Product Sparse Coding

Tiezheng Ge\textsuperscript{1*} \quad Kaiming He\textsuperscript{2} \quad Jian Sun\textsuperscript{2}

\textsuperscript{1}University of Science and Technology of China \quad \textsuperscript{2}Microsoft Research

Abstract

Sparse coding is a widely involved technique in computer vision. However, the expensive computational cost can hamper its applications, typically when the codebook size must be limited due to concerns on running time. In this paper, we study a special case of sparse coding in which the codebook is a Cartesian product of two subcodebooks. We present algorithms to decompose this sparse coding problem into smaller subproblems, which can be separately solved. Our solution, named as Product Sparse Coding (PSC), reduces the time complexity from $O(K)$ to $O(\sqrt{K})$ in the codebook size $K$. In practice, this can be 20-100\times faster than standard sparse coding. In experiments we demonstrate the efficiency and quality of this method on the applications of image classification and image retrieval.

1. Introduction

Sparse Coding (SC) \cite{23} is a broadly studied and successful technique in computer vision. It represents a given vector as a sparse linear combination of the elements in a codebook. Its applications involve image denoising \cite{1}, image super-resolution \cite{31}, image segmentation \cite{21}, image classification \cite{32}, face recognition \cite{29}, etc.

However, sparse coding is computationally expensive. The state-of-the-art solutions, such as the Least Angle Regression (LARS) \cite{6} and the Feature Sign algorithm \cite{19}, present a time complexity of $O(K)$ in the codebook size $K$. This computational cost is still demanding when the codebook is large and a considerable set of vectors have to be encoded, e.g., as in the scenario of image classification \cite{32}. As a result, the sparse-coding-based methods may retreat to use smaller codebooks (e.g., $K=1000$ in \cite{32}) for practical running time, but the smaller codebooks may hamper the representing ability and the quality. Thus the sparse-coding-based methods can appear less competitive due to the limited codebook size, typically when the accuracy of other encoding methods can be improved by simply enlarging the codebook (before over-fitting) \cite{4}.

Closely related to sparse coding, Vector Quantization (VQ) \cite{12} is another widely used technique in computer vision. Vector quantization finds the nearest codeword to encode a vector (Fig. 1(a)). Although this seems a simple computation, it turns out to be nontrivial if exponentially large codebooks are used, e.g., in the case of data compression \cite{12} and nearest neighbor search \cite{14, 3}. The Product Quantization (PQ) \cite{12, 14} is an efficient solution to exponentially large codebooks. The basic idea is to decompose the vector space into the Cartesian product of subspaces and separately quantize each subspace by a subcodebook (Fig. 1(b)). With $m$ small subcodebooks of a size $k$, the effective codebook size in the full space is $K=k^m$, while the complexity is $O(\sqrt{K})$. The product quantization techniques have witnessed great success in large scale problems, including nearest neighbor search \cite{14, 3, 9, 10, 22, 30} and large scale learning \cite{25, 27}.

Driven by PQ, in this work we present a method called Product Sparse Coding (PSC). It shares the same encoding model as sparse coding, but requires the codebook to be a Cartesian product of smaller subcodebooks. The relation of SC vs. PSC is analogous to the relation of VQ vs. PQ (Fig. 1). If we can separately solve smaller subproblems in each subspace, we can reduce the linear time complexity of sparse coding.

But unlike PQ, the PSC problem is not readily separable because the subproblems are mutually dependent. In this paper we investigate the case of two subspaces. We find the dependency of the two subspaces can be determined by a single unknown variable. We propose a binary search solution to compute this variable. Then we can separately solve smaller subproblems. Under some conditions, we theoretically prove this solution to the PSC problem is globally optimal. We further present an approximate solution that is more efficient and works well in practice.

For a codebook of the size $K$, PSC has a time complexity of $O(\sqrt{K})$. So it is a very efficient solution to large codebooks that are infeasible for sparse coding methods. For example, it can increase the speed by 20-100\times when $K > 10^4$. The gain in speed is at the price of constraining the codebook to be a product of subcodebooks (Fig. 1(d)). So our solution is a trade-off between speed and quality.

\textsuperscript{*}This work is done when Tiezheng Ge is an intern at Microsoft Research Asia.
This trade-off can be worthwhile or even necessary, typically when the high computational cost prohibits the usage of large codebooks for sparse coding. This will be demonstrated in the applications of image classification [32] and image retrieval [15]. We will show PSC has competitive accuracy among various state-of-the-art methods for these applications, while is very efficient.

