Supervised Hashing Using Graph Cuts and Boosted Decision Trees

Guosheng Lin, Chunhua Shen, Anton van den Hengel

Abstract—To build large-scale query-by-example image retrieval systems, embedding image features into a binary Hamming space provides great benefits. Supervised hashing aims to map the original features to compact binary codes that are able to preserve label based similarity in the binary Hamming space. Most existing approaches apply a single form of hash function, and an optimization process which is typically deeply coupled to this specific form. This tight coupling restricts the flexibility of those methods, and can result in complex optimization problems that are difficult to solve. In this work we proffer a flexible yet simple framework that is able to accommodate different types of loss functions and hash functions. The proposed framework allows a number of existing approaches to hashing to be placed in context, and simplifies the development of new problem-specific hashing methods. Our framework decomposes the hashing learning problem into two steps: binary code (hash bit) learning and hash function learning. The first step can typically be formulated as binary quadratic problems, and the second step can be accomplished by training a standard binary classifier. For solving large-scale binary code inference, we show how it is possible to ensure that the binary quadratic problems are submodular such that efficient graph cut methods may be used. To achieve efficiency as well as efficacy on large-scale high-dimensional data, we propose to use boosted decision trees as the hash functions, which are nonlinear, highly descriptive, and are very fast to train and evaluate. Experiments demonstrate that the proposed method significantly outperforms most state-of-the-art methods, especially on high-dimensional data.

Index Terms—Hashing, Binary Codes, Graph Cuts, Decision Trees, Nearest Neighbour Search, Image Retrieval.

1 INTRODUCTION

An explosion in the size of the datasets has been witnessed in the past a few years. It becomes more and more demanding to cope with image datasets with tens of millions of images, in terms of both efficient storage and processing. Hashing methods construct a set of hash functions that map the original features into compact binary codes. Hashing enables fast nearest neighbor search by using look-up tables or Hamming distance based ranking. Moreover, compact binary codes are extremely efficient for large-scale data storage. Example applications include image retrieval ([1], [2]), image matching [3], object detection [4], etc.

Loss functions for learning-based hashing are typically defined on the basis of the Hamming distance or Hamming affinity of similar and dissimilar data pairs. Hamming affinity is calculated by the inner product of two binary codes (a binary code takes a value from $\{-1, 1\}$). Existing methods thus tend to optimize a single form of hash function. The common forms of hash functions are linear perceptron functions (e.g., Minimal Loss Hashing (MLH) [5], Semi-supervised Hashing (SPLH) [2], Iterative Quantization (ITQ) [6], Locality-Sensitive Hashing (LSH) [7]), kernel functions (Supervised Hashing with Kernels (KSH) [8]), eigenfunctions (Spectral Hashing (SPH) [9], Multidimensional Spectral Hashing (MDSH) [10]). The optimization procedure is then coupled with the selected family of hash functions. Different types of hash functions offer a trade-off between testing time and fitting capacity. For example, compared with kernel functions, the simple linear perceptron function is usually much more efficient for evaluation but can have a relatively low accuracy for nearest neighbor search. This coupling often results in a highly non-convex optimization problem which can be very challenging to optimize. As an example, the loss functions in MDSH, KSH and Binary Reconstructive Embeddings (BRE) [11] all take a similar form that aims to minimize the difference between the Hamming affinity (or distance) of data pairs and the ground truth. However, the optimization procedures used are coupled with the form of hash functions (eigenfunctions, kernel functions) and thus different optimization techniques are needed for each.

Our framework, however, is able to accommodate any loss function defined on the Hamming distance/affinity of data pairs, such as the loss functions used in KSH, BRE or MLH. We decompose the learning into two steps: the binary codes inference step and the hash function learning step. We can formulate the optimization problem of any Hamming distance/affinity based loss as binary quadratic problems, hence different types of loss functions are unified into the same optimization problem, which significantly simplifies the optimization. With this decomposition the hash function learning becomes a binary classification problem, hence we can learn various types of hash function, like perceptrons, kernel and decision tree hash functions, by simply training binary

• *Authors are with the School of Computer Science, The University of Adelaide, Australia; and Australian Research Council Centre of Excellence for Robotic Vision;*
Corresponding author: Chunhua Shen (chunhua.shen@adelaide.edu.au);

classifiers.

Many supervised hashing approaches require complex optimization for directly learning hash functions, and hence may only be tractable for small scale training data. In our approach, we propose an efficient graph cut based block search algorithm for solving the large-scale binary code inference problem, thus our method can be easily trained on large-scale datasets.

Recent advances in the feature learning ([12], [13]) show that high-dimensional features are essential for achieving good performance. For example, the dimension of codebook based features is usually in the tens of thousands. Many existing hashing methods become impractically slow when trained on large scale high-dimensional features. Non-linear hash functions, e.g., the kernel hash function employed in KSH, have shown much improved performance over the linear hash function. However, kernel functions can be extremely expensive to evaluate for both training and testing on high-dimensional features. Here we propose to learn decision trees as hash functions for non-linear mapping. Decision trees only involve simple comparison operations, thus they are very efficient to evaluate. Moreover, decision trees are able to work on quantized data without significant performance loss, and hence only consume a small amount of memory for training.

The main contributions of this work are as follows.

- We propose a flexible and efficient hashing framework which is able to incorporate various kinds of loss functions and hash functions.

  We decompose the learning procedure into two steps: binary code inference, and hash function learning. This decomposition simplifies the hash function learning problem into a standard binary classification problem. An arbitrary classifier, such as linear or kernel Support Vector Machines (SVM), boosting, or neural networks, may thus be adopted to learn the hash functions.

  We are able to incorporate various types of loss function in a unified manner. We show that any type of loss function (e.g., the loss functions in KSH, BRE, MLH) defined on Hamming affinity or Hamming distance, can be equivalently converted into a standard quadratic function, thus we can solve a standard binary quadratic problem for binary code inference.
- For binary code inference, we propose sub-modular formulations and an efficient graph cut [14] based block search method for solving large-scale binary code inference.
- We propose to use (ensembles of) decision trees as hash functions for supervised hashing, which can easily deal with a very large number of high-dimensional training data and has the desired non-linear mapping.
- Our method significantly outperforms many existing methods in terms of retrieval accuracy. For high-dimensional data, our method is also orders of magnitude faster for training.

We made the code available at https://bitbucket.org/chhshen/fasthash/.

1.1 Related work

Hashing methods aim to preserve some notion of similarity (or distance) in the Hamming space. These methods can be roughly categorized as being either supervised or unsupervised. Unsupervised hashing methods ([7], [9], [10], [6], [15], [16], [17], [18]) try to preserve the similarity which is often calculated in the original feature space. For example, LSH [7] generates random linear hash functions to approximate cosine similarity; SPH ([9], [10]) learns eigenfunctions that preserve Gaussian affinity; ITQ [6] approximates the Euclidean distance in the Hamming space. Supervised hashing is designed to preserve the label-based similarity ([19], [11], [5], [20], [2], [8], [21], [22]). This might take place, for example, in the case where images from the same category are defined as being semantically similar to each other. Supervised hashing has received increasing attention recently (e.g., KSH [8], BRE [11]). Our method targets supervised hashing. Preliminary results of our work appeared in [23] and [24].

Various optimization techniques are proposed in existing methods. For example, random projection is used in LSH and Kernelized Locality-Sensitive Hashing (KLSH) [25]; spectral graph analysis for exploring the data manifold is used in SPH [9], MDSH [10], STH [26], Hashing with Graphs (AGH) [16], and inductive hashing [18]; vector quantization is used in ITQ [6], and K-means Hashing [15]; kernel methods are used in KSH [8] and KLSH [25]. MLH [5] optimizes a hinge-like loss. The optimization techniques in most existing work are tightly coupled with their loss functions and hash functions. In contrast, our method breaks this coupling and easily incorporates various types of loss function and hash function.

A number of existing hash methods have explicitly or implicitly employed two-step optimization based strategies for hash function learning, like Self-Taught Hashing (STH) [26], MLH [5], Hamming distance metric learning [27], ITQ [6] and angular quantization based binary code learning [28]. However, in these existing methods, the optimization techniques for binary inference and hash function learning are deeply coupled to their specific form of loss function and hash functions, and none of them is as general as our learning framework.

STH [26] explicitly employs a two-step learning scheme for optimizing the Laplacian affinity loss. The Laplacian affinity loss in STH only tries to pull together similar data pairs but does not push away dissimilar data pairs, which may lead to inferior performance [29]. Moreover, STH employs a spectral method for binary code inference, which usually leads to inferior binary solutions due to its loose relaxation. Moreover, the spectral method does not scale well on large training data.

In contrast, we are able to incorporate any hamming distance or affinity based loss function, and propose an efficient graph cut based method for large scale binary code inference.

MLH [5] learns hash functions by optimizing a convex-concave upper-bound of a hinge loss function (or BRE loss function). They need to solve a binary code inference problem during optimization, for which they propose a so-called loss-adjusted inference algorithm. A similar technique is also applied in [27]. The training of ITQ [6] also involves a two-step optimization strategy. ITQ iteratively generates the binary code and learns a rotation matrix by minimizing the quantization error against the binary code. They generate the binary code simply by thresholding.

The problem of similarity search on high-dimensional data is also addressed in [30]. Their method extends vocabulary tree based search methods ([31], [32]) by replacing vocabulary trees with boosted trees. This type of search method represents the image as the evidence of a large number of visual words, which are vectors with thousands or even millions dimensions. Then this visual word representation is fed into an inverted index based search algorithm to output the final retrieval result. Clearly hashing methods are different from these inverted index based search methods. Our method is in the vein of supervised hashing methods: mapping data points into binary codes so that the hamming distance on binary codes reflects the label based similarity.

2 Flexible Two-Step Hashing

Let $\mathcal{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\} \subset \mathbb{R}^d$ denote a set of training points. Label based similarity information is described by an affinity matrix: $\mathbf{Y}$, which is the ground truth for supervised learning. The element in $\mathbf{Y}$: y_{ij} indicates the similarity between two data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$; and $y_{ij} = y_{ji}$. Specifically, $y_{ij} = 1$ if two data points are similar (relevant), $y_{ij} = -1$ if dissimilar (irrelevant) and $y_{ij} = 0$ if the pairwise relation is undefined. We aim to learn a set of hash functions which preserve the label based similarity in the Hamming space. A hash function is denoted by $h(\cdot)$ with binary output: $h(\boldsymbol{x}) \in \{-1, 1\}$. The output of m hash functions is denoted by $\Phi(\boldsymbol{x})$:

$$\Phi(\boldsymbol{x}) = [h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_m(\boldsymbol{x})], \tag{1}$$

which is a m-bit binary vector: $\Phi(\boldsymbol{x}) \in \{-1, 1\}^m$.

