Double-Bit Quantization for Hashing

Weihao Kong and **Wu-Jun Li**
Shanghai Key Laboratory of Scalable Computing and Systems
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
{kongweihao,liwujun}@cs.sjtu.edu.cn

Abstract

Hashing, which tries to learn similarity-preserving binary codes for data representation, has been widely used for efficient nearest neighbor search in massive databases due to its fast query speed and low storage cost. Because it is NP hard to directly compute the best binary codes for a given data set, mainstream hashing methods typically adopt a two-stage strategy. In the first stage, several projected dimensions of real values are generated. Then in the second stage, the real values will be quantized into binary codes by thresholding. Currently, most existing methods use *one single bit* to quantize each projected dimension. One problem with this single-bit quantization (SBQ) is that the threshold typically lies in the region of the highest point density and consequently a lot of neighboring points close to the threshold will be hashed to totally different bits, which is unexpected according to the principle of hashing. In this paper, we propose a novel quantization strategy, called *double-bit quantization* (DBQ), to solve the problem of SBQ. The basic idea of DBQ is to quantize each projected dimension into double bits with adaptively learned thresholds. Extensive experiments on two real data sets show that our DBQ strategy can significantly outperform traditional SBQ strategy for hashing.

Introduction

With the explosive growth of data on the Internet, there has been increasing interest in approximate nearest neighbor (ANN) search in massive data sets. Common approaches for efficient ANN search are based on similarity-preserving hashing techniques (Gionis, Indyk, and Motwani 1999; Andoni and Indyk 2008) which encode similar points in the original space into close binary codes in the hashcode space. Most methods use the Hamming distance to measure the closeness between points in the hashcode space. By using hashing codes, we can achieve constant or sub-linear search time complexity (Torralba, Fergus, and Weiss 2008; Liu et al. 2011). Furthermore, the storage needed to store the binary codes will be greatly decreased. For example, if we encode each point with 128 bits, we can store a data set of 1 million points with only 16M memory. Hence, hashing provides a very effective and efficient way to perform ANN for massive data sets, and many hashing methods have been proposed by researchers from different research communities. The existing hashing methods can be mainly divided into two categories (Gong and Lazebnik 2011; Liu et al. 2011; 2012): data-independent methods and data-dependent methods.

Representative data-independent methods include locality-sensitive hashing (LSH) (Gionis, Indyk, and Motwani 1999; Andoni and Indyk 2008) and its extensions (Datar et al. 2004; Kulis and Grauman 2009; Kulis, Jain, and Grauman 2009), and shift invariant kernel hashing (SIKH) (Raginsky and Lazebnik 2009). LSH and its extensions use simple random projections which are independent of the training data for hash functions. SIKH chooses projection vectors similar to those of LSH, but SIKH uses a shifted cosine function to generate hash values. Both LSH and SIKH have an important property that points with high similarity will have high probability to be mapped to the same hashcodes. Compared with the data-dependent methods, data-independent methods need longer codes to achieve satisfactory performance (Gong and Lazebnik 2011), which will be less efficient due to the higher storage and computational cost.

Considering the shortcomings of data-independent methods, more and more recent works have focused on data-dependent methods whose hash functions are learned from the training data. Semantic hashing (Salakhutdinov and Hinton 2007; 2009) adopts a deep generative model to learn the hash functions. Spectral hashing (SH) (Weiss, Torralba, and Fergus 2008) uses spectral graph partitioning for hashing with the graph constructed from the data similarity relationships. Binary reconstruction embedding (BRE) (Kulis and Darrell 2009) learns the hash functions by explicitly minimizing the reconstruction error between the original distances and the Hamming distances of the corresponding binary codes. Semi-supervise hashing (SSH) (Wang, Kumar, and Chang 2010) exploits some labeled data to help hash function learning. Selt-taught hashing (Zhang et al. 2010) uses supervised learning algorithms for hashing based on self-labeled data. Composite hashing (Zhang, Wang, and Si 2011) integrates multiple information sources for hashing. Minimal loss hashing (MLH) (Norouzi and Fleet 2011) formulates the hashing problem as a structured prediction problem. Both accuracy and time are jointly optimized to

learn the hash functions in (He et al. 2011). One of the most recent data-dependent methods is iterative quantization (ITQ) (Gong and Lazebnik 2011) which finds an orthogonal rotation matrix to refine the initial projection matrix learned by principal component analysis (PCA) so that the quantization error of mapping the data to the vertices of binary hypercube is minimized. It outperforms most other state-of-the-art methods with relatively short codes.