How does PSC work? Consider a “color” subcodebook that consists of 4 elements \{red, yellow, green, blue\}, and another “shape” subcodebook that consists of \{circle, square, triangle, ellipse\}. The Cartesian product of these two subcodebooks involves 16 distinct elements (Fig. 1(d)). To encode a new item, we need not explicitly enumerate all 16 elements, but instead only need to visit 4 elements in each of the two subcodebooks.

2. Formulations

2.1. Background: from Vector Quantization to Product Quantization

Let \( x \in \mathbb{R}^d \) be a vector to be encoded. Suppose the codebook is given. The encoding problem of VQ [12] can be formulated as:

\[
\begin{align*}
\min_{\mathbf{y}} & \quad \| \mathbf{x} - \mathbf{A} \mathbf{y} \|^2, \\
\text{s.t.} & \quad \|\mathbf{y}\|_0 = 1, |\mathbf{y}| = 1, \mathbf{y} \succeq 0.
\end{align*}
\]

Here \( \mathbf{A} \) is a \( d \)-by-\( K \) matrix called a codebook, \( \mathbf{y} \) is a \( K \)-by-1 vector called a code, \( \| \cdot \| \) is the \( l_2 \) norm, \( \| \cdot \|_0 \) is the zero norm (number of non-zero entries), and \( | \cdot | \) are the \( l_1 \) norm. The codebook \( \mathbf{A} \) has \( K \) codewords as its columns. The constraint means \( \mathbf{y} \) has one and only one non-zero entry whose value is 1. Minimizing (1) is equivalent to finding the nearest codeword. See Fig. 1 (a).

The Product Quantization (PQ) [12, 14] can be considered as a special case of VQ when the codebook is the Cartesian product of subcodebooks. In the case of two subcodebooks, the encoding problem of PQ can be written as [9, 10]:

\[
\begin{align*}
\min_{\mathbf{y}} & \quad \| \mathbf{x} - \mathbf{A} \mathbf{y} \|^2, \\
\text{s.t.} & \quad \|\mathbf{y}\|_0 = 1, |\mathbf{y}| = 1, \mathbf{y} \succeq 0 \\
& \quad \text{and} \quad \mathbf{A} = \mathbf{A}_1 \times \mathbf{A}_2.
\end{align*}
\]

where “\( \times \)” denotes the Cartesian product. \( \mathbf{A}_1 \) and \( \mathbf{A}_2 \) are two subcodebooks of a size \( \frac{d}{2} \)-by-\( k \). Any codeword in \( \mathbf{A} \) is the concatenation of a subcodeword in \( \mathbf{A}_1 \) and a subcodebook in \( \mathbf{A}_2 \). So \( \mathbf{A} \) is a \( d \)-by-\( K \) matrix with \( K = k^2 \) (if there are \( m \) subspaces, then \( K = k^m \)). See Fig. 1 (b).

The PQ problem in (2) can be separated into smaller independent subproblems. Each subproblem is simply applying VQ in the subspaces with the subcodebook. The cost of each subproblem is merely \( O(\sqrt{K}) \), whereas the cost of directly applying VQ on (2) would be \( O(K) \).

2.2. From Sparse Coding to Product Sparse Coding

SC is closely related to VQ [32, 5]. In this paper we consider the SC problem in this form:

\[
\begin{align*}
\min_{\mathbf{y}} & \quad \| \mathbf{x} - \mathbf{A} \mathbf{y} \|^2 + \lambda |\mathbf{y}|, \\
\text{s.t.} & \quad \mathbf{y} \succeq 0
\end{align*}
\]

where \( \mathbf{A} \) is a \( d \)-by-\( K \) codebook and \( \lambda \) is a regularization parameter. The constraint \( \mathbf{y} \succeq 0 \) means all entries in the code are non-negative. An illustration is in Fig. 1 (c).

It can be time-consuming to solve (3). The state-of-the-art methods have a time complexity linear in \( K \) and also in other nontrivial factors (more details are in Sec. 3.5).

Motivated by the relation between VQ and PQ, we propose a new formulation called Product Sparse Coding.
To give a separate form of \( y \) here to be determined (equivalently: a vector). Then the objective function in (4) becomes:

\[
\min_{\mathbf{y}} \| \mathbf{x} - A\mathbf{y} \|^2 + \lambda \| \mathbf{y} \|, \quad \text{(4)}
\]

\[
s.t. \quad \mathbf{y} \geq 0
\]

and \( A = A_1 \times A_2 \).