The loss function in hashing learning for preserving pairwise similarity relations is typically defined in terms of the Hamming distance or Hamming affinity of data pairs. The Hamming distance between two binary codes is the number of bits taking different values:

$$d_{\mathrm{H}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{r=1}^{m} \delta(h_r(\boldsymbol{x}_i) \neq h_r(\boldsymbol{x}_j)). \tag{2}$$

Here $\delta(\cdot) \in \{0, 1\}$ is an indicator function which outputs 1 if the input is true and 0 otherwise. Generally, the formulation of hashing learning encourages small Hamming distances for similar data pairs and large distances for dissimilar data pairs. Closely related to Hamming distance, the Hamming affinity is calculated by the inner product of two binary codes:

$$s_{\mathrm{H}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{r=1}^{m} h_r(\boldsymbol{x}_i) h_r(\boldsymbol{x}_j). \tag{3}$$

As shown in [8], the Hamming affinity is in one-to-one correspondence with the Hamming distance. We solve the following optimization for hash function learning:

$$\min_{\Phi(\cdot)} \sum_{i=1}^{n} \sum_{j=1}^{n} \delta(y_{ij} \neq 0) L(\Phi(\boldsymbol{x}_i), \Phi(\boldsymbol{x}_j); y_{ij}). \tag{4}$$

Here $\delta(y_{ij} \neq 0) \in \{0, 1\}$ indicates whether the relation between two data points is defined, and $L(\cdot)$ is a loss function that measures how well the binary codes match the similarity ground truth y_{ij}. Various types of loss functions $L(\cdot)$ have been proposed, and will be discussed in detail in the next section. Most existing methods try to directly optimize the objective function in (4) in order to learn the parameters of hash functions ([8], [5], [11], [10]). This inevitably means that the optimization process is tightly coupled to the form of hash functions used, which makes it non-trivial to extend a method to use other types of hash functions. Moreover, this coupling usually results in challenging optimization problems.

As an example, the KSH loss function, which is defined on Hamming affinity, is written as follows:

$$L_{\mathrm{KSH}} = \sum_{i=1}^{n} \sum_{j=1}^{n} \delta(y_{ij} \neq 0) \left[m y_{ij} - \sum_{r=1}^{m} h_r(\boldsymbol{x}_i) h_r(\boldsymbol{x}_j) \right]^2. \tag{5}$$

We use $\delta(y_{ij} \neq 0)$ to prevent undefined pairwise relations from having an impact on the training objective. Intuitively, this optimization encourages the Hamming affinity value of a data pair to be close to the ground truth value. The form of hash function in KSH is the kernel function:

$$h(\boldsymbol{x}) = \mathrm{sign} \left[\sum_{q=1}^{Q} w_q \kappa(\boldsymbol{x}'_q, \boldsymbol{x}) + b \right], \tag{6}$$

in which $\mathcal{X}' = \{\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_Q\}$ are Q support vectors; KSH directly solve the optimizations in (5) for learning the hash functions. If we prefer other forms of hash functions, the optimization of KSH would not be applicable. For example if using the decision-tree hash function which is more suitable for high-dimensional data, it is not clear how to learn decision trees by directly optimizing (5). Moreover, KSH uses a set of predefined support vectors which are randomly sampled from the training set, and it does not have a sparse solution of the weighting parameters. Hence this unsophisticated kernel method would be impracticable for large-scale training and computationally expensive for evaluation.

Here we develop a general and flexible two-step learning framework, which is readily to incorporate various forms of loss functions and hash functions. Basically, partly inspired by STH [26], we decompose the learning procedure into two steps: the first step for binary code inference and the second step for hash function learning. We introduce auxiliary variables $z_{r,i} \in \{-1, 1\}$ as the output of the r-th hash function on $\boldsymbol{x}_i$:

$$z_{r,i} = h_r(\boldsymbol{x}_i). \tag{7}$$

Clearly, $z_{r,i}$ represents the r-th bit of the binary code of the i-th data point. With these auxiliary variables, the problem in (4) can be decomposed into two subproblems:

$$\min_{\mathbf{Z}} \sum_{i=1}^{n}\sum_{j=1}^{n} \delta(y_{ij} \neq 0) L(\boldsymbol{z}_i, \boldsymbol{z}_j; y_{ij}), \tag{8a}$$
$$\text{s.t.} \quad \mathbf{Z} \in \{-1, 1\}^{m \times n}; \tag{8b}$$

and,

$$\min_{\Phi(\cdot)} \sum_{r=1}^{m}\sum_{i=1}^{n} \delta(z_{r,i} = h_r(\boldsymbol{x}_i)). \tag{9}$$

Here $\mathbf{Z}$ is the matrix of m-bit binary codes for all n training data points; $\boldsymbol{z}_i$ is the binary code vector corresponding to i-th data point. $\delta(\cdot)$ is an indicator function. In this way, the hashing learning in (4) now becomes two relatively simpler tasks—solving (8) (Step 1) and (9) (Step 2). Clearly, Step 1 is to solve for binary codes, and Step 2 is to solve simple binary classification problems.

We sequentially solve for one bit at a time conditioning on previous bits. Hence we solve these two steps alternatively, rather than completely separating these two steps. After solving for one bit, the binary codes are updated by applying the learned hash function. Hence the learned hash function is able to influence the binary code inference for the next bit. This bit-wise optimization strategy helps to simplify the optimization, and the error of one learned hash function can be propagated and compensated for when learning the next bit.

In the following sections, we describe how to solve these two steps for one bit. In Sec. 3, we show that any hamming distance or affinity based loss function can be equivalently reformulated as a binary quadratic problem. For binary code inference, we propose a graph cut based block search method for efficiently solving the binary code inference (Sec. 3.2). Later we discuss training different types of hash functions in Sec. 4. Especially we introduce the decision tree hash functions which provide the desirable non-linear mapping and are highly efficient for evaluation (Sec. 4.1). With the proposed efficient binary code inference algorithm, our method is not only flexible, but also capable of large-scale training. We refer to our method as FastHash. The algorithm is shown in Algorithm 1.

Algorithm 1: FastHash (flexible two-step hashing)

```
Input: training data points: $\{\boldsymbol{x}_1, ...\boldsymbol{x}_n\}$; affinity
        matrix: $\mathbf{Y}$; bit length: $m$.
Output: hash functions: $\Phi = [h_1, ..., h_m]$.
1 Initialization: construct blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$ for Block
  GraphCut, Algorithm 2 shows an example;
2 for $r = 1, ..., m$ do
3     Step-1: call Algorithm 3 to solve the binary code
      inference in (13a), obtain binary codes of the
      $r$-th bit;
4     Step-2: solve binary classification in (34) to
      obtain one hash function $h_r$ (e.g., solve linear
      SVM in (36) or boosted tree learning in (39)) ;
5     Update the binary codes of the $r$-th bit by
      applying the learned hash function $h_r$;
```

Algorithm 2: An example for constructing blocks

```
Input: training data points: $\{\boldsymbol{x}_1, ...\boldsymbol{x}_n\}$; affinity
        matrix: $\mathbf{Y}$.
Output: blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$.
1 $\mathcal{V} \leftarrow \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$; $t = 0$;
2 repeat
3     $t = t + 1$; $\mathcal{B}_t \leftarrow \emptyset$;
4     Randomly selected $\boldsymbol{x}_i$ from $\mathcal{V}$;
5     Initialize $\mathcal{U}$ as the joint set of $\mathcal{V}$ and similar
      examples of $\boldsymbol{x}_i$ ;
6     for each $\boldsymbol{x}_j$ in $\mathcal{U}$ do
7         if $\boldsymbol{x}_j$ is not dissimilar with any examples in $\mathcal{B}_t$
          then
8             add $\boldsymbol{x}_j$ to $\mathcal{B}_t$ ;
9             remove $\boldsymbol{x}_j$ from $\mathcal{V}$ ;
10 until $\mathcal{V} = \emptyset$;
```

3 Step 1: Binary Code Inference

When solving for the r-th bit, the binary codes of the previous $(r-1)$ bits are fixed, and the bit length m is set to r. The binary code inference problem is:

$$\min_{\boldsymbol{z}_{(r)} \in \{-1,1\}^n} \sum_{i=1}^{n}\sum_{j=1}^{n} \delta(y_{ij} \neq 0) l_r(z_{r,i}, z_{r,j}; y_{ij}). \tag{10}$$

Here $\boldsymbol{z}_{(r)}$ is the n-dimensional binary code vector we seek. It represents the binary hash codes of the n training data points for the r-th bit. $z_{r,i}$ is the binary code of the i-th data point and the r-th bit. l_r represents the loss function output for the r-th bit, conditioning on previous bits:

$$l_r(z_{r,i}, z_{r,j}; y_{ij}) = L(z_{r,i}, z_{r,j}; \boldsymbol{z}_i^{(r-1)}, \boldsymbol{z}_j^{(r-1)} y_{ij}). \tag{11}$$

Here $\boldsymbol{z}_i^{(r-1)}$ is the binary code vector of the i-th data point in all previous $(r-1)$ bits.

Based on the following proposition, we are able to rewrite the binary code inference problem with any

Algorithm 3: Step-1: Block GraphCut for binary code inference

```
Input: affinity matrix: $\mathbf{Y}$; bit length: $r$;
       blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$; binary codes:
       $\{\boldsymbol{z}_1, ..., \boldsymbol{z}_{r-1}\}$.
Output: binary codes of one bit: $\boldsymbol{z}_r$.
1 repeat
2     Randomly permute all blocks;
3     for each $\mathcal{B}_i$ do
4         Solve the inference in (27a) on $\mathcal{B}_i$ using
          graph cuts;
5 until max iteration is reached;
```

Hamming affinity (or distance) based loss function $L(\cdot)$ into a standard quadratic problem.

Proposition 1: For any loss function $l(z_1, z_2)$ that is defined on a pair of binary input variables $z_1, z_2 \in \{-1, 1\}$ and $l(1,1) = l(-1,-1)$, $l(1,-1) = l(-1,1)$, we can define a quadratic function $g(z_1, z_2)$ that is equal to $l(z_1, z_2)$. We have following equations:

$$\begin{aligned} l(z_1, z_2) &= \frac{1}{2}\Big[z_1 z_2 (l^{(11)} - l^{(-11)}) + l^{(11)} + l^{(-11)}\Big], \\ &= \frac{1}{2} z_1 z_2 (l^{(11)} - l^{(-11)}) + \text{const.} \\ &= g(z_1, z_2). \end{aligned} \tag{12}$$

Here $l^{(11)}, l^{(-11)}$ are constants, $l^{(11)}$ is the loss output on identical input pair: $l^{(11)} = l(1,1)$, and $l^{(-11)}$ is the loss output on distinct input pair: $l^{(-11)} = l(-1,1)$.

