Fast Supervised Hashing with Decision Trees for High-Dimensional Data

Guosheng Lin, Chunhua Shen*, Qinfeng Shi, Anton van den Hengel, David Suter
The University of Adelaide, SA 5005, Australia

Abstract

Supervised hashing aims to map the original features to compact binary codes that are able to preserve label based similarity in the Hamming space. Non-linear hash functions have demonstrated their advantage over linear ones due to their powerful generalization capability. In the literature, kernel functions are typically used to achieve non-linearity in hashing, which achieve encouraging retrieval performance at the price of slow evaluation and training time. Here we propose to use boosted decision trees for achieving non-linearity in hashing, which are fast to train and evaluate, hence more suitable for hashing with high dimensional data. In our approach, we first propose sub-modular formulations for the hashing binary code inference problem and an efficient GraphCut based block search method for solving large-scale inference. Then we learn hash functions by training boosted decision trees to fit the binary codes. Experiments demonstrate that our proposed method significantly outperforms most state-of-the-art methods in retrieval precision and training time. Especially for high-dimensional data, our method is orders of magnitude faster than many methods in terms of training time.

1. Introduction

Hashing methods construct a set of hash functions that map the original features into compact binary codes. Hashing enables fast search by using look-up tables or hamming distance based ranking. Moreover, compact binary codes are extremely efficient for large-scale data storage. Applications include image retrieval, large-scale object detection [5] and so on.

Hashing methods aim to preserve some notion of similarity (or distance) in the Hamming space. These methods can be roughly categorized as supervised and unsupervised. Unsupervised hashing methods try to preserve the similarity in the original feature space. For example, Locality-Sensitive Hashing (LSH) [7] randomly generates linear hash functions to approximate cosine similarity; Spectral Hashing [26] learns eigenfunctions that preserve Gaussian affinity; Iterative Quantization (ITQ) [8] approximates the Euclidean distance in the Hamming space; and Hashing on manifolds [22] takes the intrinsic manifold structure into consideration.

Supervised hashing is designed to preserve some label-based similarity [13–16]. 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 increasingly attention recently such as Supervised Hashing with Kernels (KSH) [16], Two-Step Hashing (TSH) [15], Binary Reconstructive embeddings (BRE) [13]. Although supervised hashing is more flexible and appealing for real-world applications, the learning is usually much slower than that of unsupervised hashing. Despite the fact that hashing is only of practical interest in the case where it may be applied to large numbers of high-dimensional features, most supervised hashing approaches are demonstrated only on relatively small numbers of low dimensional features. For example, codebook based features have achieved remarkable success on image classification [4, 12], of which the number of feature dimension usually comes to tens of thousands. To exploit this recent advance of feature learning, it is very desirable for supervised hashing to be able to deal with large-scale data efficiently on sophisticated high-dimensional features. To bridge this gap, we propose a supervised hashing method which is able to leverage large training sets and efficiently incorporate with high-dimensional features.

Non-linear hash functions, e.g., the kernel hash function employed in KSH and TSH, have shown much improved performance over the linear hash function. However, kernel functions could be extremely expensive for both training and testing on high-dimensional features. Thus a scalable supervised hashing method with non-linear hash functions is desirable too.

Our main contributions are as follows. (i) We propose to use (ensembles of) decision trees as hash functions for supervised hashing, which can easily deal with a very large number of training data with high dimensionality (tens of thousands), and has the desirable non-linear mapping. To our knowledge, our method is the first general hashing method that uses decision trees as hash functions. (ii) In order to efficiently learn decision trees for supervised hashing,

*Corresponding should be addressed to C. Shen. Code is available at: https://bitbucket.org/chhshen/fasthash/.