Because it is NP hard to directly compute the best binary codes for a given data set (Weiss, Torralba, and Fergus 2008), both data-independent and data-dependent hashing methods typically adopt a two-stage strategy. In the first stage, several projected dimensions of real values are generated. Then in the second stage, the real values will be quantized into binary codes by thresholding. Currently, most existing methods use *one single bit* to quantize each projected dimension. More specifically, given a point $\mathbf{x}$, each projected dimension i will be associated with a real-valued *projection function* $f_i(\mathbf{x})$. The ith hash bit of $\mathbf{x}$ will be 1 if $f_i(\mathbf{x}) \geq \theta$. Otherwise, it will be 0. One problem with this single-bit quantization (SBQ) is that the threshold θ (0 for most cases if the data are zero centered) typically lies in the region of the highest point density and consequently a lot of neighboring points close to the threshold might be hashed to totally different bits, which is unexpected according to the principle of hashing. Figure 1 illustrates an example distribution of the real values before thresholding which is computed by PCA[1]. We can find that point "B" and point "C" in Figure 1(a) which lie in the region of the highest density will be quantized into 0 and 1 respectively although they are very close to each other. On the contrary, point "A" and point "B" will be quantized into the same code 0 although they are far away from each other. Because a lot of points will lie close to the threshold, it is very unreasonable to adopt this kind of SBQ strategy for hashing.

To the best of our knowledge, only one existing method, called AGH (Liu et al. 2011), has found this problem of SBQ and proposed a new quantization method called hierarchical hashing (HH) to solve it. The basic idea of HH is to use three thresholds to divide the real values of each dimension into four regions, and encode each dimension with double bits. However, for any projected dimension, the Hamming distance between the two farthest points is the same as that between two relatively close points, which is unreasonable. Furthermore, although the HH strategy can achieve very promising performance when combined with AGH projection functions (Liu et al. 2011), it is still unclear whether HH will be truly better than SBQ when it is combined with other projection functions.

In this paper, we clearly claim that using double bits *with adaptively learned thresholds* to quantize each projected dimension can completely solve the problem of SBQ. The result is our novel quantization strategy called *double-bit quantization* (DBQ). Extensive experiments on real data sets demonstrate that our DBQ can significantly outperform SBQ and HH.

[1]Distribution of real-valued points computed by other hashing methods, such as SH and ITQ, are similar.

Problem Definition

Given a set of n data points $\mathcal{S} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$ with $\mathbf{x}_i \in \mathbb{R}^d$, the goal of hashing is to learn a mapping to encode point $\mathbf{x}_i$ with a binary string $\mathbf{y}_i \in \{0,1\}^c$, where c denotes the code size. To achieve the similarity-preserving property, we require close points in the original space $\mathbb{R}^d$ to have similar binary codes in the code space $\{0,1\}^c$. To get the c-bit codes, we need c binary hash functions $\{h_k(\cdot)\}_{k=1}^c$. Then the binary code can be computed as $\mathbf{y}_i = [h_1(\mathbf{x}_i), h_2(\mathbf{x}_i), \cdots, h_c(\mathbf{x}_i)]^T$. Most hashing algorithms adopt the following two-stage strategy:

- In the first stage, c real-valued functions $\{f_k(\cdot)\}_{k=1}^c$ are used to generate an intermediate vector $\mathbf{z}_i = [f_1(\mathbf{x}_i), f_2(\mathbf{x}_i), \cdots, f_c(\mathbf{x}_i)]^T$, where $\mathbf{z}_i \in \mathbb{R}^c$. These real-valued functions are often called *projection functions* (Andoni and Indyk 2008; Wang, Kumar, and Chang 2010; Gong and Lazebnik 2011), and each function corresponds to one of the c *projected dimensions*;
- In the second stage, the real-valued vector $\mathbf{z}_i$ is encoded into binary vector $\mathbf{y}_i$, typically by thresholding. When the data have been normalized to have zero mean which is adopted by most methods, a common encoding approach is to use function $sgn(x)$, where $sgn(x) = 1$ if $x \geq 0$ and 0 otherwise. For a matrix or a vector, $sgn(\cdot)$ will denote the result of element-wise application of the above function. Hence, let $\mathbf{y}_i = sgn(\mathbf{z}_i)$, we can get the binary code of $\mathbf{x}_i$. This also means that $h_k(\mathbf{x}_i) = sgn(f_k(\mathbf{x}_i))$.