Here \( A_1 \) and \( A_2 \) are subcodebooks of a size \( \frac{n}{2} \)-by-\( k \), and \( A \) is their Cartesian product and is \( d \)-by-\( K \) with \( K = k^2 \). This is illustrated in Fig. 1 (d).

If we can separate this problem into two subproblems as in PQ, the time complexity of each subproblem can become linear in \( \sqrt{K} \). However, this problem is not readily separable due to the regularization \( \lambda \| \mathbf{y} \| \). In the following we propose solutions to this issue.

3. Algorithms

3.1. Separate the Problem

In the case of two subspaces, we denote \( \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \) where \( \mathbf{x}_1 \) and \( \mathbf{x}_2 \) are the first and second subvector of \( \mathbf{x} \) (that is, the first/second half of its entries). Further, any codeword in \( A \) can be represented as \( \begin{bmatrix} \mathbf{a}_{1,i} \\ \mathbf{a}_{2,j} \end{bmatrix} \) where \( \mathbf{a}_{1,i} \) is the \( i \)-th codeword in \( A_1 \) and \( \mathbf{a}_{2,j} \) the \( j \)-th in \( A_2 \), for \( i, j = 1, \ldots, k \). We denote the coefficient of this codeword as \( y_{ij} \) (note \( y \) is still a vector). Then the objective function in (4) becomes:

\[
\min \| \mathbf{x}_1 - \sum_{i,j} \begin{bmatrix} \mathbf{a}_{1,i} \\ \mathbf{a}_{2,j} \end{bmatrix} y_{ij} \|^2 + \lambda \| \mathbf{y} \|, \quad \text{(5)}
\]

The first term can be expanded as:

\[
\| \mathbf{x}_1 - \sum_i (\mathbf{a}_{1,i} \sum_j y_{ij}) \|^2 + \| \mathbf{x}_2 - \sum_j (\mathbf{a}_{2,j} \sum_i y_{ij}) \|^2. \quad \text{(6)}
\]

We introduce two vectors \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \) of the size \( k \)-by-\( 1 \), whose entries are:

\[
\mathbf{u}_{1,i} = \sum_j y_{ij}, \quad \mathbf{u}_{2,j} = \sum_i y_{ij}. \quad \text{(7)}
\]

Note \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \) are two marginal sums. Then (6) becomes:

\[
\| \mathbf{x}_1 - A_1 \mathbf{u}_1 \|^2 + \| \mathbf{x}_2 - A_2 \mathbf{u}_2 \|^2. \quad \text{(8)}
\]

This gives a separate representation of the first term in (5). This is also a way of separating the PQ problem in (2).

Because \( \mathbf{y} \geq 0 \), so the vectors \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \) are subject to the constraint \( \sum_{i,j} |y_{ij}| = \sum_i |u_{1,i}| = \sum_j |u_{2,j}| \), or equivalently:

\[
|\mathbf{y}| = |\mathbf{u}_1| = |\mathbf{u}_2|. \quad \text{(9)}
\]

To give a separate form of \( |\mathbf{y}| \), we introduce a parameter \( \lambda_1 \) to be determined (\( 0 < \lambda_1 < \lambda \)). Denoting \( \lambda_2 = \lambda - \lambda_1 \), we can rewrite the PSC problem (4) as:

\[
\min_{\mathbf{u}_1, \mathbf{u}_2} \| \mathbf{x}_1 - A_1 \mathbf{u}_1 \|^2 + \lambda_1 |\mathbf{u}_1| + \| \mathbf{x}_2 - A_2 \mathbf{u}_2 \|^2 + \lambda_2 |\mathbf{u}_2|,
\]

\[
s.t. \quad \mathbf{u}_1 \geq 0, \quad \mathbf{u}_2 \geq 0 \quad \text{and} \quad |\mathbf{u}_1| = |\mathbf{u}_2|. \quad \text{(10)}
\]

If we ignore the constraint \( |\mathbf{u}_1| = |\mathbf{u}_2| \), we can have two separate subproblems:

\[
\min_{\mathbf{u}_1} \| \mathbf{x}_1 - A_1 \mathbf{u}_1 \|^2 + \lambda_1 |\mathbf{u}_1|,
\]

\[
s.t. \quad \mathbf{u}_1 \geq 0 \quad \text{and} \quad |\mathbf{u}_1| \leq |\mathbf{u}_2|. \quad \text{(11)}
\]

Each is a SC problem as in (3). But the codebooks \( A_1 \) and \( A_2 \) are much smaller. Solving these two subproblems can be much faster.