Algorithm 1: An example for constructing blocks

Input: Training data points: $\{\boldsymbol{x}_1, ... \boldsymbol{x}_n\}$; Affinity matrix: $\mathbf{Y}$.
Output: blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$
$\mathcal{V} \leftarrow \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$; $t = 0$;
repeat
  $t = t + 1$; $\mathcal{B}_t \leftarrow \emptyset$; $\boldsymbol{x}_i$: randomly selected from $\mathcal{V}$;
  initialize $\mathcal{U}$ as joint of $\mathcal{V}$ and similar examples of $\boldsymbol{x}_i$;
  for *each* $\boldsymbol{x}_j$ *in* $\mathcal{U}$ **do**
    if $\boldsymbol{x}_j$ *is not dissimilar with any examples in* $\mathcal{B}_t$ **then**
      add $\boldsymbol{x}_j$ to $\mathcal{B}_t$; remove $\boldsymbol{x}_j$ from $\mathcal{V}$;
until $\mathcal{V} = \emptyset$;

we apply a two-step learning strategy which decomposes the learning into the binary code inference and the simple binary classification training of decision trees. For binary code inference, we propose sub-modular formulations and an efficient GraphCut [3] based block search method for solving large-scale inference. (iii) Our method significantly outperforms many state-of-the-art methods in terms of retrieval precision. For high-dimensional data, our method is usually orders of magnitude faster in terms of training time.

The two-step learning strategy employed in our method is inspired by the recent work of TSH [15]. Other work in [18, 21, 23] also learns hash functions by training classifiers. The spectral method in TSH for binary code inference does not scale well on large training data, and it may also lead to inferior result due to the loose relaxation of spectral methods. Moreover, TSH only demonstrates satisfactory performance with kernel hash functions on small-scale training data with low dimensionality, which is clearly not practical for large-scale learning on high-dimensional features. In contrast with TSH, we explore efficient decision trees as hash functions and propose an efficient GraphCut based method for binary code inference. Experiments show that our method significantly outperforms TSH.

2. The proposed method

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 of two data point $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$; and $y_{ij} = y_{ji}$. Specifically, $y_{ij} = 1$ if two data points are similar, $y_{ij} = -1$ if dissimilar (irrelevant) and $y_{ij} = 0$ if the pairwise relation is undefined. We aim to learn a set of hash functions to preserve the label based similarity in the Hamming space. m hash functions are denoted as: $\Phi(\boldsymbol{x}) = [h_1(\boldsymbol{x}), ..., h_m(\boldsymbol{x})]$. The output of hash functions are m-bit binary codes: $\Phi(\boldsymbol{x}) \in \{-1, 1\}^m$. Closely related to Hamming distance, the Hamming affinity is calculated by the inner product of two binary codes : $s(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{k=1}^{m} h_k(\boldsymbol{x}_i) h_k(\boldsymbol{x}_j)$. Similar to KSH [16], we formulate hashing learning based on Hamming affinity, which is to encourage positive affinity value of similar data pairs and negative for dissimilar data pairs. The optimization is written as:

$$\min_{\Phi(\cdot)} \sum_{i=1}^{n} \sum_{j=1}^{n} |y_{ij}| \left[m y_{ij} - \sum_{k=1}^{m} h_k(\boldsymbol{x}_i) h_k(\boldsymbol{x}_j) \right]^2. \quad (1)$$

Note that KSH does not include the multiplication of $|y_{ij}|$ in the objective. We use $|y_{ij}|$ to prevent undefined pairwise relation from harming the hashing task. If the relation is undefined, $|y_{ij}| = 0$, otherwise, $|y_{ij}| = 1$. In contrast to KSH which uses kernel functions, here we employ decision trees as hash functions. We define each hash function as a linear combination of decision trees, that is,

$$h(\boldsymbol{x}) = \text{sign}\left(\textstyle\sum_{q=1}^{Q} w_q T_q(\boldsymbol{x})\right). \quad (2)$$

Here Q is the number of decision trees. $T(\cdot) \in \{-1, 1\}$ denotes a tree function with binary output; The weighting $\boldsymbol{w} = [w_1, ..., w_Q]$ and trees $\mathbf{T} = [T_1, ..., T_Q]$ are parameters we need to learn for one hash function. Comparing to kernel method, decision trees enjoy faster testing on high-dimensional data as well as the non-linear fitting ability.