Proof: This proposition can be easily proved by exhaustively checking all possible inputs of the loss function. Notice that there are only two possible output values of the loss function. For the input $(z_1 = 1, z_2 = 1)$:

$$\begin{aligned} g(1,1) &= \frac{1}{2}\Big[1 \times 1 \times (l^{(11)} - l^{(-11)}) + l^{(11)} + l^{(-11)}\Big] \\ &= l(1,1), \end{aligned}$$

For the input $(z_1 = -1, z_2 = 1)$:

$$\begin{aligned} g(-1,1) &= \frac{1}{2}\Big[-1 \times 1 \times (l^{(11)} - l^{(-11)}) + l^{(11)} + l^{(-11)}\Big] \\ &= l(-1,1), \end{aligned}$$

The input $(z_1 = -1, z_2 = -1)$ is the same as $(z_1 = 1, z_2 = 1)$ and the input $(z_1 = 1, z_2 = -1)$ is the same as $(z_1 = -1, z_2 = 1)$. In conclusion, the function $l(\cdot,\cdot)$ and $g(\cdot,\cdot)$ have the same output for any possible inputs. □

Any hash loss function $l(\cdot,\cdot)$ which is defined on the Hamming affinity or Hamming distance of data pairs is able to meet the requirement that: $l(1,1) = l(-1,-1), l(1,-1) = l(-1,1)$. Applying this proposition, the optimization of (10) can be equivalently reformulated as:

$$\min_{\boldsymbol{z}_{(r)} \in \{-1,1\}^n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} z_{r,i} z_{r,j}, \tag{13a}$$

$$\text{where, } a_{i,j} = \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}), \tag{13b}$$

$$l_{r,i,j}^{(11)} = l_r(1,1; y_{ij}), \quad l_{r,i,j}^{(-11)} = l_r(-1,1; y_{ij}). \tag{13c}$$

Here $a_{i,j}$ is constant. The above optimization is an unconstrained binary quadratic problem. It can be written in a matrix form:

$$\min_{\boldsymbol{z}_{(r)}} \boldsymbol{z}_{(r)}^\top \mathbf{A} \boldsymbol{z}_{(r)}, \tag{14a}$$

$$\text{s.t. } \boldsymbol{z}_{(r)} \in \{-1,1\}^n. \tag{14b}$$

Here the (i,j)-th element of matrix $\mathbf{A}$ is defined by $a_{i,j}$ in (13b). We have shown that the original optimization in (10) for one bit can be equivalently reformulated as a binary quadratic problem (BQP) in (13a). We discuss algorithms for solving this BQP in the next section.

Here we describe a selection of such loss functions, most of which arise from recent hashing methods. These loss functions are defined on Hamming distance/affinity, thus they are applicable to Proposition 1. Recall that m is the number of bits, $d_{\mathrm{H}}(\cdot,\cdot)$ is the Hamming distance and $\delta(\cdot) \in \{0,1\}$ is an indicator function.

FastH-KSH The KSH loss function is based on Hamming affinity. MDSH also uses a similar form of loss function (weighted Hamming affinity instead).

$$L_{\mathrm{KSH}}(\boldsymbol{z}_i, \boldsymbol{z}_j) = (m y_{ij} - \boldsymbol{z}_i^\top \boldsymbol{z}_j)^2. \tag{15}$$

FastH-Hinge The Hinge loss function is based on Hamming distance:

$$L_{\mathrm{Hinge}}(\boldsymbol{z}_i, \boldsymbol{z}_j) = \begin{cases} [0 - d_{\mathrm{H}}(\boldsymbol{z}_i, \boldsymbol{z}_j)]^2 & \text{if } y_{ij} > 0, \\ [\max(0.5m - d_{\mathrm{H}}(\boldsymbol{z}_i, \boldsymbol{z}_j), 0)]^2 & \text{if } y_{ij} < 0. \end{cases} \tag{16}$$

FastH-BRE The BRE loss function is based on Hamming distance:

$$L_{\mathrm{BRE}}(\boldsymbol{z}_i, \boldsymbol{z}_j) = [m\delta(y_{ij} < 0) - d_{\mathrm{H}}(\boldsymbol{z}_i, \boldsymbol{z}_j)]^2. \tag{17}$$

FastH-ExpH Here ExpH is an exponential loss function using the Hamming distance:

$$L_{\mathrm{ExpH}}(\boldsymbol{z}_i, \boldsymbol{z}_j) = \exp\left[y_{ij} d_{\mathrm{H}}(\boldsymbol{z}_i, \boldsymbol{z}_j)/m + \delta(y_{ij} < 0)\right]. \tag{18}$$

These loss functions are evaluated in the experiment section later. It is worth noting that the Hinge loss (16) encourages the Hamming distance of dissimilar pairs to be *at least* more than *half* of the bit length. This is plausible because the Hamming distance of dissimilar pairs is only required to be large enough, but not necessarily be the maximum value. In contrast, other regression-like loss functions (e.g., KSH, BRE) push the distance of dissimilar pairs to the maximum value (the bit length), which may introduce unnecessary penalties. As empirically verified in our experiments, the Hinge loss usually performs better.

One motivation of KSH in [8] for using the Hamming affinity based loss function rather than the Hamming distance is that they can apply efficient optimization algorithms. However, here we show that both Hamming affinity and Hamming distance based loss functions can be easily solved in our general two-step framework using identical optimization techniques.

To apply the result of Proposition 1 in (13a), we take the KSH loss function as an example. Recall that $l_r(z_{r,i}, z_{r,j}; y_{ij})$ in (10) is the loss function output for the r-th bit and data pair (i, j). Using the KSH loss in (15), we have:

$$l_r(z_{r,i}, z_{r,j}; y_{ij}) = (ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)} - z_{r,i}z_{r,j})^2 \quad (19)$$

Recall that $l^{(11)}$ is the loss output on identical input pairs, and $l^{(-11)}$ is the loss output on distinct input pairs. With the above equation, we can write $l_{r,i,j}^{(11)}$ and $l_{r,i,j}^{(-11)}$ in (13c) for the KSH loss as:

$$\begin{aligned} l_{r,i,j}^{(11)} &= l_r(1, 1; y_{ij}) \\ &= (ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)} - 1)^2; \quad (20a) \\ l_{r,i,j}^{(-11)} &= l_r(-1, 1; y_{ij}) \\ &= (ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)} + 1)^2. \quad (20b) \end{aligned}$$

Finally, the matrix element $a_{i,j}$ in the BQP problem (13a) is written as:

$$\begin{aligned} a_{i,j}^{\text{KSH}} =& \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}) \quad (21a) \\ =& \delta(y_{ij} \neq 0)[\,(ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)} - 1)^2 \\ & - (ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)} + 1)^2\,] \quad (21b) \\ =& -4\delta(y_{ij} \neq 0)(ry_{ij} - \boldsymbol{z}_i^{(r-1)\top} \boldsymbol{z}_j^{(r-1)}). \quad (21c) \end{aligned}$$

By substituting into (13a) and removing constant multipliers, we obtain the binary code inference problem for the r-th bit using the KSH loss function:

$$\min_{\boldsymbol{z}_r \in \{-1,1\}^n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} z_{r,i} z_{r,j}, \quad (22a)$$

$$\text{where, } a_{i,j} = -\delta(y_{ij} \neq 0)(ry_{ij} - \sum_{p=1}^{r-1} z_{p,i}^* z_{p,j}^*). \quad (22b)$$

Here z^* denotes the binary code in previous bits.

3.1 Spectral method for binary inference

To solve the BQP problem in (14a) for obtaining binary codes, we first describe a simple spectral relaxation based method, then present an efficient graph cut based method for large-scale inference. Spectral relaxation drops the binary constraints. The optimization becomes:

$$\begin{aligned} &\min_{\boldsymbol{z}_{(r)}} \boldsymbol{z}_{(r)}^\top \mathbf{A} \boldsymbol{z}_{(r)}, \\ &\text{s.t.} \quad \|\boldsymbol{z}_{(r)}\|_2^2 = n. \end{aligned} \quad (23)$$

The solution (denoted $\boldsymbol{z}_{(r)}^0$) of the above optimization is simply the eigenvector that corresponds to the minimum eigenvalue of the matrix $\mathbf{A}$. To achieve a better solution, we can solve the following relaxed problem of (14a):

$$\begin{aligned} &\min_{\boldsymbol{z}_{(r)}} \boldsymbol{z}_{(r)}^\top \mathbf{A} \boldsymbol{z}_{(r)}, \\ &\text{s.t.} \quad \boldsymbol{z}_{(r)} = [-1, 1]^n. \end{aligned} \quad (24)$$

We use the solution $\boldsymbol{z}_{(r)}^0$ of spectral relaxation in (23) as an initialization and solve the above problem using the efficient LBFGS-B solver [33]. The solution is then thresholded at 0 to output the final binary codes.

3.2 Block GraphCut for binary code inference

We have shown that the simple spectral method can be used to solve the binary code inference problem in (14a). However solving eigenvalue problems does not scale up to large training sets, and the loose relaxation leads to inferior results. Here we propose sub-modular formulations and an efficient graph cut based block search method for solving large-scale inference problems. This block search method is much more efficient than the spectral method and able to achieve better solutions.

Specifically, we first group data points into a number of blocks, then iteratively optimize for these blocks until converge. At each iteration, we randomly pick one block, then optimize for (update) the corresponding binary variables of this block, conditioning on the remaining variables. In other words, when optimizing for one block, only those binary variables that correspond to the data points of the target block will be updated; and for the variables that are not involved in the target block, their values remain unchanged. Clearly each block update would strictly decrease the objective.