Suppose \( \lambda_1 \) has been set to a “special” value \( \lambda_1^* \) that the two subproblems will produce a pair of solutions \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \) satisfying \( |\mathbf{u}_1| = |\mathbf{u}_2| \). Given \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \), the solution \( \mathbf{y} \) is not unique because only two of its marginal sums are given.

We show the following \( \mathbf{y} \) is a solution that satisfies (7):

\[
y_{ij} = u_{1,i} u_{2,j} / \sqrt{|\mathbf{u}_1| |\mathbf{u}_2|}, \quad \text{(12)}
\]

or equivalently:

\[
\mathbf{y} = \text{vec}(\mathbf{u}_1 \mathbf{u}_2^T) / \sqrt{|\mathbf{u}_1| |\mathbf{u}_2|} \quad \text{(13)}
\]

where vec(\( \cdot \)) rearrange the matrix \( \mathbf{u}_1 \mathbf{u}_2^T \) into a vector.

If such a value \( \lambda_1^* \) exists, then we can prove the solution \( \mathbf{y} \) in (13) is a global optimum of the PSC problem (4) (see Theorem A.1). If we can find the value \( \lambda_1^* \), then we can obtain a solution to the PSC problem (4) simply from the solutions to the subproblems (11).

Before we introduce an algorithm to find \( \lambda_1^* \), we should note \( \lambda_1^* \) does not always exist (explained later). In case it does not exist, the global optimal solution to (4) can not be obtained through the subproblems.

3.2. An Iterative Algorithm

Next we describe an algorithm to compute \( \lambda_1^* \) if it exists. Consider the solutions \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \) to the separated subproblems in (11). We can proof if \( \lambda_1 \) increases, then \( |\mathbf{u}_1| \) is non-increasing. Intuitively, the increased \( \lambda_1 \) will penalize \( |\mathbf{u}_1| \) more, so \( |\mathbf{u}_1| \) will not increase. A formal proof is in Theorem A.2. So \( |\mathbf{u}_1| \) is a monotonically decreasing function in \( \lambda_1 \). Similarly, because \( \lambda_2 = \lambda - \lambda_1 \), so \( |\mathbf{u}_2| \) is a monotonically increasing function in \( \lambda_1 \). See Fig. 2 (left). If the two monotonically curves intersect in the range of \( (0, \lambda) \), then \( \lambda_1^* \) exists.

The monotonicity leads to a simple binary search (half-interval search) algorithm of finding \( \lambda_1^* \). Consider an initial
value of $\lambda_1$, like $\lambda/2$. We use this value to solve the separate subproblems in (11). If $|u_1| = |u_2|$, then we have found the expected value. If not, without loss of generality, we consider $|u_1| < |u_2|$. Because of the monotonicity, it is easy to show any $\lambda_1 \in [\lambda/2, \lambda)$ will also leads to $|u_1| < |u_2|$ (see Fig. 2 left). So the search range of $\lambda_1$ can be halved and becomes $(0, \lambda/2)$.

We can iterate this procedure. After each iteration, the search range is halved, and the true $\lambda_1^*$ can only be found in this range. The iteration stops when $|u_1| = |u_2|$ or the search range is sufficiently narrow. This solution is described in Algorithm 1. This algorithm always converges to the solution $|u_1| = |u_2|$ if $\lambda_1^*$ exists. We called it Iterative Product Sparse Coding (IPSC). Fig. 1 shows the behavior of IPSC in 100 randomly sampled SIFT vectors. We see that after 10 iterations (the search range spans $\lambda/1024$) the gap between $|u_1|$ and $|u_2|$ is ignorable.

But there are cases that $\lambda_1^*$ does not exist. Fig. 2 (right) shows an example - the two monotonic curves do not intersect. If this happens, the PSC problem (4) is not separable in our way. Fig. 2 (right) also indicates this will happen when $\lambda_1 \to 0$ gives $|u_1| < |u_2|$, or $\lambda_1 \to \lambda$ gives $|u_1| > |u_2|$. This suggests the magnitudes of the two subvectors $x_1$ and $x_2$ are very imbalanced. In our experiments, these cases are in a few number. In a set of one million randomly sampled SIFT vectors, there are about 1% of such cases. In case it happens, IPSC stops at the max iteration and can still output $y$ using (13), but the global optimality is lost. In the pooling-based applications [32], we find there is no observable impact in practice.