Optimizing (1) directly for learning decision trees is difficult, and the technique used in KSH is no longer applicable. Inspired by TSH [15], we introduce auxiliary variables $z_{k,i} \in \{-1, 1\}$ as the output of the k-th hash function on $\boldsymbol{x}_i$: $z_{k,i} = h_k(\boldsymbol{x}_i)$. Clearly, $z_{k,i}$ is the binary code of i-th data point in the k-th bit. With these auxiliary variables, the problem (1) can be decomposed into two sub-problems:

$$\min_{\mathbf{Z} \in \{-1,1\}^{m \times n}} \sum_{i=1}^{n} \sum_{j=1}^{n} |y_{ij}| \left(m y_{ij} - \sum_{k=1}^{m} z_{k,i} z_{k,j} \right)^2; \quad (3a)$$

$$\min_{\Phi(\cdot)} \ \textstyle\sum_{k=1}^{m} \sum_{i=1}^{n} \delta(z_{k,j} = h_k(\boldsymbol{x}_i)). \quad (3b)$$

Here $\mathbf{Z}$ is the matrix of m-bit binary codes for all training data points. Note that (3a) is a binary code inference problem, and (3b) is a simple binary classification problem. This way, the complicated decision trees learning for supervised hashing (1) now becomes two relatively simpler tasks—solving (3a) (Step 1) and (3b) (Step 2).

Step 1: Binary code inference. For (3a), we sequentially optimize for one bit at a time, conditioning on previous bits. When solving for the k-th bit, the cost in (3a) is written as:

$$\begin{aligned} &\textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} |y_{ij}| (k y_{ij} - \sum_{p=1}^{k} z_{p,i} z_{p,j})^2 \\ =& \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} |y_{ij}| (k y_{ij} - \sum_{p=1}^{k-1} z_{p,i} z_{p,j} - z_{k,i} z_{k,j})^2 \\ =& \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} -2|y_{ij}| (k y_{ij} - \sum_{p=1}^{k-1} z_{p,i} z_{p,j}) z_{k,i} z_{k,j} \\ &+ \mathit{const}. \end{aligned} \quad (4)$$

Hence the optimization for the k-th bit can be equivalently formulated as a binary quadratic problem:

$$\min_{\boldsymbol{z}_k \in \{-1,1\}^n} \ \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} z_{k,i} z_{k,j}, \quad (5a)$$

$$\text{where, } a_{ij} = -|y_{ij}| (k y_{ij} - \textstyle\sum_{p=1}^{k-1} z^*_{p,i} z^*_{p,j}). \quad (5b)$$

Here z^* denotes a binary code in previous bits. We use a stage-wise scheme for solving each bit. Specifically, when

Algorithm 2: Step 1: Block GraphCut for binary code inference

Input: Affinity matrix: $\mathbf{Y}$; bit length: k; max inference iteration; blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$; binary codes: $\{\boldsymbol{z}_1, ..., \boldsymbol{z}_{k-1}\}$.
Output: Binary codes of one bit: $\boldsymbol{z}_k$

```
repeat
    Randomly permute all blocks;
    for each B_i do
        Solve the inference in (8a) on B_i using GraphCut;
until max iteration is reached ;
```

Algorithm 3: FastHash

Input: Training data points: $\{\boldsymbol{x}_1, ...\boldsymbol{x}_n\}$; Affinity matrix: $\mathbf{Y}$; bit length: m; blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$.
Output: Hash functions: $\Phi = [h_1, ..., h_m]$

```
for k = 1, ..., m do
    Step-1: call Algorithm 2 to obtain binary codes of k-th bit;
    Step-2: train trees in (9) to obtain hash function h_k;
    update the binary codes of k-th bit by the output of h_k;
```

solving for the k-th bit, the bit length is set to k instead of m, which is shown in (5b). In this way, the optimization of current bit depends on the loss caused by previous bits, which usually leads to better inference results.