We can see that the above $sgn(\cdot)$ function actually quantizes each projected dimension into *one single bit* with the threshold 0. As stated in the Introduction section, this SBQ strategy is unreasonable, which motivates the DBQ work of this paper.

Double-Bit Quantization

This section will introduce our double-bit quantization (DBQ) in detail. First, we will describe the motivation of DBQ based on observation from real data. Then, the adaptive threshold learning algorithm for DBQ will be proposed. Finally, we will do some qualitative analysis and discussion about the performance of DBQ.

Observation and Motivation

Figure 1 illustrates the point distribution (histogram) of the real values before thresholding on one of the projected dimensions computed by PCA on *22K LabelMe* data set (Torralba, Fergus, and Weiss 2008) which will be used in our experiments. It clearly reveals that the point density is highest near the mean, which is zero here. Note that unless otherwise stated, we assume the data have been normalized to have zero mean, which is a typical choice by existing methods.

The popular coding strategy SBQ which adopts zero as the threshold is shown in Figure 1(a). Due to the thresholding, the intrinsic neighboring structure in the original space is destroyed. For example, points A, B, C, and D are four points sampled from the X-axis of the point distribution

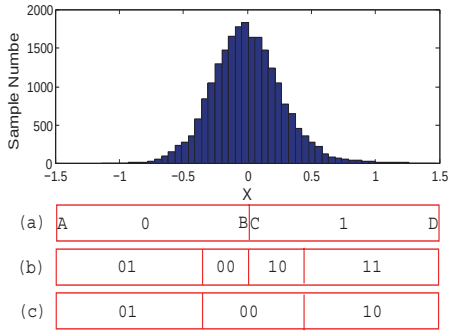


Figure 1: Point distribution of the real values computed by PCA on *22K LabelMe* data set, and different coding results based on the distribution: (a) single-bit quantization (SBQ); (b) hierarchical hashing (HH); (c) double-bit quantization (DBQ).

graph. After SBQ, points A and B, two distant and almost irrelevant points, receive the same code 0 in this dimension. However, B and C, two points which are extremely close in the original space, get totally different codes (0 for B, and 1 for C). Because the threshold zero lies in the densest region, the occurrence probability of the cases like B and C is very high. Hence, it is obvious that SBQ is not very reasonable for coding.

The HH strategy (Liu et al. 2011) is shown in Figure 1(b). Besides the threshold *zero* which has been shown to be a bad choice, HH uses two other thresholds to divide the whole dimension into four regions, and encode each region with double bits. Note that the thresholds are shown in vertical lines in Figure 1(b). If we use $d(A, B)$ to denote the Hamming distance between A and B, we can find that $d(A, D) = d(A, B) = d(C, D) = d(B, C) = 1$ for HH, which is obviously not reasonable.

In fact, if we adopt double bits to encode four regions like those in Figure 1(b), the neighboring structure will be destroyed no matter how we encode the four regions. That is to say, no matter how we assign the four codes ('00','01','10','11') to the four regions, we cannot get any result which can preserve the neighboring structure. This result is caused by the limitation of Hamming distance. More specifically, the largest Hamming distance between 2-bit codes is 2. However, to keep the relative distances between 4 different points, the largest Hamming distance should be at least 3. Hence, no matter how we choose the 2-bit codes for the four regions, we cannot get any neighborhood-preserving result.

In this paper, DBQ is proposed to preserve the neighboring structure by omitting the '11' code, which is shown in Figure 1(c). More specifically, we find two thresholds which do not lie in the densest region to divide the dimension into three regions, and then use double bits to code. With our DBQ code, $d(A, D) = 2$, $d(A, B) = d(C, D) = 1$, and $d(B, C) = 0$, which is obviously reasonable to preserve the similarity relationships in the original space. Please note that the neighboring structure near the thresholds can still be destroyed in DBQ. But we can design some *adaptive threshold learning* algorithm to push the thresholds far way from the dense regions, and solve the problems of SBQ and HH.

Adaptive Threshold Learning

Now we describe how to adaptively learn the optimal thresholds from data. To find the reasonable thresholds, we want the neighboring structure in the original space to be kept as much as possible. The equivalent goal is to make the points in each region as similar as possible.

Let a denote the left threshold, b denote the right threshold and $a < b$, S denote real values of the whole point set on one projected dimension, S_1, S_2, S_3 denote the subsets divided by the thresholds, i.e., $S_1 = \{x| - \infty < x \leq a, x \in S\}$, $S_2 = \{x|a < x \leq b, x \in S\}$, $S_3 = \{x|b < x < \infty, x \in S\}$. Our goal is to find a and b to minimize the following objective function:

$$E = \sum_{x \in S_1} (x - \mu_1)^2 + \sum_{x \in S_2} (x - \mu_2)^2 + \sum_{x \in S_3} (x - \mu_3)^2,$$

where μ_i is the mean of the points in S_i.

