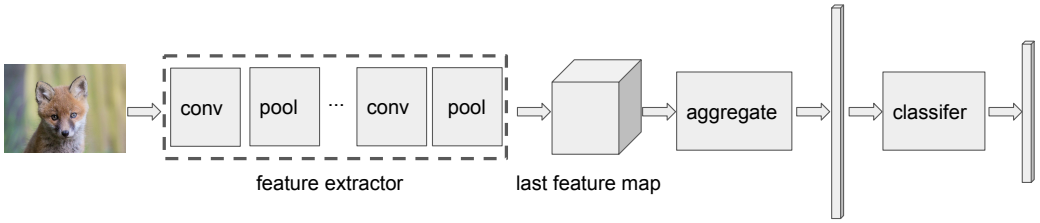
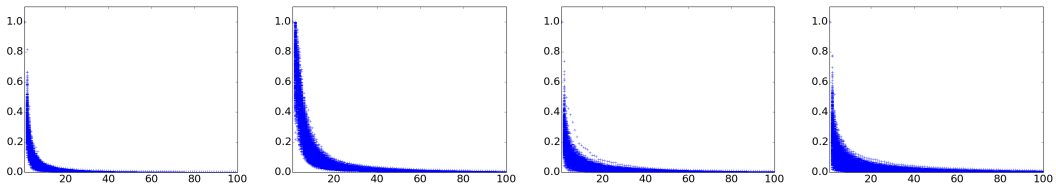


Figure 1: The basic architecture of a convolutional neural network (CNN). It consists of tree parts, a feature extractor, an aggregate module and a classifier.



(a) CUB (b) Airplane (c) MIT (d) DTD

Figure 2: The scatter plots of singular values of bilinear matrices on some typical computer vision datasets: CUB (Welinder et al. (2010)), Airplane (Maji et al. (2013)), MIT (Quattoni & Torralba (2009)) and DTD (Cimpoi et al. (2014)). The indices of singular values are along x axis and scaled magnitudes of singular values are on y axis. For the ease of illustration, each magnitude is divided by its corresponding largest singular value and the scaled magnitudes are in the range of $[0, 1]$. We plot the first 100 singular values of all samples on each dataset.

global vector $\mathbf{f} = \mathbf{W}\text{vec}(\mathcal{F}) + \mathbf{b} \in \mathbb{R}^d$, where $\mathbf{W} \in \mathbb{R}^{d \times WHD}$ and $\mathbf{b} \in \mathbb{R}^d$ are parameters of a fully-connected layer. In contrast, some more advanced architectures such as Inception and ResNet implement the aggregation module by a global average pooling, where they conduct average pooling along the width and height dimensions, and generate a holistic vector $\mathbf{f} \in \mathbb{R}^D$. Compared with a fully-connected layer, the global average pooling is more robust to spatial transforms. To obtain a more effective holistic feature, NetVLAD (Arandjelovic et al. (2016)) incorporates VLAD (Jegou et al. (2011)) into the convolutional neural network. Similarly, Miech et al. (2017) integrates the Fisher vector into the neural network and proposes a learnable pooling. In parallel, bilinear convolutional neural network (Lin et al. (2015)) implements the aggregation module by a bilinear pooling operation, which encodes the second-order information. It achieves a better performance than average pooling and max pooling in many computer vision tasks, such as fine-grained recognition (Lin & Maji (2017)), generic image recognition (Li et al. (2018)) and video classification (Wang et al. (2017b)).

Method	Algorithm	Complexity	GPU Support	CBP Support
O^2P	SVD	$\mathcal{O}(D^3)$	limited	No
G^2DeNet	SVD	$\mathcal{O}(D^3)$	limited	No
MPN-COV	Eigen Decomp	$\mathcal{O}(D^3)$	limited	No
Improved B-CNN	Newton-Schulz	$\mathcal{O}(D^3)$	good in FP	No
iQRT-COV	Newton-Schulz	$\mathcal{O}(KD^3)$	good	No
MoNet	SVD	$\mathcal{O}(D^3)$	limited	Yes
RUN (Ours)	Power Method	$\mathcal{O}(KDN)$	good	Yes

Table 1: Differences between the proposed RUN with O^2P (Ionescu et al. (2015)), G^2DeNet (Wang et al. (2017a)), MPN-COV (Lin & Maji (2017)), Improved B-CNN (Lin & Maji (2017)), iQRT-COV (Li et al. (2018)) and MoNet (Gou et al. (2018)). Here, K is the number of iterations, D is the local feature dimension and N is the number of local features. Our RUN takes a low computation complexity $\mathcal{O}(KDN)$, is well supported by GPU and well supports compact bilinear pooling (CBP).

The burstiness (Jégou et al. (2009)) in computer vision was first discussed in the context of the bag-of-word model. It points out the phenomenon that most regions of an image are assigned to the same visual word. In such case, the representation of the image is determined by a single visual word, however, some low-frequency visual words are ignore which might be important. To obtain a more effective feature, Perronnin *et al.* (Perronnin et al. (2010)) conduct the element-wise square-root normalization to balance contributions of different visual words. In the context of bilinear features, singular vectors correspond to visual words, the burstiness corresponds to the first a few singular values that are significantly larger than the remaining singular values, as shown in Figure 2. To alleviate this problem, a straightforward way is to normalize singular values. Recent studies (Li et al. (2017); Lin & Maji (2017)) show that, the normalization on singular values is vital for achieving high recognition performance. Existing normalization methods such as matrix square-root normalization (Li et al. (2017)) and matrix logarithm normalization (Ionescu et al. (2015)) rely on the singular value decomposition (SVD). But SVD is not easily parallelizable and hence not well suited in the parallel GPU platform. To boost the efficiency in the GPU platform, improved B-CNN (Lin & Maji (2017)) and i-SQRT (Li et al. (2018)) attempts to approximate the matrix square root via the Newton-Schulz (NS) iteration (Higham (2008)). Since NS iteration only needs matrix-matrix product, it is easily parallelizable and well suited in the GPU platform. The NS iteration has a computation complexity of $\mathcal{O}(D^3K)$, where D is the number of channels of the last features map and K is the number of iterations. Since D can be very large, NS iteration is still expensive. In addition, NS iteration is conducted on the bilinear matrix and cannot normalize compact bilinear features (Gao et al. (2016)) from tensor sketch or random projection.

To speed up the matrix normalization, we propose a rank-1 update normalization (RUN). The proposed RUN is an iterative algorithm inspired by power method. In each iteration, it only needs two matrix-vector multiplications, which takes low computation cost and is easily parallelizable and well suited in the GPU platform. In total, the computation complexity of the proposed RUN is $\mathcal{O}(KDN)$. Here, $N = WH$ is the number of local features per image, which is in a comparable scale with D, K is the iteration number, which is set as 2 by default on all testing datasets. Therefore, the complexity of our RUN is considerably lower than the $\mathcal{O}(D^3K)$ complexity used in NS iteration. Moreover, our RUN supports the normalization on a compact bilinear feature generated from tensor sketch or random projection. In addition, the proposed RUN is differentiable, as a result, we can easily plug it into a neural network to support an end-to-end training. Experiments on four public benchmarks show the effectiveness and efficiency of our method. Table 1 summarizes differences between our method and other related work.