Formally, let $\mathcal{B}$ denote a block of data points. We want to optimize for the corresponding binary variables of the block $\mathcal{B}$. We denote by $\hat{z}_r$ a binary code in the r-bit that is not involved in the target block. First we rewrite the objective in (13a) to separate the variables of the target block from other variables. The objective in (13a) can be rewritten as:

$$\begin{aligned} &\sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} z_{r,i} z_{r,j} \quad (25a) \\ =& \sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{B}} a_{i,j} z_{r,i} z_{r,j} + \sum_{i \in \mathcal{B}} \sum_{j \notin \mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j} \\ & + \sum_{i \notin \mathcal{B}} \sum_{j \in \mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j} + \sum_{i \notin \mathcal{B}} \sum_{j \notin \mathcal{B}} a_{i,j} \hat{z}_{r,i} \hat{z}_{r,j} \quad (25b) \\ =& \sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{B}} a_{i,j} z_{r,i} z_{r,j} + 2 \sum_{i \in \mathcal{B}} \sum_{j \notin \mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j} \\ & + \sum_{i \notin \mathcal{B}} \sum_{j \notin \mathcal{B}} a_{i,j} \hat{z}_{r,i} \hat{z}_{r,j}. \quad (25c) \end{aligned}$$

When optimizing for one block, those variables which are not involved in the target block are treated as constants; hence $\hat{z}_r$ is treated as a constant. By removing the constant part, the optimization for one block is:

$$\min_{\boldsymbol{z}_{r,\mathcal{B}} \in \{-1,1\}^{|\mathcal{B}|}} \sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{B}} a_{i,j} z_{r,i} z_{r,j} + 2 \sum_{i \in \mathcal{B}} \sum_{j \notin \mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j}. \tag{26}$$

We aim to optimize $\boldsymbol{z}_{r,\mathcal{B}}$ which is a vector of variables that are involved in the target block $\mathcal{B}$. Substituting the constant $a_{i,j}$ by its definition in (13b), the above optimization is written as:

$$\min_{\boldsymbol{z}_{r,\mathcal{B}} \in \{-1,1\}^{|\mathcal{B}|}} \sum_{i \in \mathcal{B}} u_i z_{r,i} + \sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{B}} v_{ij} z_{r,i} z_{r,j}, \tag{27a}$$

$$\text{where, } v_{ij} = \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}) \tag{27b}$$

$$u_i = 2 \sum_{j \notin \mathcal{B}} \hat{z}_{r,j} \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}). \tag{27c}$$

Here u_i, v_{ij} are constants. The key to constructing a block is to ensure (27a) for such a block is sub-modular, thus we are able to apply the efficient graph cut method. We refer to this as Block GraphCut (Block-GC), shown in Algorithm 3. Specifically in our hashing problem, by leveraging similarity information, we can easily construct blocks to meet the sub-modularity requirement. Here we assume that the loss function satisfies the following conditions:

$$\forall y_{ij} \geq 0 \text{ and } \forall r:$$
$$a_{i,j} = \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}) \leq 0, \tag{28}$$

which intuitively means that, for two similar data points, the loss of assigning identical binary values for one bit is smaller than assigning distinct binary values. As loss functions always encourage two similar data points to have similar binary codes, this condition can be naturally satisfied. All of the loss functions (e.g., KSH, BRE, Hinge) that we described before meet this requirement. As an example, the definition of $a_{i,j}^{\mathrm{KSH}}$ in (21) for the KSH loss satisfies the above conditions. The following proposition shows how to construct such a block:

Proposition 2: $\forall i, j \in \mathcal{B}$, if $y_{ij} \geq 0$, the optimization in (27a) is a sub-modular problem. In other words, for any data point in the block, if it is *n*ot dissimilar with any other data points in the block, then (27a) is sub-modular.

Proof: If $y_{ij} \geq 0$, according to the conditions in (28), we have:

$$v_{ij} = \delta(y_{ij} \neq 0)(l_{r,i,j}^{(11)} - l_{r,i,j}^{(-11)}) \leq 0. \tag{29}$$

With the following definition:

$$\theta_{i,j}(z_{r,i}, z_{r,j}) = v_{ij} z_{r,i} z_{r,j}, \tag{30}$$

the following holds:

$$\theta_{i,j}(-1, 1) = \theta_{i,j}(1, -1) = -v_{ij} \geq 0; \tag{31}$$

$$\theta_{i,j}(1, 1) = \theta_{i,j}(-1, -1) = v_{ij} \leq 0. \tag{32}$$

Hence we have the following relations: $\forall i, j \in \mathcal{B}$:

$$\theta_{i,j}(1, 1) + \theta_{i,j}(-1, -1) \leq 0 \leq \theta_{i,j}(1, -1) + \theta_{i,j}(-1, 1), \tag{33}$$

which prove the sub-modularity of (27a) [34]. □

Blocks can be constructed in many ways as long as they satisfy the condition in Proposition 2. A simple greedy method is shown in Algorithm 2. Note that one block can overlap with another and the union of all blocks needs to cover all n variables. If one block only consist of one variable, Block-GC becomes the ICM method ([35], [36]) which optimizes for one variable at a time.

4 STEP 2: HASH FUNCTION LEARNING

The second step is to solve a binary classification problem for learning one hash function. The binary codes obtained in the first step are used as the classification labels. Any binary classifiers (e.g., decision trees, neural networks) and any advanced large-scale training techniques can be directly applied to hash function learning at this step. For the r-th bit, the classification problem is:

$$\min_{h_r(\cdot)} \sum_{i=1}^{n} \delta(z_{r,i} = h_r(\boldsymbol{x}_i)). \tag{34}$$

Usually the zero-one loss in the above problem is replaced by some convex surrogate loss. For example, when training a perceptron hash function:

$$h(\boldsymbol{x}) = \mathrm{sign}\,(\boldsymbol{w}^\top \boldsymbol{x} + b), \tag{35}$$

we can train a linear SVM classifier by solving:

$$\min_{\boldsymbol{w}, b} \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \max[1 - z_{r,i}(\boldsymbol{w}^\top \boldsymbol{x} + b), 0]. \tag{36}$$

Any binary classifier can be applied here. We could also train an kernel SVM to learn a kernel hash function:

$$h(\boldsymbol{x}) = \mathrm{sign}\left[\sum_{q=1}^{Q} w_q \kappa(\boldsymbol{x}'_q, \boldsymbol{x}) + b\right], \tag{37}$$

in which $\mathcal{X}' = \{\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_Q\}$ are Q support vectors. Sophisticated kernel learning methods can be applied here, for example, LIBSVM or the stochastic kernel SVM training method with a support vector budget in [37].

After learning the hash function for one bit, the binary code is updated by applying the learned hash function. Hence, the learned hash function is able to influence the learning of the next bit.

4.1 Boosted trees as hash functions

Decision trees could be a good choice for hash functions with nonlinear mapping. Compared to kernel method, decision trees only involve simple comparison operations for evaluation; thus they are much more efficient for testing, especially on high-dimensional data. We define one hash function as a linear combination of trees:

$$h(\boldsymbol{x}) = \mathrm{sign}\left[\sum_{q=1}^{Q} w_q T_q(\boldsymbol{x})\right]. \tag{38}$$

TABLE 1 – Comparison of KSH and our FastHash. KSH results are presented with different numbers of support vectors. Both of our FastHash and FastHash-Full significantly outperform KSH in terms of training time, binary encoding time (test time) and retrieval precision.

Method	#Train	#Support Vector	Train time	Test time	Precision
		CIFAR10 (features:11200)			
KSH	5000	300	1082	22	0.480
KSH	5000	1000	3481	57	0.553
KSH	5000	3000	52747	145	0.590
FastH	5000	N/A	331	21	**0.634**
FastH-Full	50000	N/A	1794	21	**0.763**
		IAPRTC12 (features:11200)			
KSH	5000	300	1129	7	0.199
KSH	5000	1000	3447	21	0.235
KSH	5000	3000	51927	51	0.273
FastH	5000	N/A	331	9	**0.285**
FastH-Full	17665	N/A	620	9	**0.371**
		ESPGAME (features:11200)			
KSH	5000	300	1120	8	0.124
KSH	5000	1000	3358	22	0.139
KSH	5000	3000	52115	46	0.163
FastH	5000	N/A	309	9	**0.188**
FastH-Full	18689	N/A	663	9	**0.261**
		MIRFLICKR (features:11200)			
KSH	5000	300	1036	5	0.387
KSH	5000	1000	3337	13	0.407
KSH	5000	3000	52031	42	0.434
FastH	5000	N/A	278	7	**0.555**
FastH-Full	12500	N/A	509	7	**0.595**

Here Q is the number of decision trees; $T(\cdot) \in \{-1, 1\}$ denotes a tree function with binary output. We train a boosting classifier to learn the weighting coefficients and trees for one hash function. The classification problem for the r-th hash function is written as:

$$\min_{\boldsymbol{w} \geq 0} \sum_{i=1}^{n} \exp \left[-z_{r,i} \sum_{q=1}^{Q} w_q T_q(\boldsymbol{x}_i) \right]. \tag{39}$$

We apply Adaboost to solve the problem. At each boosting iteration, a decision tree as well as its weighting coefficient is learned. Every node of a binary decision tree is a decision stump. Training a stump is to find a feature dimension and threshold that minimizes the weighted classification error. From this point of view, we are performing feature selection and hash function learning at the same time. We can easily make use of efficient decision tree learning techniques available in the literature. Here we summarize some techniques that are included in our implementation:

1) We use the efficient stump implementation proposed in the recent work of [38], which is around 10 times faster than conventional implementation.

2) Feature quantization can significantly speed up tree training without noticeable performance loss in practice, and also largely reduce the memory consuming. As in [38], we linearly quantize feature values into 256 bins.

3) We apply the weight-trimming technique described in [39], [38]. At each boosting iteration, the smallest 10% weightings are trimmed (set to 0).

4) We apply the LazyBoost technique to speed up the tree learning process. For one node splitting in tree training, only a random subset of feature dimensions are evaluated for splitting.

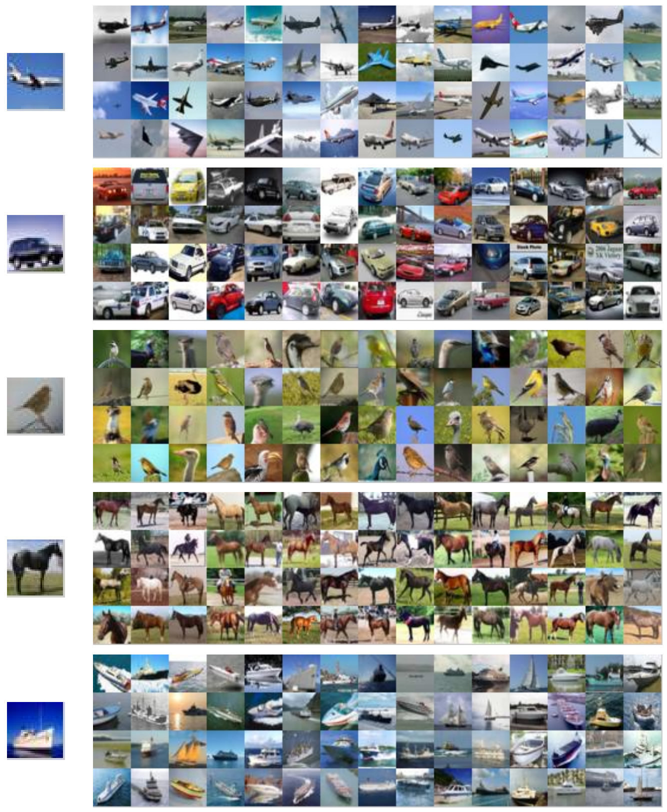


Fig. 1 – Some retrieval examples of our method FastHash on CIFAR10. The first column shows query images, and the rest are retrieved images in the database.

5 Experiments

To evaluate the proposed method, here we present the results of comprehensive experiments on several large image datasets. The evaluation measures include training time, binary encoding time and retrieval accuracy. We compare to a number of recent supervised and unsupervised hashing methods. To explore our method with different settings, we perform comparisons of using different binary code inference algorithms, various kinds of loss functions and hash functions.