### 3.3. An Approximate Algorithm

Next we describe a non-iterative approximate solution that works well in practice. In our PSC problem in (4), the task is to find the code $y$. If we ignore the constraint $|u_1| = |u_2|$, then any pair of $u_1$ and $u_2$ can still produce a vector $y$ through (13). So we can simply use a heuristic $\lambda_1$, like $\lambda/2$, to separate the problem.

Formally, the Approximate Product Sparse Coding (APSC) has these steps: (i) set $\lambda_1 = \lambda_2 = \lambda/2$, (ii) solve the two subproblems in (11), and (iii) compute $y$ using (13).

In experiments we find the simple APSC is a reasonable approximation of the global optimum achieved by IPSC (if $\lambda_1^*$ exists). Table 1 shows the objective function values in (4) computed using the resulting $y$ given by IPSC/APSC. The values are averaged over $10^5$ randomly sampled SIFT vectors (in which $\lambda_1^*$ exists). We see the relative error is smaller than 1%. A possible explanation is that the two subspaces of the SIFT vectors are about balanced, and $\lambda_1^*$ might be not far from $\lambda/2$.

As an approximate algorithm, APSC need not consider the existence of $\lambda_1^*$. We find APSC works well in the appli-
<table>
<thead>
<tr>
<th>IPSC</th>
<th>APSC</th>
<th>relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.590</td>
<td>0.594</td>
<td>7e-3</td>
</tr>
</tbody>
</table>

Table 1: The values of the objective function averaged over $10^5$ randomly sampled SIFT vectors.

3.4. Codebook Training

Next we describe the codebook training. Note, however, the above derivations are true for any codebook satisfying $A = A_1 \times A_2$, not necessarily trained in the way below.

Given a sample set $\{x^s\}$, we optimize:

$$\min_{A_1, A_2, \{y^s\}} \sum_s \left( \|x^s - Ay^s\|^2 + \lambda |y^s| \right),$$

subject to $y^s \succeq 0$, $\forall s$

$$A = A_1 \times A_2,$$

and $\|a_{1,i}\|^2 = \|a_{2,j}\|^2 = 1/2$, $i,j = 1,...,k$

where $y^s$ is the code of a sample $x^s$. Here $a_{1,i}$ and $a_{2,j}$ are the subcodewords in the subcodebooks $A_1$ and $A_2$. $\|a_{1,i}\|^2 = \|a_{2,j}\|^2 = \frac{1}{2}$ guarantees that the $l_2$-norm of any codeword in $A$ is 1.

We use an EM-alike solution to optimize (14) just like traditional VQ/SC. With $A_1, A_2$ fixed, we solve each $y^s$ using IPSC or APSC. With $\{y^s\}$ fixed, we first compute $\{u_1^s, u_2^s\}$ using (7). Then we compute $A_1$ and $A_2$ through separately minimizing two subproblems given by (8), each of which can be solved by [19]. This process is iterated.

In the entire procedure we never need to express the product matrix $A$. If the APSC is used to solve for $y^s$, then the algorithm is equivalent to separately training $A_1$ and $A_2$ in the two subspaces. The training using APSC is very efficient. For example, it takes less than one minute to train the codebooks using $10^5$ SIFT when $K=16384 (k=128)$.

3.5. Complexity

The PSC is a special case of sparse coding when a product codebook is used. Because PSC only need to solve smaller subproblems, its complexity is low even the product codebook is large.

We adopt a state-of-the-art SC solver - the Feature Sign (FS) algorithm [19]. For a general (non-product) codebook $A$ of size $d$-by-$K$, the time complexity of FS is roughly $O(Kd) + O(KS)$ per vector, where $S$ is the “sparsity” of the code (number of non-zero entries). Here $O(Kd)$ is contributed by projecting $x$ onto the codebook, and $O(KS)$ is due to the feature sign steps.

Now consider a product codebook $A$. If the sparsity in each of $u_1$ or $u_2$ is $s$, then the sparsity of the code $y$ in (13) is $S = s^2$. We use the FS algorithm to solve the two subproblems. Then the APSC algorithm has a time complexity $O(2k^2) + O(2ks) = O(\sqrt{Kd}) + O(2\sqrt{KS})$. The complexity is much smaller than SC mainly due to the square root.