2 Matrix Normalization and Compact Bilinear Pooling

Given a feature map $\mathcal{F}$, bilinear pooling reshapes $\mathcal{F}$ into a two-dimensional matrix $\mathbf{F} \in \mathbb{R}^{WH \times D}$ and calculates the bilinear matrix by $\mathbf{B} = \mathbf{F}^\top \mathbf{F}$. B-CNN (Lin et al. (2015)) implements the bilinear pooling as a layer of a convolutional neural network to support an end-to-end training. It achieves a better performance on fine-grained classification than standard AlexNet with a fully-connected layer as aggregation module. The research on B-CNN proceeds along two main directions: 1) improve the effectiveness of bilinear pooling through matrix normalization (Lin & Maji (2017); Li et al. (2017)); 2) improve the efficiency of bilinear feature through compact bilinear pooling (Gao et al. (2016); Cui et al. (2017)). Our work is related with both directions since we propose a fast matrix normalization method, and make it compatible with compact bilinear pooling. Below we review these two directions, respectively.

2.1 Matrix Normalization

There are two popular matrix normalization methods, the matrix square-root normalization used in Improved B-CNN (Lin & Maji (2017)) and the matrix logarithm normalization used in O^2P (Ionescu et al. (2015)). They first conduct singular value decomposition (SVD) on the bilinear matrix $\mathbf{B}$ by

$$\mathbf{B} \rightarrow \mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top.$$

Then they conduct normalization on singular values and obtain the normalized bilinear feature by

$$\hat{\mathbf{B}} \leftarrow \mathbf{U}\text{g}(\mathbf{\Sigma})\mathbf{U}^\top,$$

where $\text{g}(\mathbf{\Sigma})$ is conducted on singular values in an element-wise manner. Matrix square-root normalization adopts $\text{g}(\mathbf{\Sigma}) = \mathbf{\Sigma}^{1/2}$ and matrix logarithm normalization adopts $\text{g}(\mathbf{\Sigma}) = \log(\mathbf{\Sigma})$. However,

as mentioned before, SVD is not easily parallelizable and not well supported in the GPU platform, limiting its efficiency in training and inference. Improved B-CNN (Lin & Maji (2017)) and i-SQRT (Li et al. (2018)) utilize Newton-Schulz (NS) iteration to approximate the matrix square root. Given a bilinear matrix $\mathbf{B}$, NS initializes $\mathbf{Y}_0 = \mathbf{B}$ and $\mathbf{Z}_0 = \mathbf{I}$. For each iteration, $\mathbf{Z}_k$ and $\mathbf{Y}_k$ is updated by

$$\mathbf{Y}_k = \frac{1}{2}\mathbf{Y}_{k-1}(3\mathbf{I} - \mathbf{Z}_{k-1}\mathbf{Y}_{k-1}),$$
$$\mathbf{Z}_k = \frac{1}{2}(3\mathbf{I} - \mathbf{Z}_{k-1}\mathbf{Y}_{k-1})\mathbf{Z}_{k-1},$$

where $\mathbf{Y}_k$ converges to $\mathbf{B}^{1/2}$. Since it involves only matrix-matrix product, it is easily parallelizable and well supported in the GPU platform. The computation complexity of each iteration is $\mathcal{O}(D^3)$, where D is the local feature dimension. Since D is large, computing Newton-Schulz (NS) iteration is still expensive. In contrast, our method is based on iterations of matrix-vector multiplications, which are computationally cheaper than the matrix-matrix multiplications used in NS iteration. What's more, we will show in next subsection that, the NS iteration is not compatible with existing compact bilinear pooling methods, whereas ours readily supports normalization on compact bilinear features.

2.2 Compact Bilinear Pooling

The dimension of a bilinear feature is $D \times D$, which is extremely high. On one hand, it is more prone to over-fitting due to huge number of model parameters in the classifier, especially in the few-shot learning scenario. On the other hand, in the retrieval application, it is extremely expensive to store and compare high-dimensional bilinear features. To overcome these drawbacks, CBP (Gao et al. (2016)) is proposed. It treats the outer product used in bilinear pooling as a kernel embedding, and seek to approximate the explicit kernel feature map. To be specific, by rearranging the feature map $\mathcal{F}$ to $\mathbf{F} = [\mathbf{f}_1, \cdots, \mathbf{f}_{WH}]^\top$, the bilinear matrix $\mathbf{B}$ is obtained by

$$\mathbf{B} = \mathbf{F}^\top\mathbf{F} = \sum_{i=1}^{WH} \mathbf{f}_i\mathbf{f}_i^\top = \sum_{i=1}^{WH} \mathrm{h}(\mathbf{f}_i), \tag{1}$$

where $\mathrm{h}(\mathbf{f}_i) = \mathbf{f}_i\mathbf{f}_i^\top \in \mathbb{R}^{D\times D}$ is the explicit feature map of the polynomial kernel. CBP seeks for a low-dimensional projection function $\phi(\mathbf{f}_i) \in \mathbb{R}^d$ with $d \ll D^2$ such that

$$\langle \phi(\mathbf{x}), \phi(\mathbf{y})\rangle \approx \langle \mathrm{vec}(\mathrm{h}(\mathbf{x})), \mathrm{vec}(\mathrm{h}(\mathbf{y}))\rangle, \tag{2}$$

where $\mathrm{vec}(\cdot)$ is the operation to unfold the 2D matrix to 1D vector. In this case, the approximated low-dimensional bilinear feature is obtained by $\tilde{\mathbf{B}} = \sum_{i=1}^{WH} \phi(\mathbf{f}_i)$.

CBP investigates two types of approximation methods: Random Maclaurin (Kar & Karnick (2012)) and Tensor Sketch Pham & Pagh (2013), which are given in **Algorithm** 1 and **Algorithm** 2. Since the compact bilinear feature $\tilde{\mathbf{B}}$ has broken the matrix structure, the matrix normalization methods conducted on the bilinear feature $\mathbf{B}$, such as Newton-Schulz iteration, is no longer feasible for normalizing $\tilde{\mathbf{B}}$. To tackle this, MoNet (Gou et al. (2018)) conducts SVD directly on the original feature $\mathbf{F}$ instead of $\mathbf{B}$ and then conducts compact bilinear pooling. Nevertheless, as we mentioned, the SVD is not well supported on GPU platform, limiting the training and inference efficiency. In contrast, we will see in the next section that our method only relies on matrix-vector multiplications, and hence is easily parallelizable and well supported in the GPU platform and supports the normalization on a compact bilinear feature generated from tensor sketch or random projection.

3 Rank-1 Update Normalization (RUN)

To overcome the limitations of previous methods, we propose a rank-1 update normalization (RUN). Below we give the motivation of the proposed RUN and then summarize it in Algorithm 3.

Assuming that, through SVD, the bilinear feature $\mathbf{B}$ can be decomposed into $\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top$, where $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_D]$ is orthogonal and $\mathbf{\Sigma} = \mathrm{diag}([\sigma_1, \cdots, \sigma_d])$ is diagonal with $\sigma_1 \geq \sigma_1 \geq \sigma_D$. First we initialize a random vector $\mathbf{v}_0 = [v_1, ..., v_D] \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. That is $\{v_i\}_{i=1}^D$ are *i.i.d.* random variables with standard normal distribution. We perform K steps of power method as follows:

$$\mathbf{v}_k = \mathbf{B}\mathbf{v}_{k-1}, \text{ for } k = 1, \ldots, K. \tag{3}$$

Algorithm 1 Tensor Sketch

Input: $\mathbf{x} \in \mathbb{R}^d$
Output: $\phi_{TS}(\mathbf{x}) \in \mathbb{R}^D$