The similarity preserving performance is evaluated in small binary codes based image retrieval [40], [2]. Given a query image, the retrieved images in the database are returned by hamming distance ranking based on their binary codes. The retrieval quality is measured in 3 different aspects: the precision of the top-K (K=100) retrieved examples (denoted as Precision), mean average precision (MAP) and the area under the Precision-Recall curve (Prec-Recall). The training time and testing time (binary encoding time) are recorded in seconds.

Results are reported on 6 large image datasets which cover a wide variety of images. The dataset CIFAR10 [1] contains $60,000$ images in small resolution. The multi-label datasets IAPRTC12 and ESPGAME [41] contain around $20,000$ images, and MIRFLICKR [42] is a collection of 25000 images. SUN397 [43] contains more than

1. http://www.cs.toronto.edu/~kriz/cifar.html

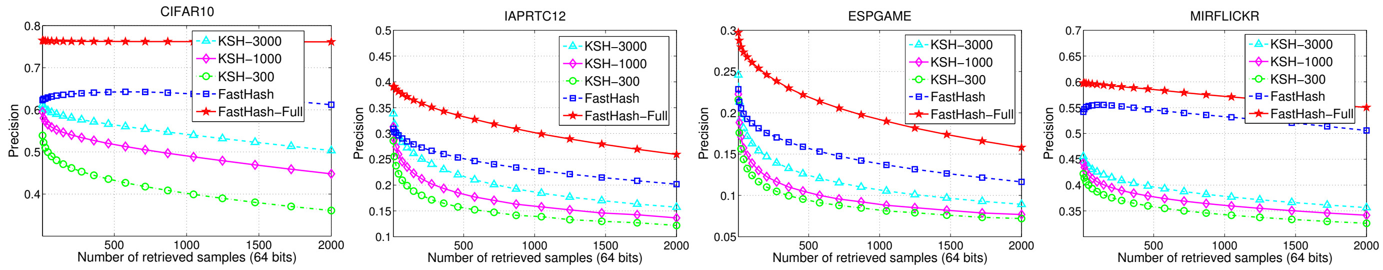


Fig. 2 – Comparison of KSH and our FastHash on all datasets. The number after "KSH" is the number of support vectors. Both of our FastHash and FastHash-Full significantly outperform KSH.

$100,000$ scene images form 397 categories. The large dataset ILSVRC2012 contains roughly 1.2 million images of 1000 categories from ImageNet [44].

For the multi-class datasets: CIFAR10, SUN397 and ILSVRC2012, the ground truth pairwise similarity is defined as multi-class label agreement. For multi-label datasets: IAPRTC12, ESPGAME and MIRFLICKR, of which the keyword (tags) annotation is provided in [41], two images are considered as semantically similar if they are annotated with at least 2 identical keywords (or tags). In the training stage of supervised methods, a maximum number of 100 similar and dissimilar neighbors are defined for each example; hence the pairwise similarity label matrix is sparse.

Following the conventional protocol in [8], [11], [2] for hashing method evaluation, a large portion of the dataset is allocated as an image database for training and retrieval, and the rest is put aside as test queries. Specifically, for CIFAR10, IAPRTC12, ESPGAME and MIRFLICKER, the training data in the the provided split are used as image database and the test data are used as test queries. The splits for SUN397 and ILSVRC2012 are described in their corresponding sections.

We extract codebook-based features following the conventional pipeline from [12], [45]: we employ K-SVD for codebook (dictionary) learning with a codebook size of 800, soft-thresholding for patch encoding and spatial pooling of 3 levels, which results 11200-dimensional features. For further evaluation, we increase the codebook size to 1600 to generate 22400-dimensional features. We also extract the low-dimensional GIST [46] features (512 or 320 dimensions) for evaluation. For the dataset ILSVRC2012, we extract the convolution neural network features with 4096 dimensions ([13], [47]), which is described in detail in the corresponding section.

If not specified, we use the following in our method: the KSH loss function (described in (15)), the proposed Block GraphCut algorithm in Step-1, and the decision tree hash function in Step-2. The tree depth is set to 4, and the number of boosting iterations is 200. Different settings of hash functions or loss functions will be evaluated in the later sections. For comparison methods, we follow the original papers for parameter setting. If not specified, 64-bit binary codes are generated for evaluation.

5.1 Comparison with KSH

KSH [8] has been shown to outperform many state-of-the-art comparators. Here we evaluate our method using the KSH loss and compare against the original KSH method on high-dimensional codebook features. KSH employs a simple kernel technique by predefining a set of support vectors then learning linear weights for each hash function. For our method, we use boosted decision trees as hash functions. KSH is trained on a sampled set of 5000 examples, and the number of support vectors for KSH is varied from 300 to 3000. The results are summarized in Table 1, which shows that increasing the number of support vectors consistently improves the retrieval precision of KSH. However, even on this small training set, including more support vectors will dramatically increase the training time and binary encoding time of KSH. We have run our FastHash both on the same sampled training set and the whole training set (labeled as FastHash-Full) in order to show that our method can be efficiently trained on the whole dataset. Our FastHash and FastHash-Full outperform KSH by a large margin in terms of both training speed and retrieval precision. It also shows that the decision tree hash functions in FastHash are much more efficient for testing (binary encoding) than the kernel function in KSH. FastHash is orders of magnitude faster than KSH in training, and thus much better suited to large training sets and high-dimensional data. We also show the precision curves of top-K retrieved examples in Figure 2. The number after "KSH" is the number of support vectors. Besides high-dimensional features, we also compare with KSH on the low-dimensional GIST feature, and FastHash also significantly performs better; see Table 2 for details. Some retrieval examples of our method are shown in Figure 1.

5.2 Evaluation on different features

We evaluate our method both on the low-dimensional (320 or 512) GIST features and the high-dimensional (11200) codebook features. Several state-of-the-art supervised methods are included in this comparison: KSH [8], Supervised Self-Taught Hashing (STHs) [26], and Semi-supervised Hashing (SPLH) [2]. Comparison methods are trained on a sampled training set (5000 examples). Results are presented in Table 2. The codebook features

TABLE 2 – Results using two types of features: low-dimensional GIST features and the high-dimensional codebook features. Our FastHash and FastHash-Full significantly outperform the comparators on both feature types. In terms of training time, our FastHash is also much faster than others on the high-dimensional codebook features.

		GIST feature (320 / 512 dimensions)					Codebook feature (11200 dimensions)				
Method	#Train	Train time	Test time	Precision	MAP	Prec-Recall	Train time (s)	Test time (s)	Precision	MAP	Prec-Recall
					CIFAR10						
KSH	5000	52173	8	0.453	0.350	0.164	52747	145	0.590	0.464	0.261
BREs	5000	481	1	0.262	0.198	0.082	18343	8	0.292	0.216	0.089
SPLH	5000	102	1	0.368	0.291	0.138	9858	4	0.496	0.396	0.219
STHs	5000	380	1	0.197	0.151	0.051	6878	4	0.246	0.175	0.058
FastH	5000	304	21	**0.517**	**0.462**	**0.243**	331	21	**0.634**	**0.575**	**0.358**
FastH-Full	50000	1681	21	**0.649**	**0.653**	**0.450**	1794	21	**0.763**	**0.775**	**0.605**
					IAPRTC12						
KSH	5000	51864	5	0.182	0.126	0.083	51927	51	0.273	0.169	0.123
BREs	5000	6052	1	0.138	0.109	0.074	6779	3	0.163	0.124	0.097
SPLH	5000	154	1	0.160	0.124	0.084	10261	2	0.220	0.157	0.119
STHs	5000	628	1	0.099	0.092	0.062	10108	2	0.160	0.114	0.076
FastH	5000	286	9	**0.232**	**0.168**	**0.117**	331	9	**0.285**	**0.202**	**0.146**
FastH-Full	17665	590	9	**0.316**	**0.240**	**0.178**	620	9	**0.371**	**0.276**	**0.210**
					ESPGAME						
KSH	5000	52061	5	0.118	0.077	0.054	52115	46	0.163	0.100	0.072
BREs	5000	714	1	0.095	0.070	0.050	16628	3	0.111	0.076	0.059
SPLH	5000	185	1	0.116	0.083	0.062	11740	2	0.148	0.104	0.074
STHs	5000	616	1	0.061	0.047	0.033	11045	2	0.087	0.064	0.042
FastH	5000	289	9	**0.157**	**0.106**	**0.070**	309	9	**0.188**	**0.125**	**0.081**
FastH-Full	18689	448	9	**0.228**	**0.169**	**0.109**	663	9	**0.261**	**0.189**	**0.126**
					MIRFLICKR						
KSH	5000	51983	3	0.379	0.321	0.234	52031	42	0.434	0.350	0.254
BREs	5000	1161	1	0.347	0.310	0.224	13671	2	0.399	0.345	0.250
SPLH	5000	166	1	0.379	0.337	0.241	9824	2	0.444	0.391	0.277
STHs	5000	613	1	0.268	0.261	0.172	10254	2	0.281	0.272	0.174
FastH	5000	307	7	**0.477**	**0.429**	**0.299**	338	7	**0.555**	**0.487**	**0.344**
FastH-Full	12500	451	7	**0.525**	**0.507**	**0.345**	509	7	**0.595**	**0.558**	**0.420**

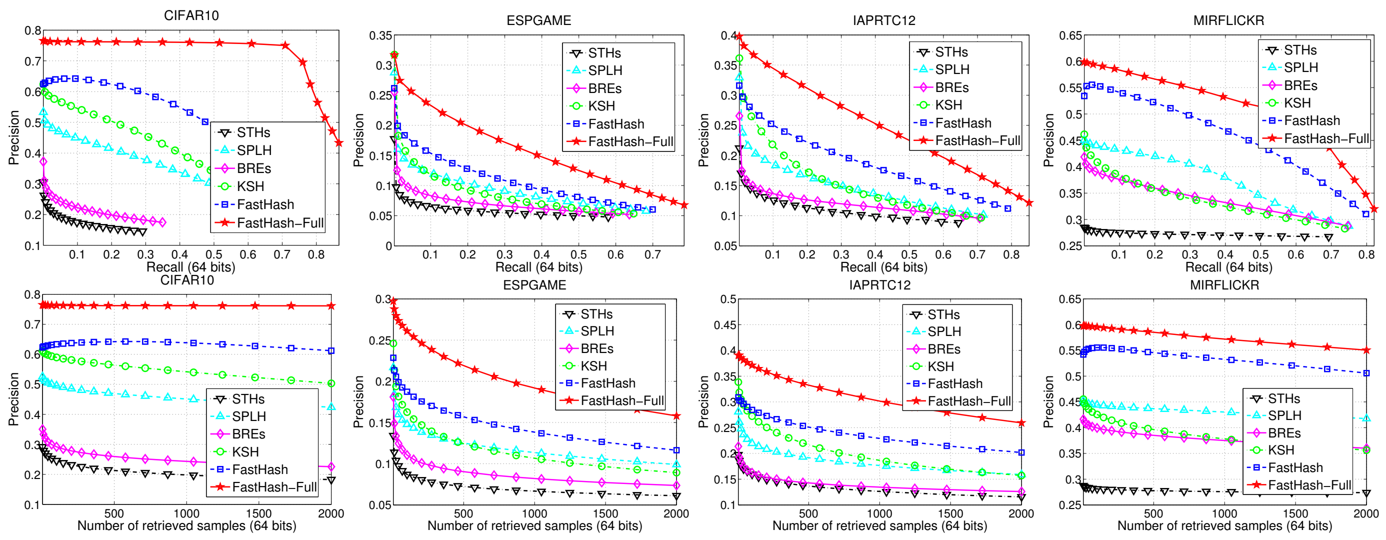


Fig. 3 – Results on high-dimensional codebook features. Our FastHash significantly outperform others.

consistently show better results than the GIST features. The results for codebook features are also plotted in Figure 3. It shows that the competing methods can be efficiently trained on the GIST features. However, when applied to high dimensional features, even on a small training set (5000), their training time dramatically increase. It is very difficult to train these methods on the whole training set. The training time of KSH mainly depends on the number of support vectors (3000 is used here). We run our FastHash on the same sampled training set (5000 examples) and the whole training set (labeled as FastHash-Full). Results show that FastHash can be efficiently trained on the whole dataset. FastHash significantly outperform others both in GIST and codebook features. The training of FastHash is also orders of magnitudes faster than others on the high-dimensional codebook features.