Alternatively, one can apply spectral relaxation method to solve (5a), as in TSH. However solving eigenvalue problems does not scale up to large training sets, and the spectral relaxation is rather loose (hence leading to inferior results). Here we propose sub-modular formulations for the binary code inference problem and an efficient GraphCut based block search method for solving large-scale inference. We first group data points into a number of blocks, then optimize the corresponding variables of one block at a time while conditioning on the rest of the variables. Let $\mathcal{B}$ denote a block of data points. The cost in (5a) can be rewritten as:

$$\begin{aligned}&\textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} z_{k,i} z_{k,j}\\ =&\textstyle\sum_{i\in\mathcal{B}} \sum_{j\in\mathcal{B}} a_{ij} z_{k,i} z_{k,j} + \sum_{i\in\mathcal{B}} \sum_{j\notin\mathcal{B}} a_{ij} z_{k,i} z_{k,j}\\ &\textstyle+ \sum_{i\notin\mathcal{B}} \sum_{j\in\mathcal{B}} a_{ij} z_{k,i} z_{k,j} + \sum_{i\notin\mathcal{B}} \sum_{j\notin\mathcal{B}} a_{ij} z_{k,i} z_{k,j}.\end{aligned}$$

When optimizing for one block, those variables which are not involved in the target block are set to constants. Hence, the optimization for one block can be written as:

$$\min_{\boldsymbol{z}_{k,\mathcal{B}} \in\{-1,1\}^{|\mathcal{B}|}} \textstyle\sum_{i\in\mathcal{B}} \sum_{j\in\mathcal{B}} a_{ij} z_{k,i} z_{k,j} + 2 \sum_{i\in\mathcal{B}} \sum_{j\notin\mathcal{B}} a_{ij} z_{k,i} \hat{z}_{k,j}. \quad (7)$$

Here $\hat{z}_k$ denotes a binary code of the k-th bit which is not involved in the target block. With the definition of a_{ij} in (5b), the optimization for one block can be written as:

$$\min_{\boldsymbol{z}_{k,\mathcal{B}} \in\{-1,1\}^{|\mathcal{B}|}} \sum_{i\in\mathcal{B}} u_i z_{k,i} + \sum_{i\in\mathcal{B}} \sum_{j\in\mathcal{B}} v_{ij} z_{k,i} z_{k,j}, \quad (8a)$$

$$\text{where, } v_{ij} = -|y_{ij}|(k y_{ij} - \textstyle\sum_{p=1}^{k-1} z^*_{p,i} z^*_{p,j}), \quad (8b)$$

$$u_i = -2 \textstyle\sum_{j\notin\mathcal{B}} \hat{z}_{k,j} |y_{ij}| (k y_{ij} - \sum_{p=1}^{k-1} z^*_{p,i} z^*_{p,j}). \quad (8c)$$

Here u_i, v_{ij} are constants. The key to construct a block is to ensure (8a) of such a block is sub-modular, so we can apply efficient GraphCut. We refer to this as Block GraphCut (Block-GC), shown in Algorithm 2. Specifically in our hashing problem, by leveraging similarity information, we can easily construct blocks which meet the sub-modular requirement, as shown in the following proposition:

Proposition 1. *$\forall i, j \in \mathcal{B}$, if $y_{ij} \geq 0$, the optimization in (8a) is a sub-modular problem. In other words, for any data point in the block, if it is not dissimilar with any other data points in the block, then (8a) is sub-modular.*

Proof. If $y_{ij} \geq 0$, $k y_{ij} \geq \sum_{p=1}^{k-1} z^*_{p,i} z^*_{p,j}$ holds. Thus $v_{ij} = -|y_{ij}|(k y_{ij} - \sum_{p=1}^{k-1} z^*_{p,i} z^*_{p,j}) \leq 0$. Let $\theta_{ij}(z_{k,i}, z_{k,j}) = v_{ij} z_{k,i} z_{k,j}$, we have $\theta_{ij}(-1,1) = \theta_{ij}(1,-1) = -v_{ij} \geq 0; \theta_{ij}(1,1) = \theta_{ij}(-1,-1) = v_{ij} \leq 0$. Hence $\forall i, j \in \mathcal{B}, \theta_{ij}(1,1) + \theta_{ij}(-1,-1) \leq 0 \leq \theta_{ij}(1,-1) + \theta_{ij}(-1,1)$, which prove the sub-modularity of (8a) [19]. □

Blocks can be constructed in many ways as long as they satisfy the condition in Proposition 1. A simple greedy method is shown in Algorithm 1. Note that the blocks can overlap and the union of them needs to cover all n variables. If one variable is one block, Block-GC becomes ICM [2,20] which optimizes for one variable at a time.