1: Generate random vectors $\mathbf{h}_1, \mathbf{h}_2 \in \mathbb{N}^c$ and $\mathbf{s}_1, \mathbf{s}_2 \in \{+1, -1\}^c$. $\mathbf{h}_1(i)$ and $\mathbf{h}_2(i)$ are uniformly sampled from $\{1, 2, \cdots, D\}$, $\mathbf{s}_1(i)$ and $\mathbf{s}_2(i)$ are uniformly sampled from $\{+1, -1\}$.
2: Sketch $\Psi(x, h, s) = \{(Q\mathbf{x})_1, \cdots, (Q\mathbf{x})_D\}$, where $Q(\mathbf{x})_j = \sum_{t:\mathbf{h}(t)=j} \mathbf{s}(t)\mathbf{x}(t)$.
3: Compute $\phi_{TS}(\mathbf{x}) = \mathrm{FFT}^{-1}(\mathrm{FFT}(\Psi(\mathbf{x}, \mathbf{h}_1, \mathbf{s}_1)) \odot \mathrm{FFT}(\Psi(\mathbf{x}, \mathbf{h}_2, \mathbf{s}_2)))$, where $\odot$ denotes element-wise multiplication.
4: **return** $\phi_{TS}(\mathbf{x})$

Algorithm 2 Random Maclaurin

Input: $\mathbf{x} \in \mathbb{R}^d$
Output: $\phi_{RM}(\mathbf{x}) \in \mathbb{R}^D$

1: Generate random matrices $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d \times D}$ with each entry 1 or -1 with equal probability.
2: $\phi_{RM}(\mathbf{x}) \leftarrow \frac{1}{\sqrt{D}}(\mathbf{W}_1\mathbf{x}) \odot (\mathbf{W}_2\mathbf{x})$.
3: **return** $\phi_{RM}(\mathbf{x})$

Then the rank-1 matrix is constructed by

$$\mathbf{R}_K = \mathbf{B}\mathbf{v}_K\mathbf{v}_K^\top / \|\mathbf{v}_K\|_2^2 \tag{4}$$

After that, we update the matrix $\mathbf{B}$ by subtracting $\mathbf{R}_K$:

$$\mathbf{B}_K = \mathbf{B} - \epsilon\mathbf{R}_K, \tag{5}$$

where $\epsilon \in [0, 1]$ is a small constant. The classic convergence result of power method tells that if $\sigma_1 > \sigma_2$, $\mathbf{v}_K$ will converge to $\mathbf{u}_1$ directionally. Therefore, $\mathbf{B}_K$ converges to $\mathbf{B} - \epsilon\sigma_1\mathbf{u}_1\mathbf{u}_1^\top$. That is

$$\lim_{K\to\infty} \mathbf{B}_K = \mathbf{U}\,\mathrm{diag}([\sigma_1(1-\epsilon), \sigma_2, \ldots, \sigma_D])\mathbf{U}^\top, \tag{6}$$

i.e., the eigenvalues of $\mathbf{B}_\infty$ remain unchanged except the largest one, which is decreased by $\epsilon\sigma_1$. More generally, $\mathbf{B}_K$ is an estimation of a normalized bilinear matrix. To be specific, it satisfies following theorem:

Theorem 1 *Let $\mathbf{B}_K$ be obtained via Eq. (3)-(5), where $\mathbf{v}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Then the expectation of $\mathbf{B}_K$ is given by*

$$\mathbb{E}(\mathbf{B}_K) = \mathbf{U}\,\mathrm{diag}([\sigma_1(1-\epsilon\alpha_1), \cdots, \sigma_D(1-\epsilon\alpha_D))\mathbf{U}^\top, \tag{7}$$

where $1 \geq \alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_D$.

Due to limitation of the space, the proof of the Theorem 1 is given in Appendix A. The operation in the right-hand side of Eq. (7) scales each singular value σ_i by $(1 - \epsilon\alpha_i)$. As $1 \geq \alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_D$ and $\epsilon \in [0, 1]$, thus

$$0 \leq 1 - \epsilon\alpha_1 \leq 1 - \epsilon\alpha_2 \leq \cdots \leq 1 - \epsilon\alpha_D \leq 1. \tag{8}$$

It gives a smaller scale factor to a larger singular value, making singular values more balanced.

Since computing $\mathbf{B}_K$ only requires K times of matrix-vector multiplications, it only takes $\mathcal{O}(KD^2)$ complexity and is well supported in GPU platform. In experiments section, we will show when K is small, *e.g.*, $K = 2$, it has achieved excellent performance. Nevertheless, obtaining the above approximated normalized bilinear feature $\mathbf{B}_K$ requires the original bilinear matrix $\mathbf{B}$ obtained from bilinear pooling. Thus, it is not applicable to the compact bilinear feature which has broken the structure of square matrix. To make the proposed fast matrix normalization method compatible with compact bilinear pooling, we seek to directly conduct normalization on the original feature map $\mathbf{F} \in \mathbb{R}^{N \times D}$, where $N = WH$ is the number of local features and D is the local feature dimension. It is based on following iterations:

$$\mathbf{v}_k = \mathbf{F}^\top\mathbf{F}\mathbf{v}_{k-1}, \text{for } k = 1, \ldots, K, \tag{9}$$

Algorithm 3 Rank-1 Update Normalization (RUN)

Input: Local features $\mathbf{F} \in \mathbb{R}^{N\times D}, \eta, K$
Output: Normalized local features $\mathbf{F}_K$.
1: Generate $\mathbf{v}_0 = [v_1, ..., v_D] \in \mathbb{R}^D$, where $\{v_i\}_{i=1}^D$ are *i.i.d.* random variables with normal distribution.
2: **for** $k \in [1, K]$ **do**
3: $\quad \mathbf{v}_k = \mathbf{F}^\top \mathbf{F}\mathbf{v}_{k-1}$
4: $\mathbf{F}_K = \mathbf{F} - \eta \frac{\mathbf{F}\mathbf{v}_K \mathbf{v}_K^\top}{\|\mathbf{v}_K\|_2^2}$
5: **return** $\mathbf{F}_K$.

where the entries of $\mathbf{v}_0$ are *i.i.d.* random variables with standard normal distribution. Then we construct the updated feature map $\mathbf{F}_K$ by

$$\mathbf{F}_K = \mathbf{F} - \eta \mathbf{F}\mathbf{v}_K\mathbf{v}_K^\top / \|\mathbf{v}_K\|_2^2, \tag{10}$$

where $\mathbf{v}_K$ are obtained via equation 9 and $\eta \in (0, 1]$ is a constant. The above procedure is summarized in Algorithm 3. Since in each iteration, it only needs two matrix-vector multiplications, in total, the computational complexity of obtaining $\mathbf{F}_K$ is $\mathcal{O}(KDN)$. Let $\mathbf{u}_{F,i}$ and $\mathbf{v}_{F,i}$ be the left and right singular vectors of $\mathbf{F}$ corresponding with its ith largest singular value $\sigma_{F,i}$. If $\sigma_{F,1} \neq \sigma_{F,2}$, $\mathbf{F}\mathbf{v}_\mathbf{k}$ and $\mathbf{v}_\mathbf{k}$ will converge directionally to $\mathbf{u}_{F,1}$ and $\mathbf{v}_{F,1}$, respectively. In limit, we have

$$\lim_{K\to\infty} \mathbf{F}_K = \mathbf{F} - \eta\sigma_{F,1}\mathbf{u}_{F,1}\mathbf{v}_{F,1}^\top,$$

whose singular values are the same as that of $\mathbf{F}$, except the largest one, which is decreased by $\eta\sigma_{1,F}$. In fact, similar to Theorem 1, we have

Theorem 2 *Let $\mathbf{F}_K$ be obtained as in Algorithm 3. Then the expectation of $\mathbf{F}_K$ can be given by*

$$\mathbb{E}(\mathbf{F}_K) = \mathbf{U}_F \widehat{\Sigma}_F \mathbf{V}_F^\top, \tag{11}$$

where $\mathbf{U}_F$, $\mathbf{V}_F$ are the left and right singular vector matrices of $\mathbf{F}$, respectively, $\widehat{\Sigma}_F$ is the diagonal matrix $\operatorname{diag}[(\sigma_{F,1}(1-\eta\beta_1), \cdots, \sigma_{F,D}(1-\eta\beta_D))]$ *with* $0 \leq 1-\eta\beta_1 \leq 1-\eta\beta_2 \leq \cdots \leq 1-\eta\beta_D \leq 1$.