5.3 Comparison with dimension reduction

One possible way to reduce the training cost on high-dimensional data is to apply dimension reduction. For the methods: KSH, SPLH and STHs, we thus reduce the original 11200-dimensional codebook features to 500 dimensions by applying PCA. We also compare to CCA+ITQ [6] which combines ITQ with supervised dimensional reduction. Our FastHash still uses the original high-dimensional features. The result is summarized in Table 3. After dimension reduction, most comparison

TABLE 3 – Results of methods with dimension reduction. Our FastHash significantly outperforms others.

Method	# Train	Train time	Test time	Precision	MAP
		CIFAR10			
PCA+KSH	50000	–	–	–	–
PCA+SPLH	50000	25984	18	0.482	0.388
PCA+STHs	50000	7980	18	0.287	0.200
CCA+ITQ	50000	1055	7	0.676	0.642
FastH	50000	1794	21	**0.763**	**0.775**
		IAPRTC12			
PCA+KSH	17665	55031	11	0.082	0.103
PCA+SPLH	17665	1855	7	0.239	0.169
PCA+STHs	17665	2463	7	0.174	0.126
CCA+ITQ	17665	804	3	0.332	0.198
FastH	17665	620	9	**0.371**	**0.276**
		ESPGAME			
PCA+KSH	18689	55714	11	0.141	0.084
PCA+SPLH	18689	2409	7	0.153	0.103
PCA+STHs	18689	2777	7	0.098	0.069
CCA+ITQ	18689	814	3	0.216	0.131
FastH	18689	663	9	**0.261**	**0.189**
		MIRFLICKR			
PCA+KSH	12500	54260	8	0.384	0.313
PCA+SPLH	12500	1054	5	0.445	0.391
PCA+STHs	12500	1768	5	0.347	0.301
CCA+ITQ	12500	699	3	0.519	0.408
FastH	12500	509	7	**0.595**	**0.558**

TABLE 4 – Performance of our FastHash on more features (22400 dimensions) and more bits (1024 bits). The training and binary coding time (test time) of FastHash is only linearly increased with the bit length.

Bits	#Train	Features	Train time	Test time	Precision	MAP
			CIFAR10			
64	50000	11200	1794	21	0.763	0.775
256	50000	22400	5588	71	0.794	0.814
1024	50000	22400	22687	282	0.803	0.826
			IAPRTC12			
64	17665	11200	320	9	0.371	0.276
256	17665	22400	1987	33	0.439	0.314
1024	17665	22400	7432	134	0.483	0.338
			ESPGAME			
64	18689	11200	663	9	0.261	0.189
256	18689	22400	1912	34	0.329	0.233
1024	18689	22400	7689	139	0.373	0.257
			MIRFLICKR			
64	12500	11200	509	7	0.595	0.558
256	12500	22400	1560	28	0.612	0.567
1024	12500	22400	6418	105	0.628	0.576

methods can be trained on the whole training set within 24 hours (except KSH on CIFAR10). However it still much slower than our FastHash. Our FastHash also performs significantly better on retrieval precision. Learning decision tree hash functions in FastHash actually perform feature selection and hash function learning at the same time, which shows much better performance than other hashing methods with dimensional reduction.

5.4 More features and more bits

We increase the codebook size to 1600 for generating higher dimensional features (22400 dimensions) and run up to 1024 bits. Table 4 shows that FastHash can be efficiently trained on high-dimensional features with large bit length. The training and binary coding time (test time) of FastHash increases only linearly with bit length. The retrieval result is improved when the bit length is increased.

TABLE 5 – Comparison of spectral method and the proposed Block GraphCut (Block-GC) for binary code inference. Block-GC achieves lower objective value and takes less inference time, thus performs much better.

Step-1 methods	#train	Block Size	Time (s)	Objective
		SUN397		
Spectral	100417	N/A	5281	0.7524
Block-GC-1	100417	1	**298**	0.6341
Block-GC	100417	253	2239	**0.5608**
		CIFAR10		
Spectral	50000	N/A	1363	0.4912
Block-GC-1	50000	1	**158**	0.5338
Block-GC	50000	5000	788	**0.4158**
		IAPRTC12		
Spectral	17665	N/A	426	0.7237
Block-GC-1	17665	1	**43**	0.7316
Block-GC	17665	316	70	**0.7095**
		ESPGAME		
Spectral	18689	N/A	480	0.7373
Block-GC-1	18689	1	**45**	0.7527
Block-GC	18689	336	72	**0.7231**
		MIRFLICKR		
Spectral	12500	N/A	125	0.5718
Block-GC-1	12500	1	**28**	0.5851
Block-GC	12500	295	40	**0.5449**

TABLE 6 – Comparison of combinations of hash functions and binary inference methods. Decision tree hash functions perform much better than linear SVM (LSVM) hash functions. The proposed Block GraphCut (Block-GC) performs much better than the spectral method.

Step-1 method	Step-2 method	Precision	MAP	Prec-Recall
		CIFAR10		
Block-GC	TREE	**0.763**	**0.775**	**0.605**
Spectral	TREE	0.731	0.695	0.501
Block-GC	LSVM	0.669	0.621	0.435
Spectral	LSVM	0.624	0.512	0.322
		IAPRTC12		
Block-GC	TREE	**0.371**	**0.276**	**0.210**
Spectral	TREE	0.355	0.265	0.201
Block-GC	LSVM	0.327	0.238	0.186
Spectral	LSVM	0.275	0.207	0.160
		ESPGAME		
Block-GC	TREE	**0.261**	**0.189**	**0.126**
Spectral	TREE	0.249	0.183	0.123
Block-GC	LSVM	0.227	0.157	0.109
Spectral	LSVM	0.183	0.133	0.093
		MIRFLICKR		
Block-GC	TREE	**0.595**	**0.558**	**0.420**
Spectral	TREE	0.584	0.551	0.413
Block-GC	LSVM	0.536	0.498	0.344
Spectral	LSVM	0.489	0.466	0.319

5.5 Binary code inference evaluation

Here we evaluate different algorithms for solving the binary code inference problem which is involved in Step-1 of our learning process. We compare the proposed Block GraphCut (Block-GC) with the simple spectral method. The number of iterations of Block-GC is set to 2, which is the same as that in other experiments. Results are summarized in Table 5. We construct blocks using Algorithm 2. The averaged block size is reported in the table. We also evaluate a special case where the block size is set to 1 for Block-CG (labeled as Block-CG-1), in which case Block-GC is reduced to the ICM ([35], [36]) method. It shows that when the training set gets larger, the spectral method becomes slow. The objective value shown in the table is divided by the number of defined pairwise relations. Results show that the proposed Block-

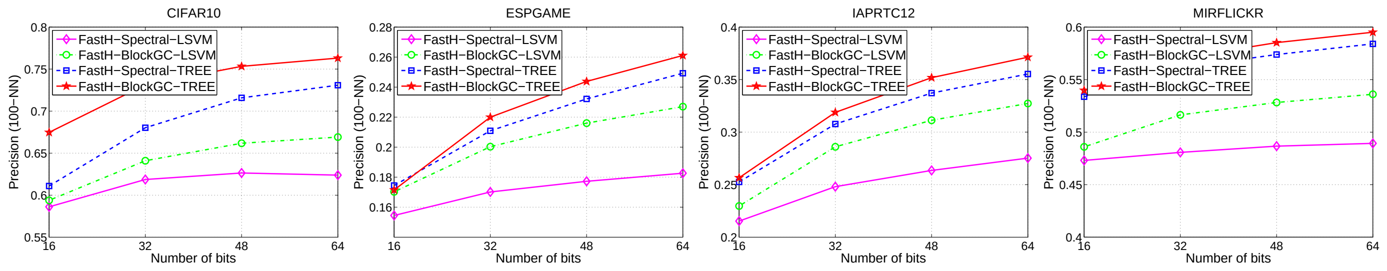


Fig. 4 – Comparison of combinations of hash functions and binary inference methods. Decision tree hash functions perform much better than linear SVM. The proposed Block-GC performs much better than the spectral method.

GC achieves much lower objective values and takes less inference time, and hence outperforms the spectral method. The inference time for Block-CG increases only linearly with training set size.

Our method is able to incorporate different kinds of hash functions in Step-2. Here we provide results comparing different combinations of hash functions (Step-2) and binary code inference methods (Step-1). We evaluate linear SVM and decision tree hash functions with the spectral method and the proposed Block-GC. Codebook features are used here. Results are summarized in Table 6. We also plot the retrieval performance in Figure 4. As expected, decision tree hash functions perform much better than linear SVM hash functions, and the proposed Block-GC performs much better than the spectral method, which indicates that Block-GC is able to generate high quality binary codes.

5.6 Using different loss functions

Our method is able to incorporate different kinds of loss functions and hash functions. Here we compare the performance of 4 kinds of loss function: KSH (15), Hinge (16), BRE (17) and ExpH (18), combined with linear SVM (35) and boosted decision tree (38) hash functions. Results are summarized in Table 7. It shows that the Hinge loss usually achieves the best performance, and the remaining loss functions have similar performances. The Hinge loss function in (16) encourages the hamming distance of dissimilar pairs to be at least half of the bit length, instead of unnecessarily pushing it to the maximum value. It also shows that decision tree hash functions perform much better than linear SVM. We plot the performance of decision tree hash functions combined with different kinds of loss functions in Figure 5.

5.7 Comparison with unsupervised methods

We compare to some popular unsupervised hashing methods: LSH [7], ITQ [6], Anchor Graph Hashing (AGH) [16], Spherical Hashing (SPHER) [17], Multi-dimension Spectral Hashing (MDSH) [10] [10]. The retrieval performance is shown in Figure 6. Unsupervised methods perform poorly at preserving label based similarity. Our FastHash significantly outperforms others.