Our algorithm also has a smaller memory complexity. Throughout the encoding/training algorithms, we never need to explicitly compute or store the matrix $A$. All the computation can be done by $A_1$ and $A_2$. We consider the case of encoding a large set of vectors, e.g., as in image classification [32]. One way of efficiently applying FS is to pre-compute the $K$-by-$K$ Gram matrix ($A_1^T A_1$) [32]. This matrix consumes $O(K^2)$ memory. In PSC we only need to pre-compute two smaller $k$-by-$k$ Gram matrices ($A_1^T A_1$ and $A_2^T A_2$). Their memory cost is $O(2k^2) = O(2K)$. This is a significantly smaller consumption. For example, the Gram matrix of SC takes 2.14GB memory when $K=16384$, whereas the two smaller Gram matrices of PSC take only 262KB ($k=128$). Further, it is time-consuming to access a large Gram matrix. So the memory issue also impacts running time.

We should remark the above time/memory gain is the result of using a product codebook $A = A_1 \times A_2$. Because the SC method does not have this constraint, it can use a better codebook. So there is a trade-off between the time/memory gain and quality loss. We will demonstrate this trade-off by experiments.

4. Experiments

In all the experiments we use the APSC algorithm unless specified, because we find the improvement of using IPSC is ignorable compared with APSC.

4.1. Computational Efficiency

We randomly sample $2 \times 10^5$ SIFT vectors ($d=128$) [20] extracted from the Caltech101 image set [8]. We train a codebook $A$ of the size $d$-by-$K$ for SC, and two subcodebooks $A_1$ and $A_2$ of the size $\frac{d}{2}$-by-$\sqrt{K}$ for PSC. We use $10^5$ vectors to train. The remaining $10^5$ vectors are to be encoded. All experiments are on a workstation with an Intel Xeon 2.67GHz CPU using a single thread. The implementation of all methods is in C++.

Table 2 shows the total encoding time of SC and PSC for $10^5$ vectors when $K=16384$. The parameter $\lambda$ is set 0.3, as we will use for image classification¹ (Sec. 4.2).

In Table 2, SC-FS is the Feature Sign variant that computes the Gram matrix before encoding all the vectors. The APSC also uses this Feature Sign algorithm to solve the two subproblems in (11). We see APSC is faster than SC-FS by $10^4 \times$. This is consistent with the complexity estimation: from $O(K)$ to $O(\sqrt{K})$.

We also evaluate two more variants of SC solvers based on FS. In the variant SC-FS-OTF, the full Gram matrix is

¹For SC/PSC, we normalize the SIFT vectors so the $l_2$ norms are 1. The value of $\lambda$ corresponds to this implementation.
Table 2: Computational time of encoding $10^5$ SIFT vectors. The codebook size is $K=16384$. The running time are given in seconds. The column “vs. APSC” shows the multiples of the APSC running time.

<table>
<thead>
<tr>
<th>method</th>
<th>time (s)</th>
<th>vs. APSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC-FS</td>
<td>490</td>
<td>104×</td>
</tr>
<tr>
<td>SC-FS-OTF</td>
<td>634</td>
<td>135×</td>
</tr>
<tr>
<td>SC-FS-NN</td>
<td>114</td>
<td>24×</td>
</tr>
<tr>
<td>IPSC</td>
<td>14</td>
<td>3×</td>
</tr>
<tr>
<td>APSC</td>
<td>4.7</td>
<td>-</td>
</tr>
</tbody>
</table>

The variant SC-FS-NN is used in the public code in [32]. It finds the $n$ nearest codewords to the vector, and solves a smaller SC problem that consists of these $n$ codewords. Its time complexity is $O(Kd) + O(nS)$ per vector. Note $O(Kd)$ is because of the nearest neighbor search. Here we use $n=200$ as in [32]. We see this solution is faster than SC-FS, but is still slower than PSC by 24×. This is because $O(Kd)$ still contributes substantially to the running time.

We also evaluate the running time of IPSC using 10 iterations. Its running time is 3× of the APSC. It is not linear on the iteration number because some intermediate results can be reused in the IPSC iterations.