Its proof is similar to Theorem 1, and thus we will omit it. Using the standard bilinear pooling, the normalized bilinear matrix feature can be obtained by $\bar{\mathbf{B}}_K = \mathbf{F}_K^\top \mathbf{F}_K$. When $\sigma_{F,1} \neq \sigma_{F,2}$, $\bar{\mathbf{B}}_K$ satisfies

$$\lim_{K\to\infty} \bar{\mathbf{B}}_K = \mathbf{V}_F \operatorname{diag}([\sigma_{F,1}^2(1-\eta)^2, \cdots, \sigma_{F,D}^2])\mathbf{V}_F^\top, \tag{12}$$

Since $\mathbf{V}_F$ in Eq. (12) is equal to $\mathbf{U}$ in Eq. (6), if we set $(1-\epsilon)$ in Eq. (12) equal to $(1-\eta)^2$ in Eq. (6), $\mathbf{B}_K$ and $\bar{\mathbf{B}}_K$ will converge to the same matrix. But the advantage of updating $\mathbf{F}$ as Eq. (10) rather than updating $\mathbf{B}$ as Eq. (5) is that, the former one is compatible with compact bilinear pooling, which can not be achieved by the latter one. In this case, the compact normalized bilinear feature is obtained by

$$\bar{\mathbf{b}}_K = \sum_{i=1}^{N} \phi(\mathbf{F}_K[i, :]), \tag{13}$$

where $\mathbf{F}_K[i, :]$ denotes the i-th row of $\mathbf{F}_K$ and ϕ is implemented by tensor sketch or random Maclaurin, and $\bar{\mathbf{b}}_K \in \mathbb{R}^D$ is the compact and normalized feature where $D \ll d^2$.

The proposed RUN is summarized in Algorithm 3. We implement the proposed RUN as a layer of a CNN. The layer takes the original feature map $\mathbf{F}$ as input and outputs the normalized feature map $\mathbf{F}_K$. In the forward path, $\mathbf{F}_K$ is computed by Eq. (10). Below we derive its backward path. Note that, despite that one can rely on auto-grad tool in existing deep learning framework such as Pytorch and TensorFlow to obtain the backward path, we still derive this process in Appendix B for readers to better understand the proposed algorithm. After obtaining $\mathbf{F}_K$, it is feasible to conduct the original bilinear pooling (BP) or compact bilinear pooling (CBP). Figure 3 illustrates the architecture of the proposed network.

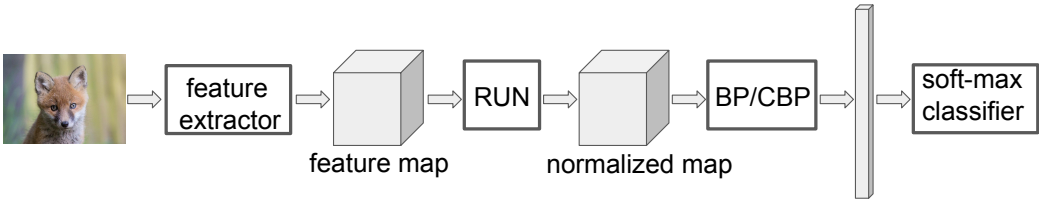


Figure 3: The architecture of the proposed convolutional neural network. RUN denotes the proposed rank-1 update normalization, which takes input the feature map of the last convolutional layer. BP denotes the bilinear pooling and CBP represents compact bilinear pooling.

4 EXPERIMENTS

In this section, we demonstrate the experimental results. We first introduce the testing datasets and implementation details. Then we show conduct ablation study on two scenarios: 1) RUN with standard bilinear pooling and 2) RUN with the compact bilinear pooling. After that, the comparisons with other pooling methods are conducted.

4.1 DATASETS

We conduct experiments on three tasks: 1) fine-grained recognition, 2) scene recognition and 3) texture recognition. On the fine-grained recognition task, experiments are conducted on CUB (Welinder et al. (2010)) and Airplane (Maji et al. (2013)) datasets. On the scene recognition task, experiments are conducted on MIT (Quattoni & Torralba (2009)) dataset. On the texture recognition task, we test our method on DTD (Cimpoi et al. (2014)) dataset. Table 2 gives a summary.

	Fine-grained		Scene	Texture
	CUB	Airplane	MIT	DTD
classes	200	100	67	47
training	5,994	6,667	4,014	1,880
testing	5,794	3,333	1,339	3,760

Table 2: Details of four datasets.

4.2 IMPLEMENTATION DETAILS

We use VGG16 (Simonyan & Zisserman (2014b)) as the backbone network to make a fair comparison with existing methods. After scaling and cropping, the input size of an input image is $448\times48\times3$ and the size of the feature map is $28 \times 28 \times 512$. After we obtain the bilinear feature, we further conduct element-wise signed square-root normalization followed by ℓ_2-normalization as the original BCNN (Lin et al. (2015)). We adopt a two-phase training strategy. In the first phase, we only update the weights of the last fully-connected layer and fix the other layers. The initial learning rate is set as 0.2 on airplane dataset and 1 on other datasets, and it decreases to 0.1 of the current learning rate if the validation error does not drop in continuous 5 epochs. We set weight decay as 10^{-8} in the first phase. The first phase finishes in 50 epochs. In the second phase, we update the weights of all layers and the initial learning rate is set as 0.02 on CUB dataset and 0.01 on other datasets, and it decreases to 0.1 of the current learning rate if the validation error does not drop in continuous 5 epochs. We set weight decay as 10^{-5} in the second phase. The second phase finishes in 40 epochs.

4.3 ABLATION STUDY ON ORIGINAL BILINEAR POOLING

In this section, we test RUN using original bilinear pooling. The feature dimension is $512 * 512 =$ 262K.

Influence of η. η in Eq. (12) controls the strength of suppressing the large singular values. Recall from Eq. (12) that, the normalized feature $\bar{\mathbf{B}}_K$ converges to:

$$\mathbf{V}^F \mathrm{diag}[(1-\eta)^2\sigma_{F,1}^2, \cdots, \sigma_{F,d}^2]\mathbf{V}_F^\top.$$

	CUB	Airplane	MIT	DTD
first epoch	2.23	1.53	2.38	5.09
last epoch	3.68	1.69	4.53	7.40

Table 3: The average $\sigma_{F,1}/\sigma_{F,2}$ on four datasets.

η	CUB	Airplane	MIT	DTD
0.0	84.1	88.9	79.8	65.6
0.1	84.8	89.3	80.6	66.6
0.2	85.3	89.5	**81.0**	67.8
0.4	86.0	89.6	80.5	68.3
0.6	86.3	**89.8**	80.8	**68.7**
0.8	86.2	89.7	80.7	68.4
1.0	**86.4**	**89.8**	80.9	68.3
1.2	86.0	**89.8**	80.9	68.2
1.5	86.2	89.7	80.5	68.3
2.0	83.9	89.0	79.7	65.7

Table 4: The influence of η on the proposed RUN.