As we have discussed above, cutting off right on 0 is not a wise way as the densest region is right there. So we set μ_2 to be 0, which means that $a < 0$ and $b > 0$. Then E can be calculated as follows:

$$\begin{aligned} E &= \sum_{x \in S} x^2 - 2 \sum_{x \in S_1} x\mu_1 + \sum_{x \in S_1} \mu_1^2 - 2 \sum_{x \in S_3} x\mu_3 + \sum_{x \in S_3} \mu_3^2 \\ &= \sum_{x \in S} x^2 - |S_1|\mu_1^2 - |S_3|\mu_3^2 \\ &= \sum_{x \in S} x^2 - \frac{(\sum_{x \in S_1} x)^2}{|S_1|} - \frac{(\sum_{x \in S_3} x)^2}{|S_3|}, \end{aligned}$$

where $|S|$ denotes the number of elements in set S.

Because $\sum_{x \in S} x^2$ is a constant, minimizing E equals to maximizing:

$$F = \frac{(\sum_{x \in S_1} x)^2}{|S_1|} + \frac{(\sum_{x \in S_3} x)^2}{|S_3|}$$
$$subject \quad to: \quad \mu_2 = 0.$$

Algorithm 1 outlines the procedure to learn the thresholds, where $sum(S)$ denotes the summation of all points in set S.

The basic idea of our algorithm is to expand S_2 from empty set by moving one point from either S_1 or S_3 each time while simultaneously keeping $sum(S_2)$ close to 0. After all the elements in initial S_1 and S_3 have been moved to S_2, all possible candidate thresholds (points in S) have been checked, and those achieving the largest F have been recorded in a and b. After we have sorted the points in the initial S_1 and S_3, the *while loop* is just an one-time scan of all the points, and hence the total number of operations in the *while loop* is just n where n is the number of points in S. Each operation is of constant time complexity if we keep $sum(S_1), sum(S_2), sum(S_3)$ in memory. Hence, the most time-consuming part in Algorithm 1 is to sort the initial S_1 and S_3, the time complexity of which is $O(n \log n)$.

After we have learned a and b, we can use them to divide the whole set into S_1, S_2 and S_3, and then use the DBQ in

Algorithm 1 The algorithm to adaptively learn the thresholds for DBQ.

Input: The whole point set S
Initialize with
$S_1 \leftarrow \{x| -\infty < x \leq 0, x \in S\}$
$S_2 \leftarrow \emptyset$
$S_3 \leftarrow \{x|0 < x < +\infty, x \in S\}$
$max \leftarrow 0$
sort the points in S_1
sort the points in S_3
while $S_1 \neq \emptyset$ or $S_3 \neq \emptyset$ **do**
  if $sum(S_2) \leq 0$ **then**
    move the smallest point in S_3 to S_2
  else
    move the largest point in S_1 to S_2
  end if
  compute F
  if $F > max$ **then**
    set a to be the largest point in S_1, and b to be the largest point in S_2
    $max \leftarrow F$
  end if
end while

Figure 1(c) to quantize the points in these subsets into 01, 00, 10, respectively.

Discussion

Let us analyze the expected performance of our DBQ. The first advantage of DBQ is about accuracy. From Figure 1 and the corresponding analysis, it is expected that DBQ will achieve better accuracy than SBQ and HH because DBQ can better preserve the similarity relationships between points. This will be verified by our experimental results.

The second advantage of DBQ is on time complexity, including both coding (training) time and query time. For a c-bit DBQ, we only need to generate $c/2$ projected dimensions while SBQ need c projected dimensions. For most methods, the projection step is the most time-consuming step. Although some extra cost is needed to adaptively learn the thresholds for DBQ, this extra computation is just sorting and an one-time scan of the training points which are actually very fast. Hence, to get the same size of code, the overall coding time complexity of DBQ will be lower than SBQ, which will also be verified in our experiment.