### 4.2. For Image Classification

Our first application of PSC is image classification [32, 4, 5]. The experiment settings follow those in the benchmark paper [4] and its public code. We evaluate on the Caltech101 dataset [8]. It contains about 9K images in 102 categories (one background). For each image we extract 128-D SIFT vectors at four scales and a step size 4. These SIFT vectors are encoded using SC, PSC, or other methods. The codes are pooled to generate the image representation. We use the spatial pyramid pooling (SPM) [18] in three levels: $1\times1$, $2\times2$, and $4\times4$ for a total of 21 regions. We use max pooling for SC and PSC. The pooled image representations are used to train a linear SVM [7]. We use 30 images per category for training and the rest for testing. The performance is evaluated by the average classification accuracy. The implementation is in Matlab, except the encoding steps are in mex. The SC-FS-NN is used as solver for SC.

Table 3 shows the performance of SC and PSC. We use $K=4096, 8100$, and $16384$, corresponding to $k=64, 90, 128$. We use $\lambda=0.3$ for both SC and PSC. We see our method is slightly worse than SC at the same $K$. This is as expected, because our codebook is not as accurate as SC due to the product constraint.

Despite the small accuracy loss, we see the speed gain is large. Table 3 shows the average time spent on encoding an image (not including SIFT extraction). We see PSC is much faster. In our implementation based on [4], the total running time of encoding the 9K images (including SIFT extraction) is less than 15 minutes using 8 cores when $K = 4096$. As a comparison, in the same setting SC takes over 4 hours to encode when $K = 4096$.

In Table 4 we further compare with other methods for classification [4]. The VQ method, implemented in [4], is the bag-of-words method [26] using SPM [18]. It is slower than PSC and is not as accurate. The Locality-constrained Linear Coding (LLC) [28] encodes a vector by finding the $n$ nearest codewords and solves a least square problem on them. The implementation in [4] sets $n$ as 5. The main cost is in the nearest neighbor search. The Fisher Vector (FV) [24] is based on a Gaussian-Mixture-Model (GMM) as the codebook. It can generate high-dimensional (e.g., 40960-d) codes using a small codebook (e.g., $K=256$). Table 4 shows PSC performs at least comparably good as LLC and FV, but is faster than these methods.

Table 4 also shows SC is a competitive method given sufficiently large codebooks. So its power is mostly limited by the intolerable running time in practice.

We also evaluate different ways of subspace decomposition in PSC. In the case of PQ, the decomposition is important for the quality [14, 9, 10, 22]. In our experiments, the vector $x$ is arranged as the so-called “natural” order [14] of SIFT. So PSC decomposes the spatial bins of a SIFT vector into upper/lower parts (horizontal split). Alternatively, we can decompose the spatial bins into left/right parts (vertical split). We also test the “structural” order [14] that decomposes the orientation bins of a SIFT vector. Table 5 shows the result of PSC using $K=4096$. We see that the two natural orders perform similarly, and the structural order is slightly worse. In the other part of this paper we use the horizontal natural order.

### 4.3. For Image Retrieval

Image retrieval [15] is a scenario related to image classification but has different concerns. Image retrieval focuses...
Table 4: The comparisons of various methods in Caltech101. The dimensionality is the size of a code. The Fisher Vector (FV) uses \(K=256\) codebook, but the dimensionality of its code is \(2Kd\) where \(d=80\) due to PCA. All the methods here use linear kernels except VQ uses Chi-squared kernels [4]. The time (in seconds) is the average encoding time per image, not including SIFT extraction.

<table>
<thead>
<tr>
<th>method</th>
<th>dimensionality</th>
<th>accuracy (%)</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>4096</td>
<td>77.91</td>
<td>10.5</td>
</tr>
<tr>
<td>SC</td>
<td>16384</td>
<td>78.55</td>
<td>14.6</td>
</tr>
<tr>
<td>PSC</td>
<td>4096</td>
<td>76.71</td>
<td>0.45</td>
</tr>
<tr>
<td>PSC</td>
<td>16384</td>
<td>78.02</td>
<td>0.65</td>
</tr>
<tr>
<td>VQ [4]</td>
<td>4000</td>
<td>74.41</td>
<td>1.94</td>
</tr>
<tr>
<td>VQ [4]</td>
<td>8000</td>
<td>74.23</td>
<td>2.18</td>
</tr>
<tr>
<td>LLC [28]</td>
<td>4000</td>
<td>76.15</td>
<td>2.41</td>
</tr>
<tr>
<td>LLC [28]</td>
<td>8000</td>
<td>76.95</td>
<td>2.65</td>
</tr>
<tr>
<td>FV [24]</td>
<td>40960</td>
<td>77.78</td>
<td>2.12</td>
</tr>
</tbody>
</table>

Table 5: The mean accuracy in Caltech101 of different subspace decompositions in PSC. \(K=4096\).