K	CUB	Airplane	MIT	DTD
1	85.7	89.7	80.5	**68.7**
2	**86.3**	89.8	**80.8**	68.4
3	86.2	89.8	**80.8**	68.3
5	86.2	**89.9**	80.7	68.4
10	86.1	**89.9**	80.7	68.4

Table 5: The influence of K on the proposed RUN.

From the above equation, we observe that, when $\eta \in (2, +\infty) \cup (-\infty, 0)$, the largest value of the normalized bilinear matrix $\bar{\mathbf{B}}_K$ is even larger than that of the original bilinear matrix $\mathbf{B}$. Hence a good value of η should in the range $[0, 2]$. Ideally, we can select the value of η according to the gap between $\sigma_{F,1}$ and $\sigma_{F,2}$. Since singular values change for different samples or different epochs, we can compute the $\sigma_{F,1}$ and $\sigma_{F,2}$ online for each sample in each epoch. But computing $\sigma_{F,1}$ and $\sigma_{F,2}$ will double the time cost compared with using a manually set η which only needs compute $\sigma_{F,1}$. An alternative solution is to compute the average $\sigma_{F,1}/\sigma_{F,2}$ of all samples using the pre-trained model and then use the average value to guide the choice of the η. But the average value changes in the training process, the average value computed from the pre-trained model might not be effective for the whole training process. Table 3 shows the average $\sigma_{F,1}/\sigma_{F,2}$ of each dataset. Since each experiment lasts for tens of epochs and it is difficult to report the ratio of each epoch, we just report the average $\sigma_{F,1}/\sigma_{F,2}$ in the first epoch and that in the last epoch. From Table 3, we observe that the average $\sigma_{F,1}/\sigma_{F,2}$ in the first epoch is different from that in the last epoch.

We further test the influence of η on the classification accuracy. As shown in Table 5, when $\eta = 0$, *i.e.*, without RUN, the accuracies are not as good as that when $\eta \in [0.4, 1.5]$. Note that, on Airplane dataset, the accuracy drop when $\eta = 0$ is not large, it is in accordance with the small value of $\sigma_{F,1}/\sigma_{F,2}$ on Airplane dataset in Table 3. In contrast, on DTD dataset, the accuracy drop is significant, it is also in accordance with the large value of $\sigma_{F,1}/\sigma_{F,2}$ on DTD dataset in Table 3.

As Table 3 shows that the average value of $\sigma_{F,1}/\sigma_{F,2}$ varies significantly on four datasets, thus we might expect that the optimal η are different on four dataset. Surprisingly, as shown in Table 5, when $\eta \in [0.4, 1.5]$ the performance is stable and not sensitive to the change of η. By default, we set $\eta = 0.6$ on all datasets. Another observation is that, when $\eta = 2.0$, its performance is as bad as that when $\eta = 0.0$. The bad performance when $\eta = 2.0$ is expected since it leads to the condition that $(1 - \eta)^2 = 1$. It is equivalent to removing the matrix normalization.

Influence of K. K in Eq. (10) represents the number of iterations in our RUN. The time cost of the proposed RUN is linear with K. Recall from Eq. (12) that, when K is large, the normalization focuses only on the largest singular value and keeps the others unchanged. In contrast, if K is not large, it also normalizes other large singular values besides the largest one. As shown in Table 5, on

Algorithm	FLOPs	GPU Time	Accuracy			
			CUB	Airplane	MIT	DTD
SVD	1.88G	6731ms	85.8	88.5	80.6	68.4
NS iteration	4.03G	833ms	85.7	89.6	80.5	68.3
power method (ours)	3.2M	2.5ms	86.3	89.8	80.8	68.4

Table 6: Comparisons with SVD-based method (Lin & Maji (2017)) and Newton-Schulz (NS) iteration (Li et al. (2018)).

Dimension	CUB		Airplane		MIT		DTD	
	RM	TS	RM	TS	RM	TS	RM	TS
$1,000$	83.1	83.8	88.9	88.5	78.0	76.1	59.9	63.4
$2,000$	84.6	83.9	89.8	89.3	78.8	78.2	63.6	66.5
$4,000$	84.4	84.8	88.8	90.5	79.9	79.4	67.0	66.9
$8,000$	85.0	85.5	89.0	90.5	80.4	80.1	67.5	66.9
$10,000$	85.2	85.7	89.1	91.0	80.7	80.5	67.5	67.3

Table 7: The influence of the dimension based on tensor sketch (TS) and random Maclaurin (RM).

DTD dataset, it achieves the best accuracy using only 2 iterations. In contrast, on Airplane dataset, it achieves the best accuracy with 5 iterations. But using 2 iterations, the accuracy on Airplane dataset is comparable with that using 5 iterations. By default, we set $K = 2$ on all datasets.

Time cost evaluation. We compare the time cost in matrix normalization in the GPU platform of the proposed method with existing methods based on SVD (Lin & Maji (2017)), and Newton-Schulz (NS) iteration. We conduct experiments based on 4 Nvidia K40 GPU cards and set the batch size as 32. Note that, in these experiments we conduct the original bilinear pooling rather than compact bilinear pooling since Newton-Schulz method is not compatible with compact bilinear pooling. As shown in Table 6, SVD-based method is very slow in the GPU platform. The FLOPs of ours is less than 0.1% of NS iteration used in Li et al. (2018). Meanwhile, considering the GPU time, the factual speed-up ratio of ours over NS iteration is beyond 330. The significant reduction in FLOPs and GPU time is contributed by two factors. Firstly, in each iteration, we only need two matrix-vector multiplications whereas NS iteration takes three times of matrix-matrix multiplications. Secondly, ours takes only 2 iterations for a good performance whereas NS iteration takes 5 iterations to achieve a good performance suggested by Li et al. (2018).

Method	Algorithm	Dimension	FLOPs	GPU Time	Accuracy	
					CUB	Airplane
MoNet-2	SVD	$10,000$	4.21G	13850ms	85.7	86.7
Ours	power method	$10,000$	3.2M	2.5ms	85.7	91.0

Table 8: Comparisons between ours and MoNet-2 (Gou et al. (2018)).

Method	Dimension	Norm Time	CUB	Airplane	MIT	DTD
Max-pooling	512	0ms	69.6	78.9	50.4	55.1
Sum-pooling	512	0ms	71.7	82.1	58.7	58.2
BCNN	262K	0ms	84.0	84.1	–	–
Improved BCNN	262K	6.7s	85.8	88.5	–	–
BCNN + Newton-Schulz	262K	833ms	85.7	89.6	80.5	68.3
CBP	8192	0ms	84.0	–	76.2	64.5
LRBP	8192	0ms	84.2	87.3	–	65.8
MoNet-2	10K	13850ms	85.7	86.7	–	–
MoNet	10K	13850ms	86.4	89.3	–	–
BP + RUN (Ours)	262K	2.5ms	86.3	89.8	80.8	68.4
CBP + RUN (Ours)	10K	2.5ms	85.7	91.0	80.5	67.3

Table 9: Comparisons with other pooling methods. We compare the feature dimension, the time cost for matrix normalization per batch (Norm Time) and the accuricies on four benchmarks.

4.4 ABLATION STUDY ON COMPACT BILINEAR POOLING

Influence of the dimension. We adopt two types of CBP, tensor sketch (TS) and random Maclaurin (RM). We set $\eta = 0.6$ and iteration number $K = 2$, and change the dimension after CBP among $\{1\text{K}, 2\text{K}, 4\text{K}, 8\text{K}, 10\text{K}\}$. As shown in Table 7, the accuracies generally increase as the dimension increases. It is expected since the a larger dimension leads to a better approximation for the polynomial kernel. Meanwhile, The accuracies achieved by TS are comparable with that achieved by RM. By default, we use TS for compact bilinear pooling.