As for the query time, similar to coding time, the number of projection operations for DBQ is only half of that for SBQ. Hence, it is expected that the query speed of DBQ will be faster than SBQ. The query time of DBQ is similar to that of HH because HH also uses half of the number of projection operations for SBQ. Furthermore, if we use inverted index or hash table to accelerate searching, ordinary c-bit SBQ or HH coding would have 2^c possible entries in the hash table. As DBQ does not allow code '11', the possible number of entries for DBQ is $3^{c/2}$. This difference is actually very significant. Let r denote the ratio between the number of entries for DBQ and that for SBQ (or HH). When $c = 32, 64, 256,$ r will be 1%, 0.01%, and 10^{-16}, respectively. Much less entries will gain higher collision probability (improve recall), faster query speed and less storage. In fact, the number of possible entries of c bits of DBQ code only equals to that of $\frac{c \log_2 3}{2} \approx 0.79c$ bits of ordinary SBQ or HH code.

Experiment

Data Sets

We evaluate our methods on two widely used data sets, CIFAR (Krizhevsky 2009) and LabelMe (Torralba, Fergus, and Weiss 2008).

The CIFAR data set (Krizhevsky 2009) includes different versions. The version we use is *CIFAR-10*, which consists of 60,000 images. These images are manually labeled into 10 classes, which are *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*. The size of each image is 32×32 pixels. We represent them with 512-dimensional gray-scale GIST descriptors (Oliva and Torralba 2001).

The second data set is *22K LabelMe* used in (Torralba, Fergus, and Weiss 2008; Norouzi and Fleet 2011). It consists of 22,019 images. We represent each image with 512-dimensional GIST descriptors (Oliva and Torralba 2001).

Evaluation Protocols and Baseline Methods

As the protocols widely used in recent papers (Weiss, Torralba, and Fergus 2008; Raginsky and Lazebnik 2009; Gong and Lazebnik 2011), Euclidean neighbors in the original space are considered as ground truth. More specifically, a threshold of the average distance to the 50th nearest neighbor is used to define whether a point is a true positive or not. Based on the Euclidean ground truth, we compute the precision-recall curve and the mean average precision (mAP) (Liu et al. 2011; Gong and Lazebnik 2011). For all experiments, we randomly select 1000 points as queries, and leave the rest as training set to learn the hash functions. All the experimental results are averaged over 10 random training/test partitions.

Our DBQ can be used to replace the SBQ stage for any existing hashing method to get a new version. In this paper, we just select the most representative methods for evaluation, which contain SH (Weiss, Torralba, and Fergus 2008), PCA (Gong and Lazebnik 2011), ITQ (Gong and Lazebnik 2011), LSH (Andoni and Indyk 2008), and SIKH (Raginsky and Lazebnik 2009). SH, PCA, and ITQ are data-dependent methods, while LSH and SIKH are data-independent methods. By adopting different quantization methods, we can get different versions of a specific hashing method. Let's take SH as an example. "SH-SBQ" denotes the original SH method based on single-bit quantization, "SH-HH" denotes the combination of SH projection with HH quantization (Liu et al. 2011), and "SH-DBQ" denotes the method combining the SH projection with double-bit quantization. Please note that threshold optimization techniques in (Liu et al. 2011) for the two non-zero thresholds in HH can not be used for the above five methods. In our experiments, we just use the same thresholds as those in DBQ. For all the evaluated methods, we use the source codes provided by the authors. For ITQ, we set the iteration number to be 100. To run SIKH, we

Table 1: mAP on LabelMe data set. The best mAP among SBQ, HH and DBQ under the same setting is shown in bold face.

# bits	32			64			128			256		
	SBQ	HH	DBQ	SBQ	HH	DBQ	SBQ	HH	DBQ	SBQ	HH	DBQ
ITQ	0.2926	0.2592	**0.3079**	0.3413	0.3487	**0.4002**	0.3675	0.4032	**0.4650**	0.3846	0.4251	**0.4998**
SH	0.0859	0.1329	**0.1815**	0.1071	0.1768	**0.2649**	0.1730	0.2034	**0.3403**	0.2140	0.2468	**0.3468**
PCA	0.0535	0.1009	**0.1563**	0.0417	0.1034	**0.1822**	0.0323	0.1083	**0.1748**	0.0245	0.1103	**0.1499**
LSH	**0.1657**	0.105	0.12272	**0.2594**	0.2089	0.2577	0.3579	0.3311	**0.4055**	0.4158	0.4359	**0.5154**
SIKH	0.0590	0.0712	**0.0772**	0.1132	0.1514	**0.1737**	0.2792	0.3147	**0.3436**	0.4759	0.5055	**0.5325**

use a Gaussian kernel and set the bandwidth to the average distance of the 50th nearest neighbor, which is the same as that in (Raginsky and Lazebnik 2009). All experiments are conducted on our workstation with Intel(R) Xeon(R) CPU X7560@2.27GHz and 64G memory.