TABLE 7 – Comparison of combinations of different loss functions and hash functions. Using the Hinge loss achieves the best result. Decision tree hash functions perform much better than linear SVM hash functions.

Loss	Step-2 method	Precision	MAP	Prec-Recall
		CIFAR10		
FastH-KSH	TREE	0.763	0.775	0.605
FastH-BRE	TREE	0.761	0.772	0.602
FastH-HINGE	TREE	**0.773**	**0.780**	**0.613**
FastH-EXPH	TREE	0.765	0.774	0.604
FastH-KSH	LSVM	0.669	0.621	0.435
FastH-BRE	LSVM	0.667	0.619	0.431
FastH-HINGE	LSVM	0.669	0.604	0.387
FastH-EXPH	LSVM	0.665	0.619	0.430
		IAPRTC12		
FastH-KSH	TREE	0.371	0.276	0.210
FastH-BRE	TREE	0.375	0.279	0.213
FastH-HINGE	TREE	**0.410**	**0.295**	**0.234**
FastH-EXPH	TREE	0.344	0.268	0.199
FastH-KSH	LSVM	0.327	0.238	0.186
FastH-BRE	LSVM	0.328	0.237	0.187
FastH-HINGE	LSVM	0.338	0.247	0.194
FastH-EXPH	LSVM	0.295	0.225	0.170
		ESPGAME		
FastH-KSH	TREE	0.261	0.189	0.126
FastH-BRE	TREE	0.262	0.189	0.125
FastH-HINGE	TREE	**0.286**	**0.200**	**0.148**
FastH-EXPH	TREE	0.253	0.194	0.124
FastH-KSH	LSVM	0.227	0.157	0.109
FastH-BRE	LSVM	0.231	0.160	0.111
FastH-HINGE	LSVM	0.225	0.155	0.109
FastH-EXPH	LSVM	0.216	0.154	0.104
		MIRFLICKR		
FastH-KSH	TREE	0.595	0.558	0.420
FastH-BRE	TREE	0.596	0.559	0.420
FastH-HINGE	TREE	**0.647**	**0.592**	**0.457**
FastH-EXPH	TREE	0.560	0.543	0.404
FastH-KSH	LSVM	0.536	0.498	0.344
FastH-BRE	LSVM	0.531	0.494	0.341
FastH-HINGE	LSVM	0.567	0.522	0.397
FastH-EXPH	LSVM	0.502	0.471	0.323

5.8 Large dataset: SUN397

The SUN397 [43] dataset contains more than $100,000$ scene images. 8000 images are randomly selected as test queries, while the remaining $100,417$ images form the training set. 11200-dimensional codebook features are used here. We compare with a number of supervised and unsupervised methods. The depth for decision trees is set to 6. Results are presented in Table 8 Supervised methods: KSH, BREs, SPLH and STHs are trained on a subset of 10K examples. Even on this sampled training set, the training of these methods are already impractically slow. In contrast, our method can be efficiently trained with a long bit length (1024 bits) on the whole training set (more than $100,000$ training examples). Our FastHash significantly outperforms other methods. The retrieval performance is also plotted in Figure 7. It shows

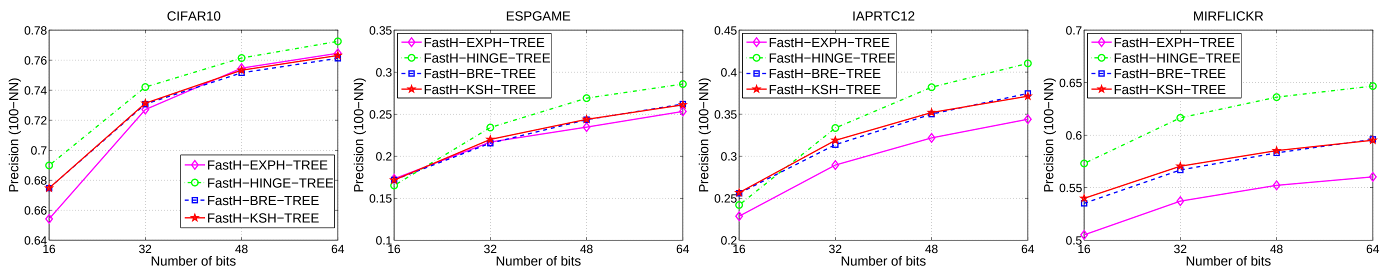


Fig. 5 – Comparison of using different loss functions with decision tree hash functions. Using the Hinge loss (FastH-Hinge) achieves the best result.

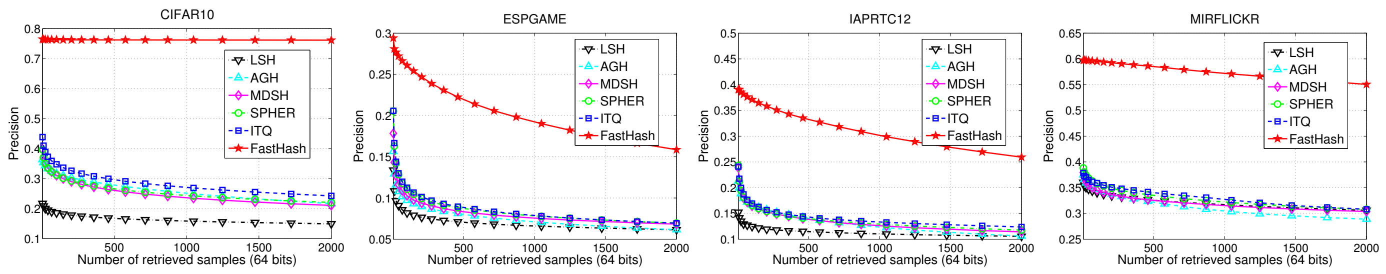


Fig. 6 – Comparison with a few unsupervised hashing methods. Unsupervised methods perform poorly for preserving label based similarity. Our FastHash performs significantly better.

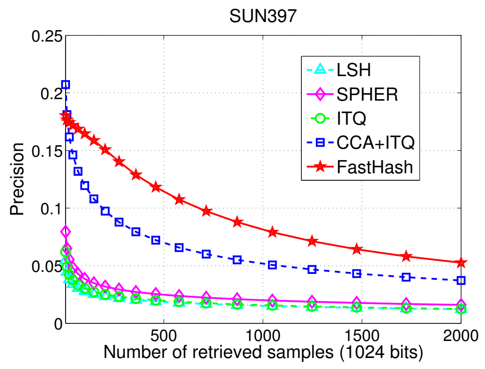


Fig. 7 – The top-2000 precision curve on large dataset SUN397 (1024 bits). Our FastHash performs the best.

TABLE 8 – Results on SUN397 dataset. Our FastHash can be efficiently trained on this large training set. FastHash significantly outperforms other methods.

Method	#Train	Bits	Train time	Test time	Precision	MAP
			SUN397			
KSH	10000	64	57045	463	0.034	0.023
BREs	10000	64	105240	23	0.019	0.013
SPLH	10000	64	27552	14	0.022	0.015
STHs	10000	64	22914	14	0.010	0.008
ITQ	100417	1024	1686	127	0.030	0.021
SPHER	100417	1024	35954	121	0.039	0.024
LSH	–	1024	–	99	0.028	0.019
CCA+ITQ	100417	512	7484	66	0.113	0.076
CCA+ITQ	100417	1024	15580	127	0.120	0.081
FastH	100417	512	29624	302	0.149	0.142
FastH	100417	1024	62076	536	**0.165**	**0.163**

the results of those comparison methods that are able to be trained to 1024 bits on the whole training set. In terms of memory usage, many comparison methods require a large amount of memory for large matrix multiplication. In contrast, the decision tree learning in our method only involves simple comparison operations on quantized feature data (256 bins), thus FastHash consumes less than 7GB for training.

5.9 Large dataset: ImageNet

The large dataset ILSVRC2012 contains more than 1.2 million images from ImageNet [44]. We use the provided training set as the database (around 1.2 million) and the validation set as test queries (50K images). Convolution neural networks (CNNs) have shown the best classification performance on this dataset [13]. As described in [47], the neuron activation values of internal layers of CNNs can be used as features. By using the Caffe toolbox [48] which implements the CNN architecture in [13], we extract 4096-dimensional features from the the seventh layer of the CNN. We compare with a number of supervised and unsupervised methods. The depth for the decision trees is set to 16. The smallest 2% of data weightings are trimmed for decision tree learning. Most comparing supervised methods become intractable on the full training set (1.2 million examples). In contrast, our method is still able to be efficiently trained on the whole training set. For comparison, we also construct a smaller dataset (denoted as ImageNet-50) by sampling 50 classes from ILSVRC2012. It contains $25,000$ training images (500 images for each class) and 2500 testing images. Results of ImageNet-50 and the full ILSVRC2012 are presented in Table 9. Our FastHash performs significantly better than others. The retrieval performance of 128 bits on the full ILSVRC2012 is plotted in Figure 8.

5.10 Image classification

Since binary codes have very small storage cost or network transfer cost, image features can be compressed to binary codes by apply hashing methods. Here we

TABLE 9 – Results on two ImageNet datasets using CNN features. ImageNet-50 is a small subset of ILSVRC2012. Our FastHash significantly outperforms others.

Method	#Train	Bits	Precision	MAP	Prec-Recall
ImageNet-50					
KSH	25000	64	0.572	0.460	0.328
BREs	25000	64	0.377	0.246	0.189
SPLH	25000	64	0.411	0.303	0.217
STHs	25000	64	0.625	0.580	0.412
ITQ+CCA	25000	64	0.690	0.668	0.517
ITQ	25000	64	0.492	0.358	0.266
SPHER	25000	64	0.345	0.210	0.155
LSH	–	64	0.064	0.046	0.023
FastHash	25000	64	**0.697**	**0.718**	**0.532**
ILSVRC2012					
CCA+ITQ	1.2M	64	0.195	0.133	0.049
CCA+ITQ	1.2M	128	0.289	0.199	0.090
CCA+ITQ	1.2M	1024	0.428	0.305	0.160
ITQ	1.2M	64	0.227	0.132	0.053
ITQ	1.2M	128	0.294	0.175	0.080
ITQ	1.2M	1024	0.368	0.227	0.108
LSH	–	1024	0.126	0.065	0.023
FastH	1.2M	64	0.383	0.301	0.107
FastH	1.2M	128	**0.458**	**0.390**	**0.171**

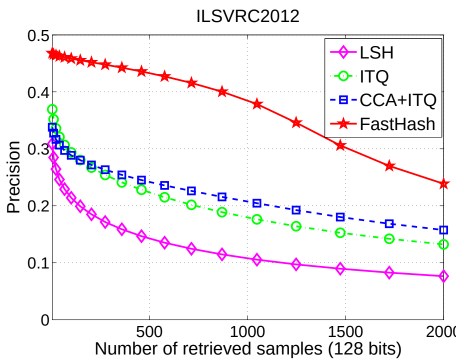


Fig. 8 – The top-2000 precision curve on large dataset ILSVRC2012 (128 bits). Our FastHash outperforms others.

evaluate the image classification performance of using binary codes as features on the dataset ILSVRC2012. Hashing methods are trained on the CNN features. We apply two types of classification methods: the K nearest neighbor (KNN) classifier and the one-vs-all linear SVM classifier. KNN classification is performed by majority voting of top-K retrieved neighbors with smallest hamming distances. Results are shown in Table 10. Our method outperforms all comparison hashing methods.