Time cost evaluation. We evaluate the time cost used in matrix normalization for compact bilinear pooling (CBP). Since the Newton-Schulz iteration cannot be conducted on the original feature $\mathbf{F}$, it is incompatible with CBP. Thus, we only compare with Monet-2 (Gou et al. (2018)) which conducts SVD on $\mathbf{F}$. $\mathbf{F} \in \mathbb{R}^{784\times512}$ is in a larger size than $\mathbf{B} \in \mathbb{R}^{512\times512}$. Meanwhile, $\mathbf{B}$ is symmetric and only needs compute its left singular vectors $\mathbf{U}$ as well as the singular values $\mathbf{\Sigma}$. But $\mathbf{F}$ is asymmetric and thus needs compute its right singular vectors $\mathbf{V}_F$ besides $\mathbf{U}_F$ and σ_F. Therefore, the FLOPs of computing SVD on $\mathbf{F}$ shown in Table 8 is larger than the FLOPs of computing SVD on $\mathbf{B}$ shown in Table 6. In contrast, the FLOPs of our RUN used for CBP is as the same as that used for original BP. As shown in Table 8, achieving comparable or even better accuracies, we reduce the FLOPs from 4.21G to 3.2M. Moreover, we reduce the time cost in the GPU from 13850ms to 2.5ms, *i.e.*, we achieve a $5540\times$ speedup. Note that, the GPU time cost speedup is larger than the FLOPs reduction ratio since the proposed RUN better supported than SVD in the GPU platform.

4.5 COMPARISON WITH OTHER POOLING METHODS.

We compare with other pooling methods. First of all, we compare with two baselines, which replace the bilinear pooling by max-pooling and sum-pooling, respectively. As shown in Table 9, the features from max-pooling and sum-pooling are compact, and they do not need the matrix normalization. But accuracies achieved by them are lower than methods based on bilinear features. We further compare with B-CNN (Lin et al. (2015)). Benefited from bilinear pooling, B-CNN has achieved good performance but using high-dimensional features. Meanwhile, since there is no matrix normalization, its performance is not as good as ours. We further compare with CBP (Gao et al. (2016)) and LRBP (Kong & Fowlkes (2017)). CBP uses Tensor Sketch and Random Maclaurin to reduce the feature dimension, whereas LRBP adopt the low-rank strategy for a compact feature. Nevertheless, neither CBP nor LRBP adopts matrix normalization. Thus their classification accuries are not as high as ours as shown in Table 9.

We further compare with Improved BCNN (Lin & Maji (2017)) and BCNN + Newton-Schulz (Lin & Maji (2017); Li et al. (2018)). To make a fair comparison with BCNN + Newton-Schulz , we directly use i-SQRT layer released by the authors of Li et al. (2018), and keep all other settings identical. As shown in Table 9, they achieve high accuracies but generate high-dimension features and take high cost in matrix normalization. Then we compare with MoNet-2 and MoNet (Gou et al. (2018)). MoNet-2 achieves high accuracies and generate compact features, but the time cost in the matrix normalization is extremely high. MoNet improves MoNet by fusing the first-order information, achieving higher accuracies, but is also slow in matrix normalization. As shown in Table 9, using CBP, our RUN achieves high accuracies, generates compact features and is very fast. Despite that, we can also further improve the performance of the proposed RUN by fusing the first-order feature likewise MoNet, it is not the focus of this paper.

5 CONCLUSION

We propose a fast rank-1 update normalization (RUN) method for addressing the burstiness in bilinear matrix efficiently. Since it only takes several times of matrix-vector multiplications, the proposed RUN not only takes cheap computation complexity in theory but also is well supported in the GPU platform in practice. More importantly, the proposed RUN supports normalization on compact bilinear features, which have broken the matrix structure. Meanwhile, RUN is differentiable and hence can be easily plugged into a convolutional neural network, which supports an end-to-end training. Our experiments on four datasets show that, combined with original bilinear pooling, we achieve comparable or even better accuracies with a $330\times$ speedup over Newton-Schulz iteration. Moreover, when using compact bilinear pooling, we achieve comparable or even better accuracies on four benchmark datasets with a $5540\times$ speedup over the SVD-based method.

REFERENCES

Relja Arandjelovic, Petr Gronat, Akihiko Torii, Tomas Pajdla, and Josef Sivic. Netvlad: Cnn architecture for weakly supervised place recognition. In *CVPR*, 2016.

Mircea Cimpoi, Subhransu Maji, Iasonas Kokkinos, Sammy Mohamed, and Andrea Vedaldi. Describing textures in the wild. In *CVPR*, 2014.

Yin Cui, Feng Zhou, Jiang Wang, Xiao Liu, Yuanqing Lin, and Serge Belongie. Kernel pooling for convolutional neural networks. In *CVPR*, 2017.

Yang Gao, Oscar Beijbom, Ning Zhang, and Trevor Darrell. Compact bilinear pooling. In *CVPR*, 2016.

Mengran Gou, Fei Xiong, Octavia Camps, and Mario Sznaier. Monet: Moments embedding network. In *CVPR*, 2018.

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

Nicholas J Higham. *Functions of matrices: theory and computation*, volume 104. Siam, 2008.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

Catalin Ionescu, Orestis Vantzos, and Cristian Sminchisescu. Matrix backpropagation for deep networks with structured layers. In *ICCV*, 2015.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. On the burstiness of visual elements. In *CVPR*, 2009.

Herve Jegou, Florent Perronnin, Matthijs Douze, Jorge Sánchez, Patrick Perez, and Cordelia Schmid. Aggregating local image descriptors into compact codes. *IEEE T-PAMI*, 2011.

Purushottam Kar and Harish Karnick. Random feature maps for dot product kernels. In *AISTATS*, 2012.

Shu Kong and Charless Fowlkes. Low-rank bilinear pooling for fine-grained classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 365–374, 2017.

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

Peihua Li, Jiangtao Xie, Qilong Wang, and Wangmeng Zuo. Is second-order information helpful for large-scale visual recognition? In *ICCV*, 2017.

Peihua Li, Jiangtao Xie, Qilong Wang, and Zilin Gao. Towards faster training of global covariance pooling networks by iterative matrix square root normalization. In *CVPR*, 2018.

Tsung-Yu Lin and Subhransu Maji. Improved bilinear pooling with cnns. In *BMVC*, 2017.

Tsung Yu Lin, Aruni Roychowdhury, and Subhransu Maji. Bilinear cnn models for fine-grained visual recognition. In *ICCV*, 2015.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

S. Maji, J. Kannala, E. Rahtu, M. Blaschko, and A. Vedaldi. Fine-grained visual classification of aircraft. Technical report, 2013.

Antoine Miech, Ivan Laptev, and Josef Sivic. Learnable pooling with context gating for video classification. *arXiv preprint arXiv:1706.06905*, 2017.

Florent Perronnin, Jorge Sanchez, and Thomas Mensink. Improving the fisher kernel for large-scale image classification. In *ECCV*, 2010.

Ninh Pham and Rasmus Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *SIGKDD*, pp. 239–247. ACM, 2013.

Ariadna Quattoni and Antonio Torralba. Recognizing indoor scenes. In *CVPR*, 2009.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In *NIPS*, 2014a.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014b.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. 2014.

Qilong Wang, Peihua Li, and Lei Zhang. G2denet: Global gaussian distribution embedding network and its application to visual recognition. In *CVPR*, 2017a.