Accuracy

Table 1 and Table 2 show the mAP results for different methods with different code sizes on *22K LabelMe* and *CIFAR-10*, respectively. Each entry in the Tables denotes the mAP of a combination of a hashing method with a quantization method under a specific code size. For example, the value "0.2926" in the upper left corner of Table 1 denotes the mAP of ITQ-SBQ with the code size 32. The best mAP among SBQ, HH and DBQ under the same setting is shown in bold face. For example, in Table 1, when the code size is 32 and the hashing method is ITQ, the mAP of DBQ (0.3079) is the best one compared with those of SBQ (0.2926) and HH (0.2592). Hence, the value 0.3079 will be in bold face. From Table 1 and Table 2, we can find that when the code size is small, the performance of data-independent methods LSH and SIKH is relatively poor and ITQ achieves the best performance under most settings especially for those with small code size, which verifies the claims made in existing work (Raginsky and Lazebnik 2009; Gong and Lazebnik 2011). This also indicates that our implementations are correct.

Our DBQ method achieves the best performance under most settings, and it outperforms HH under all settings except the ITQ with 32 bits on the CIFAR-10 data set. This implies that our DBQ with adaptively learned thresholds is very effective. The exceptional settings where our DBQ method is outperformed by SBQ are LSH and ITQ with code size smaller than 64. One possible reason might be from the fact that the c-bit code in DBQ only utilizes $c/2$ projected dimensions while c projected dimensions are utilized in SBQ. When the code size is too small, the useful information for hashing is also very weak, especially for data-independent methods like LSH. Hence, even if our DBQ can find the best way to encode, the limited information kept in the projected dimensions can not guarantee a good performance. Fortunately, when the code size is 64, the worst performance of DBQ is still comparable with that of SBQ. When the code size is 128 or larger, the performance of DBQ will significantly outperform SBQ under any setting. As stated in the Introduction session, the storage cost is still very low when the code size is 128. Hence, the setting with code size 128 can be seen as a good tradeoff between accuracy and storage cost in real systems. Please note that although we argue that our method can achieve the best performance with code size larger than 64, the overall performance of DBQ is still the best under most settings with small code size such as the case of 32 bits.

Figure 2, Figure 3 and Figure 4 show precision-recall curves for ITQ, SH and LSH with different code sizes on the 22K LabelMe data set. The relative performance among SBQ, HH, and DBQ in the precision-recall curves for PCA and SIKH is similar to that for ITQ. We do not show these curves due to space limitation. From Figure 2, Figure 3 and Figure 4, it is clear that our DBQ method significantly outperforms SBQ and HH under most settings.

Computational Cost

Table 3 shows the training time on CIFAR-10. Although some extra cost is needed to adaptively learn the thresholds for DBQ, this extra computation is actually very fast. Because the number of projected dimensions for DBQ is only half of that for SBQ, the training of DBQ is still faster than SBQ. This can be seen from Table 3. For query time, DBQ is also faster than SBQ, which has been analyzed above. Due to space limitation, we omit the detailed query time comparison here.

Table 3: Training time on CIFAR-10 date set (in seconds).

# bits	32		64		256	
	SBQ	DBQ	SBQ	DBQ	SBQ	DBQ
ITQ	14.48	8.46	29.95	14.12	254.14	80.09
SIKH	1.76	1.46	2.00	1.57	4.55	2.87
LSH	0.30	0.19	0.53	0.30	1.80	0.95
SH	5.60	3.74	11.72	5.57	133.50	37.57
PCA	4.03	3.92	4.31	3.99	5.86	4.55