The CNN features used here are extracted on the center crops of images using Caffe [48]. We also report the results of CNN methods which have the state-of-the-art results of this dataset. As shown in Table 10, the performance gap is around 8% between the error rate of our hashing method and that of Caffe with similar settings (only using center crops). However, 128-bit binary codes in our methods take up around 1000 times less storage than the CNN features with 4096-dimensional float values. It shows that our method is able to perform effective binary compression without large performance loss.

TABLE 10 – Image classification results on dataset ILSVRC2012. Binary codes are generated as features for training classifiers. Our FastHash outperforms other hashing methods for binary compression of features.

Hashing method	bits	KNN-50 test error	1-vs-all SVM test error
ILSVRC2012			
LSH	128	0.594	0.939
ITQ	128	0.557	0.919
CCA+ITQ	64	0.716	0.691
CCA+ITQ	128	0.614	0.583
FastHash	64	0.572	0.567
FastHash	128	**0.516**	**0.512**
Classification method		Test error	
Caffe [48] (center crop)		0.433	
Caffe [48]		0.413	
CNNs [13] (one model)		0.407	

6 Conclusion

We have shown that various kinds of loss functions and hash functions can be placed in a unified learning framework for supervised hashing. By using the proposed binary inference algorithm Block GraphCut and learning decision tree hash functions, our method can be efficiently trained on large-scale and high-dimensional data and achieves high testing precision, which indicates its practical significance on many applications like large-scale image retrieval.

Acknowledgements

This research was in part supported by the Data to Decisions Cooperative Research Centre. C. Shen's participation was in part support by ARC Future Fellowship.

References

[1] A. Torralba, R. Fergus, and W. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE T. Pattern Analysis Mach. Intelli.*, 2008.
[2] J. Wang, S. Kumar, and S. Chang, "Semi-supervised hashing for large scale search," *IEEE T. Pattern Analysis Mach. Intelli.*, 2012.
[3] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua, "Ldahash: Improved matching with smaller descriptors," *IEEE T. Pattern Analysis Mach. Intelli.*, 2012.
[4] T. Dean, M. A. Ruzon, M. Segal, J. Shlens, S. Vijayanarasimhan, and J. Yagnik, "Fast, accurate detection of 100,000 object classes on a single machine," in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2013.
[5] M. Norouzi and D. Fleet, "Minimal loss hashing for compact binary codes," in *Proc. Int. Conf. Mach. Learn.*, 2011.
[6] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: a procrustean approach to learning binary codes for large-scale image retrieval," *IEEE T. Pattern Analysis Mach. Intelli.*, 2012.
[7] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. Int. Conf. Very Large Data Bases*, 1999.
[8] W. Liu, J. Wang, R. Ji, Y. Jiang, and S. Chang, "Supervised hashing with kernels," in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2012.
[9] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Proc. Adv. Neural Info. Process. Syst.*, 2008.
[10] Y. Weiss, R. Fergus, and A. Torralba, "Multidimensional spectral hashing," in *Proc. Eur. Conf. Comp. Vis.*, 2012.
[11] B. Kulis and T. Darrell, "Learning to hash with binary reconstructive embeddings," in *Proc. Adv. Neural Info. Process. Syst.*, 2009.
[12] A. Coates and A. Ng, "The importance of encoding versus training with sparse coding and vector quantization," in *Proc. Int. Conf. Mach. Learn.*, 2011.
[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks." in *Proc. Adv. Neural Info. Process. Syst.*, 2012.

[14] Y. Boykov, O. Veksler, and R. Zabih, “Fast approximate energy minimization via graph cuts,” *IEEE T. Pattern Analysis Mach. Intelli.*, 2001.

[15] K. He, F. Wen, and J. Sun, “K-means hashing: An affinity-preserving quantization method for learning binary compact codes,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2013.

[16] W. Liu, J. Wang, S. Kumar, and S. F. Chang, “Hashing with graphs,” in *Proc. Int. Conf. Mach. Learn.*, 2011.

[17] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon, “Spherical hashing,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2012.

[18] F. Shen, C. Shen, Q. Shi, A. van den Hengel, and Z. Tang, “Inductive hashing on manifolds,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2013.

[19] R. Salakhutdinov and G. E. Hinton, “Learning a nonlinear embedding by preserving class neighbourhood structure,” in *Proc. Int. Conf. Artificial Intelli. and Stat.*, 2007.

[20] D. Zhang, J. Wang, D. Cai, and J. Lu, “Extensions to self-taught hashing: kernelisation and supervision,” in *Proc. ACM SIGIR Workshop on Feature Generation and Selection for Information Retrieval*, 2010.

[21] X. Li, G. Lin, C. Shen, A. v. d. Hengel, and A. Dick, “Learning hash functions using column generation,” in *Proc. Int. Conf. Mach. Learn.*, 2013.

[22] G. Lin, C. Shen, and J. Wu, “Optimizing ranking measures for compact binary code learning,” in *Proc. Eur. Conf. Comp. Vis.*, 2014.

[23] G. Lin, C. Shen, D. Suter, and A. van den Hengel, “A general two-step approach to learning-based hashing,” in *Proc. Int. Conf. Comp. Vis.*, 2013.

[24] G. Lin, C. Shen, Q. Shi, A. van den Hengel, and D. Suter, “Fast supervised hashing with decision trees for high-dimensional data,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2014.

[25] B. Kulis and K. Grauman, “Kernelized locality-sensitive hashing,” *IEEE T. Pattern Analysis Mach. Intelli.*, 2012.

[26] D. Zhang, J. Wang, D. Cai, and J. Lu, “Self-taught hashing for fast similarity search,” in *Proc. Annual ACM SIGIR Conf.*, 2010.

[27] M. Norouzi, D. M. Blei, and R. Salakhutdinov, “Hamming distance metric learning,” in *Proc. Adv. Neural Info. Process. Syst.*, 2012.

[28] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, “Angular quantization-based binary codes for fast similarity search,” in *Proc. Adv. Neural Info. Process. Syst.*, 2012.

[29] M. A. and C. Perpinan, “The elastic embedding algorithm for dimensionality reduction,” in *Proc. Int. Conf. Mach. Learn.*, 2010.

[30] Z. Li, H. Ning, L. Cao, T. Zhang, Y. Gong, and T. S. Huang, “Learning to search efficiently in high dimensions,” in *Proc. Adv. Neural Info. Process. Syst.*, 2011.

[31] D. Nister and H. Stewenius, “Scalable recognition with a vocabulary tree,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2006.

[32] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, “Object retrieval with large vocabularies and fast spatial matching,” in *Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on*, 2007.

[33] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, “Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization,” *ACM T. Math. Softw.*, 1997.

[34] C. Rother, V. Kolmogorov, V. Lempitsky, and M. Szummer, “Optimizing binary MRFs via extended roof duality,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2007.

[35] J. Besag, “On the statistical analysis of dirty pictures,” *J. of the Royal Stat. Society.*, 1986.

[36] M. Schmidt, “UGM: Matlab code for undirected graphical models,” 2012. [Online]. Available: http://www.di.ens.fr/~mschmidt/Software/UGM.html

[37] Z. Wang, K. Crammer, and S. Vucetic, “Breaking the curse of kernelization: Budgeted stochastic gradient descent for large-scale svm training,” *J. Mach. Learn. Research*, 2012.

[38] R. Appel, T. Fuchs, P. Dollar, and P. Perona, “Quickly boosting decision trees-pruning underachieving features early,” in *Proc. Int. Conf. Mach. Learn.*, 2013.

[39] J. Friedman, T. Hastie, and R. Tibshirani, “Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors),” *The annals of statistics*, 2000.

[40] A. Torralba, R. Fergus, and Y. Weiss, “Small codes and large image databases for recognition,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2008.

[41] M. Guillaumin, T. Mensink, J. Verbeek, and C. Schmid, “Tagprop: Discriminative metric learning in nearest neighbor models for image auto-annotation,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2009.

[42] M. J. Huiskes and M. S. Lew, “The MIR-Flickr retrieval evaluation,” in *Proc. ACM Int. Conf. Multimedia Info. Retrieval*, 2008.

[43] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba, “SUN database: Large-scale scene recognition from abbey to zoo,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2010.

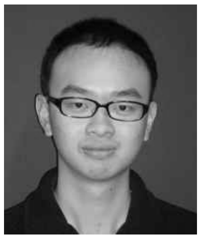
[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proc. IEEE Conf. Comp. Vis. Pattern Recogn.*, 2009.

[45] R. Kiros and C. Szepesvari, “Deep representations and codes for image auto-annotation,” in *Proc. Adv. Neural Info. Process. Syst.*, 2012.

[46] A. Oliva and A. Torralba, “Modeling the shape of the scene: A holistic representation of the spatial envelope,” *Int. J. Comp. Vis.*, 2001.

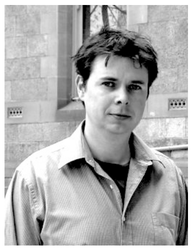
[47] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, “Decaf: A deep convolutional activation feature for generic visual recognition,” in *Proc. Int. Conf. Mach. Learn.*, 2014.

[48] Y. Jia, “Caffe: An open source convolutional architecture for fast feature embedding,” 2013, berkeley Vision and Learning Center techical report.



Guosheng Lin is a Research Fellow at School of Computer Science, The University of Adelaide. He completed his PhD degree at the same university in 2014. His research interests are on computer vision and machine learning. He received a Bachelor degree and a Master degree from the South China University of Technology in computer science in 2007 and 2010 respectively.

Chunhua Shen is a Professor at School of Computer Science, The University of Adelaide. His research interests are in the intersection of computer vision and statistical machine learning. He studied at Nanjing University, at Australian National University, and received his PhD degree from University of Adelaide. In 2012, he was awarded the Australian Research Council Future Fellowship.



Anton van den Hengel is the founding Director of The Australian Centre for Visual Technologies (ACVT). He is a Professor in School of Computer Science, The University of Adelaide. He received a PhD in Computer Vision in 2000, a Master Degree in Computer Science in 1994, a Bachelor of Laws in 1993, and a Bachelor of Mathematical Science in 1991, all from The University of Adelaide.