Yunbo Wang, Mingsheng Long, Jianmin Wang, and Philip S Yu. Spatiotemporal pyramid network for video action recognition. In *CVPR*, 2017b.

Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. 2010.

A APPENDIX

In this section, we prove the Theorem 1 in Section 3.

Recall that $\mathbf{B}_K = \mathbf{B} - \epsilon\mathbf{R}_K$, where

$$\mathbf{R}_K = \mathbf{B}\mathbf{v}_K\mathbf{v}_K^\top / \|\mathbf{v}_K\|_2^2, \tag{14}$$

Using SVD, we factorize

$$\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top, \tag{15}$$

where $\mathbf{U}$ is orthonormal containing the singular vectors and $\mathbf{\Sigma} = \mathrm{diag}(\sigma_1, \cdots, \sigma_D)$ is a diagonal matrix containing singular values. According to Eq. (3), we have

$$\mathbf{v}_K = \mathbf{B}^K\mathbf{v}_0 = \mathbf{U}\mathbf{\Sigma}^K\mathbf{U}^\top\mathbf{v}_0 = \mathbf{U}\mathbf{\Sigma}^K\mathbf{a}, \tag{16}$$

where $\mathbf{a} = \mathbf{U}^\top\mathbf{v}_0$. Plugging Eq. (15) and Eq. (16) into Eq. (14), we have

$$\mathbf{R}_K = \frac{\mathbf{U}\mathbf{\Sigma}^{K+1}\mathbf{a}\mathbf{a}^\top\mathbf{\Sigma}^K\mathbf{U}^\top}{\mathbf{a}^\top\mathbf{\Sigma}^{2K}\mathbf{a}} = \mathbf{U}\mathbf{H}\mathbf{U}^\top, \tag{17}$$

where $\mathbf{H} = (\mathbf{\Sigma}^{K+1}\mathbf{a}\mathbf{a}^\top\mathbf{\Sigma}^K)/(\mathbf{a}^\top\mathbf{\Sigma}^{2K}\mathbf{a})$. As $\mathbf{v}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\mathbf{U}\mathbf{U}^\top = \mathbf{I}$, thus $\mathbf{a} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. That is, $\mathbf{a}$'s entries $\{a_1, a_2, \cdots, a_d\}$ are *i.i.d* random variables with normal distribution. Therefore, the expectation of each off-diagonal entry of $\mathbf{H}$ is 0. That is, $\mathbb{E}(\mathbf{H})$ is a diagonal matrix. We rewrite $\mathbb{E}(\mathbf{H}) = \mathrm{diag}(h_1, \cdots, h_D)$ and

$$h_l = \mathbb{E}(\sigma_l(a_l\sigma_l^k)^2 / \sum_{i=1}^{D}(a_i\sigma_i^k)^2) = \sigma_l\alpha_l, \tag{18}$$

where

$$\alpha_l = \mathbb{E}((a_l\sigma_l^k)^2 / \sum_{i=1}^{D}(a_i\sigma_i^k)^2) \tag{19}$$

In this case, proving Theorem 1 is equivalent to proving that $\alpha_s \geq \alpha_t$ if $s < t$. As we know

$$\alpha_s - \alpha_t = \mathbb{E}\Big(\frac{a_s^2\sigma_s^{2k} - a_s^2\sigma_t^{2k}}{\sum_{i=1}^{D} a_i^2\sigma_i^{2k}}\Big). \tag{20}$$

We define $b_i = a_i^2$ and $y_i = \sigma_i^{2k}$, then seek to prove

$$\alpha_s - \alpha_t = \mathbb{E}\Big(\frac{b_s y_s - b_t y_t}{\sum_{i=1}^D b_i y_i}\Big) \geq 0, \quad \text{if } s < t. \tag{21}$$

As $y_s \geq y_t$ and $y_1 \geq y_2 \cdots \geq y_D$, we obtain

$$\frac{b_s y_s - b_t y_t}{\sum_{i=1}^D b_i y_i} \geq \frac{y_t}{y_1} \frac{b_s - b_t}{\sum_{i=1}^D b_i}. \tag{22}$$

Thus,

$$\mathbb{E}\Big(\frac{b_s y_s - b_t y_t}{\sum_{i=1}^D b_i y_i}\Big) \geq \frac{y_t}{y_1} \mathbb{E}\Big(\frac{b_s - b_t}{\sum_{i=1}^D b_i}\Big). \tag{23}$$

Since $\{a_i\}_1^D$ are *i.i.d*, $\{b_i\}_1^D$ are also *i.i.d*. Therefore,

$$\mathbb{E}\Big(\frac{b_s - b_t}{\sum_{i=1}^D b_i}\Big) = \mathbb{E}\Big(\frac{b_s}{\sum_{i=1}^D b_i}\Big) - \mathbb{E}\Big(\frac{b_t}{\sum_{i=1}^D b_i}\Big) = 0. \tag{24}$$

Plugging Eq. (24) into Eq. (23), we obtain

$$\mathbb{E}\Big(\frac{b_s y_s - b_t y_t}{\sum_{i=1}^D b_i y_i}\Big) \geq 0. \tag{25}$$

B APPENDIX

We compute the differentiation of $\bar{\mathbf{F}}_K$ based on Eq. (10):

$$\begin{aligned} d\mathbf{F}_K = d\mathbf{F} - \eta \frac{(d\mathbf{F})\mathbf{v}_K\mathbf{v}_K^\top + \mathbf{F}(d\mathbf{v}_K)\mathbf{v}_K^\top + \mathbf{F}\mathbf{v}_K(d\mathbf{v}_K^\top)}{\mathbf{v}_K^\top\mathbf{v}_K} \\ + \eta \frac{(d\mathbf{v}_K^\top)\mathbf{v}_K + \mathbf{v}_K^\top d\mathbf{v}_K}{(\mathbf{v}_K^\top\mathbf{v}_K)^2}\mathbf{F}\mathbf{v}_K\mathbf{v}_K^\top. \end{aligned} \tag{26}$$

Meanwhile, Eq. (9) leads to

$$\mathbf{v}_K = (\mathbf{F}^\top\mathbf{F})^K\mathbf{v}_0. \tag{27}$$

Since $\mathbf{v}_0$ is a constant vector, based on Eq. (27), we obtain

$$\begin{aligned} d\mathbf{v}_K &\equiv K(\mathbf{F}^\top\mathbf{F})^{K-1}[(d\mathbf{F}^\top)\mathbf{F} + \mathbf{F}^\top d\mathbf{F}]\mathbf{v}_0, \\ d\mathbf{v}_K^\top &\equiv K\mathbf{v}_0^\top[(d\mathbf{F}^\top)\mathbf{F} + \mathbf{F}^\top d\mathbf{F}](\mathbf{F}^\top\mathbf{F})^{K-1}, \end{aligned} \tag{28}$$