Conclusion

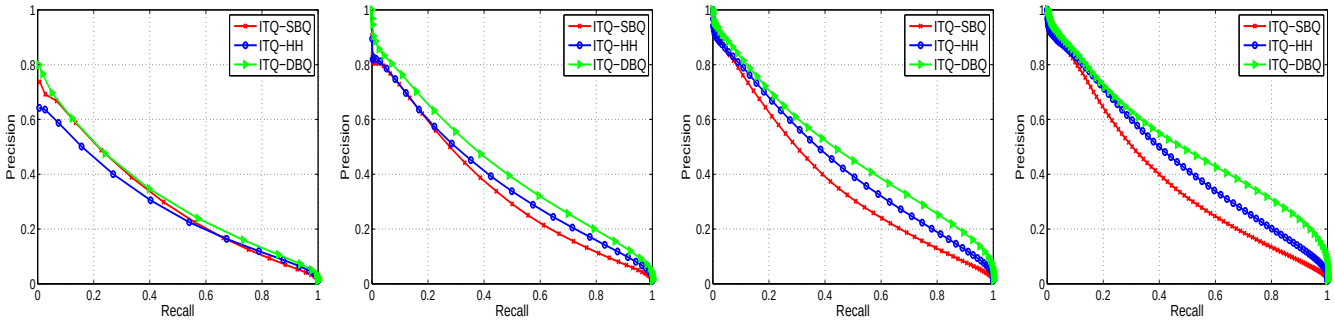
The SBQ strategy adopted by most existing hashing methods will destroy the neighboring structure in the original space, which violates the principle of hashing. In this paper, we propose a novel quantization strategy called DBQ to effectively preserve the neighboring structure among data. Extensive experiments on real data sets demonstrate that our DBQ can achieve much better accuracy with lower computational cost than SBQ.

Acknowledgments

This work is supported by the NSFC (No. 61100125) and the 863 Program of China (No. 2011AA01A202, No. 2012AA011003). We thank Yunchao Gong and Wei Liu for sharing their codes and providing useful help for our experiments.

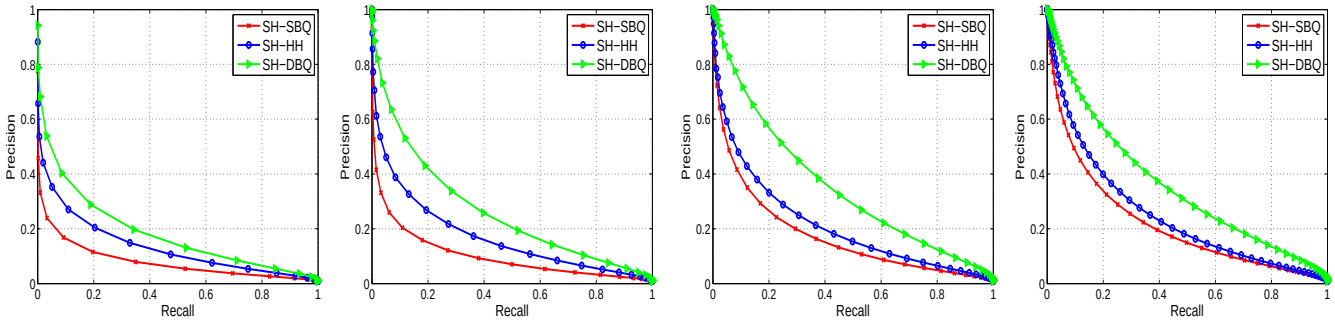
Table 2: mAP on CIFAR-10 data set. The best mAP among SBQ, HH and DBQ under the same setting is shown in bold face.

# bits	32			64			128			256		
	SBQ	HH	DBQ	SBQ	HH	DBQ	SBQ	HH	DBQ	SBQ	HH	DBQ
ITQ	**0.2716**	0.2240	0.2220	0.3293	0.3006	**0.3350**	0.3593	0.3826	**0.4395**	0.3727	0.4140	**0.5221**
SH	0.0511	0.0742	**0.1114**	0.0638	0.0936	**0.1717**	0.0998	0.1209	**0.2501**	0.1324	0.1697	**0.3337**
PCA	0.0357	0.0646	**0.1072**	0.0311	0.0733	**0.1541**	0.0261	0.0835	**0.1966**	0.0217	0.1127	**0.2053**
LSH	**0.1192**	0.0665	0.0660	**0.1882**	0.1457	0.1588	0.2837	0.2601	**0.3153**	0.3480	0.3640	**0.4680**
SIKH	0.0417	0.0359	**0.0466**	0.0953	0.0911	**0.1063**	0.1836	0.1969	**0.2263**	0.3677	0.3601	**0.3975**