<table>
<thead>
<tr>
<th>natural (hor)</th>
<th>natural (ver)</th>
<th>structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.71</td>
<td>76.58</td>
<td>76.04</td>
</tr>
</tbody>
</table>

Table 6: The mAP in the Holiday set. The dimensionality is the size of a code before PCA-whitening. For VLAD, the codebook size \(K\) is 32, 64, and 128, and its code size is \(Kd\) \((d=128)\). The mAP are evaluated after PCA-whitening that compresses the representations into 128-D.

<table>
<thead>
<tr>
<th>mAP (%) in Holiday</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensionality</td>
</tr>
<tr>
<td>4K</td>
</tr>
<tr>
<td>8K</td>
</tr>
<tr>
<td>16K</td>
</tr>
</tbody>
</table>

Table 7: The encoding time (in seconds) per image in the Holiday set. The time excludes the SIFT extraction.

<table>
<thead>
<tr>
<th>encoding time in Holiday</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensionality</td>
</tr>
<tr>
<td>4K</td>
</tr>
<tr>
<td>8K</td>
</tr>
<tr>
<td>16K</td>
</tr>
</tbody>
</table>

5. Limitations and Future Work

As discussed, the quality of PSC depends on the subspace decomposition. In the case of PQ, recent studies [22, 9, 10] have shown that the decomposition can be optimized as an orthogonal projection. However, it is more challenging to optimize the decomposition of PSC, because the \(l_1\) term is not invariant to orthogonal projections. This can be a future topic.

In this paper we only consider the case of two subspaces. It has already provided good speed up by reducing the complexity from \(O(K)\) to \(O(\sqrt{K})\). But it would be interesting to study the case of more subspaces \((m > 2)\). Though the APSC algorithm can be simply generalized by using \(\lambda/m\) in each subproblem, the IPSC algorithm is not applicable for \(m > 2\). We will study this in the future.

We have proved that the IPSC algorithm can produce a globally optimal solution if \(\lambda_1^*\) exists. In case \(\lambda_1^*\) does not exist, the global optimum cannot be simply achieved from the two separable problems. Although we have not seen any observable impact in the experiments in this paper, it is still an open question for future studies.

The applications studied in this paper all involve the pooling operations. Our method may benefit from this scenario, because in this case the statistical accuracy is concerned. In other applications when the individual accuracy is of particular importance (e.g., image super-resolution [31]), the quality of our method needs further verifications.

The essence of PSC is the \(O(\sqrt{K})\) complexity. This is in contrast to any \(O(K)\) SC algorithm. Nevertheless, PSC needs to solve two smaller sparse coding subproblems, each of which is still based on SC algorithms. The future ad-
vance of the fast SC algorithms could further improve the subproblem speed of PSC.

References


A. Appendix

Theorem A.1. If $\lambda_1^*$ exists, then $y$ in (13) gives a globally optimal solution to the PSC problem in (4).

Proof. Assume there were a solution $y'$ that leads to a smaller objective value in (4). We can compute its marginal sums $u_1'$ and $u_2'$ as in (7). Substituting $u_1'$ and $u_2'$ into (10) with the given $\lambda_1 = \lambda_1^*$, if the assumption were true, we have $|x_1 - A_1 u_1'|^2 + \lambda_1 |u_1'| + |x_2 - A_2 u_2'|^2 + \lambda_2 |u_2'| < |x_1 - A_1 u_1|^2 + \lambda_1 |u_1| + |x_2 - A_2 u_2|^2 + \lambda_2 |u_2|$. On the other hand, because $u_1$ and $u_2$ are the minimizers to the two separate subproblems in (11), the right-hand side of the above inequality is no greater than the left-hand side, a contradiction.

Theorem A.2. Assume $u = \min_{u' > 0} |x - Au|^2 + \lambda |u| and u' = \min_{u' > 0} |x - Au'|^2 + \lambda' |u'|. If $\lambda' > \lambda$, then $|u'| < |u|$. Denote $\Delta\lambda = \lambda' - \lambda > 0$. Then $|x - Au'|^2 + \lambda' |u'| = |x - Au'|^2 + \lambda' |u'| + \Delta\lambda |u'| > |x - Au'|^2 + \lambda |u| + \Delta\lambda |u'| > |x - Au|^2 + \lambda' |u'|$. So $u$ is a better minimizer than $u'$ in the $\lambda'$-problem, a contradiction.