Plugging Eq. (28) in Eq. (26), we obtain

$$d\mathbf{F}_K = \sum_{i=0}^4 l_i^1(\mathbf{F}) d\mathbf{F} r_i^1(\mathbf{F}) + \sum_{j=1}^4 l_j^2(\mathbf{F})(d\mathbf{F})^\top r_j^2(\mathbf{F}), \tag{29}$$

where $\{l_i^1(\mathbf{F}), r_i^1(\mathbf{F})\}_{i=0}^4$ and $\{l_i^2(\mathbf{F}), r_i^2(\mathbf{F})\}_{i=1}^4$ are

$$
\begin{aligned}
&l_0^1(\mathbf{F}) = \mathbf{I}, \quad r_0^1(\mathbf{F}) = \mathbf{I} - \eta \mathbf{v}_K \mathbf{v}_K^\top / (\mathbf{v}_K^\top \mathbf{v}_K), \\
&l_1^1(\mathbf{F}) = \frac{-\eta K \mathbf{F}(\mathbf{F}^\top \mathbf{F})^{K-1} \mathbf{F}^\top}{\mathbf{v}_K^\top \mathbf{v}_N}, \quad r_1^2(\mathbf{F}) = \mathbf{v}_0 \mathbf{v}_K^\top, \\
&l_2^1(\mathbf{F}) = \frac{-\eta K \mathbf{F} \mathbf{v}_K \mathbf{v}_0^\top \mathbf{F}^\top}{\mathbf{v}_K^\top \mathbf{v}_N}, \quad r_2^1(\mathbf{F}) = (\mathbf{F}^\top \mathbf{F})^{K-1}, \\
&l_3^1(\mathbf{F}) = \frac{\eta K \mathbf{v}_0^\top \mathbf{F}^\top}{(\mathbf{v}_K^\top \mathbf{v}_K)^2}, \quad r_3^1(\mathbf{F}) = (\mathbf{F}^\top \mathbf{F})^{K-1} \mathbf{v}_K \mathbf{F} \mathbf{v}_K \mathbf{v}_K^\top, \\
&l_4^1(\mathbf{F}) = \frac{\eta K \mathbf{v}_K^\top (\mathbf{F}^\top \mathbf{F})^{K-1} \mathbf{F}^\top}{(\mathbf{v}_K^\top \mathbf{v}_K)^2}, \quad r_4^1(\mathbf{F}) = \mathbf{v}_0 \mathbf{F} \mathbf{v}_K \mathbf{v}_K^\top, \\
&l_1^2(\mathbf{F}) = \frac{-\eta K \mathbf{F}(\mathbf{F}^\top \mathbf{F})^{K-1}}{\mathbf{v}_K^\top \mathbf{v}_K}, \quad r_1^2(\mathbf{F}) = \mathbf{F} \mathbf{v}_K \mathbf{v}_K^\top, \\
&l_2^2(\mathbf{F}) = \frac{-\eta K \mathbf{F} \mathbf{v}_K \mathbf{v}_0^\top}{\mathbf{v}_K^\top \mathbf{v}_K}, \quad r_2^2(\mathbf{F}) = \mathbf{F}(\mathbf{F}^\top \mathbf{F})^{K-1}, \\
&l_3^2(\mathbf{F}) = \frac{\eta K \mathbf{v}_0^\top}{(\mathbf{v}_K^\top \mathbf{v}_K)^2}, \quad r_3^1(\mathbf{F}) = \mathbf{F}(\mathbf{F}^\top \mathbf{F})^{K-1} \mathbf{v}_K \mathbf{F} \mathbf{v}_K \mathbf{v}_K^\top, \\
&l_4^2(\mathbf{F}) = \frac{\eta K \mathbf{v}_K^\top (\mathbf{F}^\top \mathbf{F})^{K-1}}{(\mathbf{v}_K^\top \mathbf{v}_K)^2}, \quad r_4^2(\mathbf{F}) = \mathbf{F} \mathbf{v}_0 \mathbf{F} \mathbf{v}_K \mathbf{v}_K^\top.
\end{aligned} \tag{30}
$$

According to the definition,

$$
dL \equiv \mathrm{vec}(\frac{\partial L}{\partial \mathbf{F}})^\top \mathrm{vec}(d\mathbf{F}) \equiv \mathrm{vec}(\frac{\partial L}{\partial \bar{\mathbf{F}}_K})^\top \mathrm{vec}(d\bar{\mathbf{F}}_K). \tag{31}
$$

Since $\mathrm{trace}(\mathbf{A}\mathbf{B}^\top) \equiv \mathrm{vec}(\mathbf{A})^\top \mathrm{vec}(\mathbf{B})$, we further obtain

$$
\mathrm{trace}(d\mathbf{F}^\top \frac{\partial L}{\partial \mathbf{F}}) \equiv \mathrm{trace}[d\bar{\mathbf{F}}_K^\top \frac{\partial L}{\partial \bar{\mathbf{F}}_K}] \tag{32}
$$

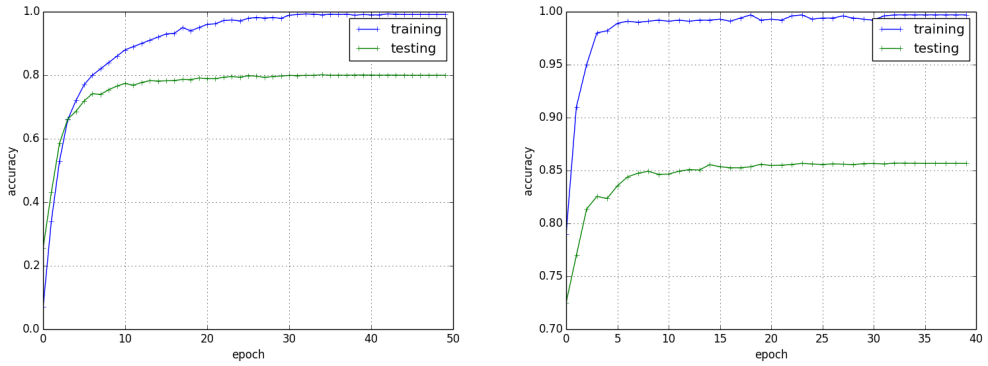
Plugging Eq. (29) into Eq. (32), we obtain

$$
\begin{aligned}
\mathrm{trace}(d\mathbf{F}^\top \frac{\partial L}{\partial \mathbf{F}}) &\equiv \mathrm{trace}\Big\{ \big[\sum_{i=1}^{5} l_i^1(\mathbf{F}) d\mathbf{F} r_i^1(\mathbf{F}) + \sum_{j=1}^{4} l_j^2(\mathbf{F}) (d\mathbf{F})^\top r_j^2(\mathbf{F}) \big]^\top \frac{\partial L}{\partial \bar{\mathbf{F}}_K} \Big\} \\
&\equiv \mathrm{trace}\Big\{ d\mathbf{F}^\top \big[\sum_{i=1}^{5} l_i^1(\mathbf{F})^\top \frac{\partial L}{\partial \bar{\mathbf{F}}_K} r_i^1(\mathbf{F})^\top + \sum_{j=1}^{4} r_j^2(\mathbf{F}) (\frac{\partial L}{\partial \bar{\mathbf{F}}_K})^\top l_j^2(\mathbf{F}) \big] \Big\}.
\end{aligned} \tag{33}
$$

Compare the LHS and RHS of Eq. (33), we obtain

$$
\frac{\partial L}{\partial \mathbf{F}} = \big[\sum_{i=1}^{5} l_i^1(\mathbf{F})^\top \frac{\partial L}{\partial \bar{\mathbf{F}}_K} r_i^1(\mathbf{F})^\top + \sum_{j=1}^{4} r_j^2(\mathbf{F}) (\frac{\partial L}{\partial \bar{\mathbf{F}}_K})^\top l_j^2(\mathbf{F}) \big]. \tag{34}
$$

Eq. (34) gives the backward path which takes $\partial L / \partial \mathbf{F}_K$ as input and outputs $\partial L / \partial \mathbf{F}$.



(a) Phase 1 (b) Phase 2

Figure 4: The accuacy changes in the training process.

C APPENDIX

In this section, we plot the accuracy change of the proposed RUN in the whole training process. We test it on the CUB dataset. We use the proposed RUN with compact bilinear pooling implemented by tensor sketch and set the feature dimension as 10K. As we mentioned, the training is two-phase. We plot the accuracy change in each phase in Figure 4.