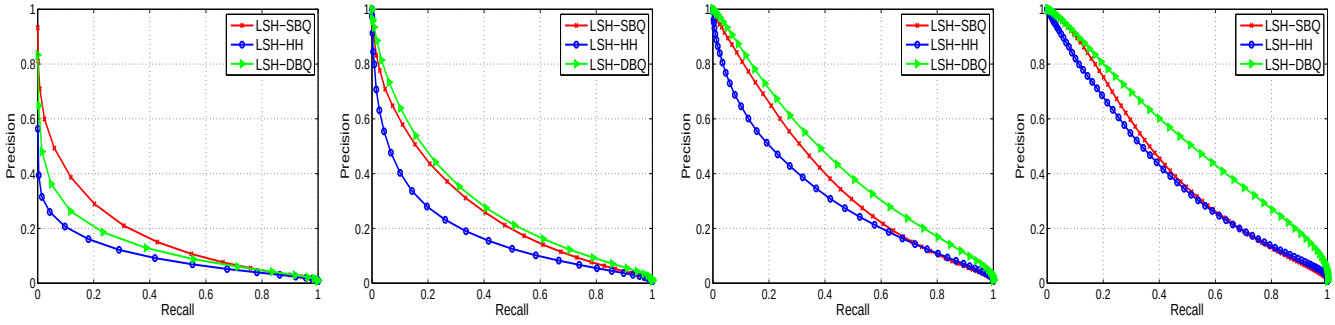
(a) ITQ 32 bits (b) ITQ 64 bits (c) ITQ 128 bits (d) ITQ 256 bits

Figure 2: Precision-recall curve for ITQ on 22K LabelMe data set



(a) SH 32 bits (b) SH 64 bits (c) SH 128 bits (d) SH 256 bits

Figure 3: Precision-recall curve for SH on 22K LabelMe data set



(a) LSH 32 bits (b) LSH 64 bits (c) LSH 128 bits (d) LSH 256 bits

Figure 4: Precision-recall curve for LSH on 22K LabelMe data set

References

Andoni, A., and Indyk, P. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51(1):117–122.

Datar, M.; Immorlica, N.; Indyk, P.; and Mirrokni, V. S. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the ACM Symposium on Computational Geometry*.

Gionis, A.; Indyk, P.; and Motwani, R. 1999. Similarity search in high dimensions via hashing. In *Proceedings of International Conference on Very Large Data Bases*.

Gong, Y., and Lazebnik, S. 2011. Iterative quantization: A procrustean approach to learning binary codes. In *Proceedings of Computer Vision and Pattern Recognition*.

He, J.; Radhakrishnan, R.; Chang, S.-F.; and Bauer, C. 2011. Compact hashing with joint optimization of search accuracy and time. In *Proceedings of Computer Vision and Pattern Recognition*.

Krizhevsky, A. 2009. Learning multiple layers of features from tiny images. Tech report, University of Toronto.

Kulis, B., and Darrell, T. 2009. Learning to hash with binary reconstructive embeddings. In *Proceedings of Neural Information Processing Systems*.

Kulis, B., and Grauman, K. 2009. Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of International Conference on Computer Vision*.

Kulis, B.; Jain, P.; and Grauman, K. 2009. Fast similarity search for learned metrics. *IEEE Trans. Pattern Anal. Mach. Intell.* 31(12):2143–2157.

Liu, W.; Wang, J.; Kumar, S.; and Chang, S.-F. 2011. Hashing with graphs. In *Proceedings of International Conference on Machine Learning*.

Liu, W.; Wang, J.; Ji, R.; Jiang, Y.-G.; and Chang, S.-F. 2012. Supervised hashing with kernels. In *Proceedings of Computer Vision and Pattern Recognition*.

Norouzi, M., and Fleet, D. J. 2011. Minimal loss hashing for compact binary codes. In *Proceedings of International Conference on Machine Learning*.

Oliva, A., and Torralba, A. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision* 42(3):145–175.

Raginsky, M., and Lazebnik, S. 2009. Locality-sensitive binary codes from shift-invariant kernels. In *Proceedings of Neural Information Processing Systems*.

Salakhutdinov, R., and Hinton, G. 2007. Semantic Hashing. In *SIGIR workshop on Information Retrieval and applications of Graphical Models*.

Salakhutdinov, R., and Hinton, G. E. 2009. Semantic hashing. *Int. J. Approx. Reasoning* 50(7):969–978.

Torralba, A.; Fergus, R.; and Weiss, Y. 2008. Small codes and large image databases for recognition. In *Proceedings of Computer Vision and Pattern Recognition*.

Wang, J.; Kumar, S.; and Chang, S.-F. 2010. Sequential projection learning for hashing with compact codes. In *Proceedings of International Conference on Machine Learning*.

Weiss, Y.; Torralba, A.; and Fergus, R. 2008. Spectral hashing. In *Proceedings of Neural Information Processing Systems*.

Zhang, D.; Wang, J.; Cai, D.; and Lu, J. 2010. Self-taught hashing for fast similarity search. In *Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Zhang, D.; Wang, F.; and Si, L. 2011. Composite hashing with multiple information sources. In *Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval*.