Step 2: Learning boosted trees as hash functions. For binary classification in (3b), usually the zero-one loss is replaced by some convex surrogate loss. Here we use the exponential loss which is common for boosting methods. The classification problem for learning the k-th hash function is written as:

$$\min_{\boldsymbol{w}\geq 0} \textstyle\sum_{i=1}^{n} \exp\Big[- z_{k,i} \sum_{q=1}^{Q} w_q T_q(\boldsymbol{x}_i)\Big]. \quad (9)$$

We apply Adaboost to solve above problem. In each boosting iteration, a decision tree as well as its weighting coefficient are 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, we are doing 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, which are able to significantly speed up the training. Here we summarize some techniques that are included in our implementation: (i) We have used the highly efficient stump implementation proposed in the recent work of [1], which is around 10 times faster than conventional stump implementation. (ii) Feature quantization can significantly speed up tree training without performance loss in practice, and also largely reduce the memory consuming. As in [1], we linearly quantize feature values into 256 bins. (iii) We apply the weight-trimming technique described in [1,6]. In each boosting iteration, the smallest 10% weightings are trimmed (set to 0). (iv) We apply the LazyBoost technique: only a random subset of feature di-

Table 1: Comparison of KSH and our FastHash. KSH results with different number of support vectors. Both of our FastHash and FastHash-Full outperform KSH by a large margin 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**

mensions are evaluated for tree node splitting.

Finally, we summarize our hashing method (FastHash) in Algorithm 3. In contrast with TSH, we alternate Step-1 and Step-2 iteratively. For each bit, the binary code is updated by applying the learned hash function. Hence, the learned hash function is able to make a feedback for binary code inference of next bit, which may lead to better performance.

3. Experiments

We here describe the results of comprehensive experiments carried out on several large image datasets in order to evaluate the proposed method in terms of training time, binary encoding time and retrieval performance. For decision tree learning in our FastHash, if not specified, the tree depth is set to 4, and the number of boosting iterations is set to 200. We compare to a number of recent supervised and unsupervised hashing methods. The retrieval performance is measured in 3 ways: the precision of top-K ($K = 100$) retrieved examples (denoted as Precision), mean average precision (MAP) and the area under the Precision-Recall curve (Prec-Recall). Results are reported on 5 image datasets which cover a wide variety of images. The dataset CIFAR10[1] contains $60,000$ images. The datasets IAPRTC12 and ESPGAME [9] contain around $20,000$ images, and MIRFLICKR [11] is a collection of $25,000$ images. SUN397 [27] is a large image dataset which contains more than $100,000$ scene images form 397 categories.

For the multi-class datasets: CIFAR10 and SUN397, the ground truth pairwise similarity is defined as multi-class label agreement. For datasets: IAPRTC12, ESPGAME and MIRFLICKR, of which the keyword (tags) annotation are provided in [9], two images are treated as semantically similar if they are annotated with at lease 2 identical keywords (or tags). Following a conventional setting in [13, 16], a large portion of the dataset is allocated as an image database for training and retrieval and the rest is put aside for testing queries. Specifically, for CIFAR10, IAPRTC12, ESPGAME and MIRFLICKER, the provided splits are used; for SUN397, 8000 images are randomly selected as test queries, while the remaining 100417 images form the training set. If not specified, 64-bit binary codes are generated using comparing methods for evaluation.

We extract codebook-based features following the conventional pipeline from [4, 12]: 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. We also tested increasing the codebook size to 1600 which results in 22400-dimensional features.

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

Table 2: Comparison of TSH and our FastHash for binary code inference in Step 1. The proposed Block GraphCut (Block-GC) achieves much lower objective value and also takes less inference time than the spectral method, and thus performs much better.

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

3.1. Comparison with KSH

KSH [16] has been shown to outperform many state-of-the-art comparators. The fact that our method employs the same loss function as KSH thus motivates further comparison against this key method. KSH employs a simple kernel technique by predefining a set of support vectors then learning linear weightings for each hash function. In the works of [15, 16], KSH is evaluated only on low dimensional GIST features (512 dimensions) using a small number of support vectors (300). Here, in contrast, we evaluate KSH on high-dimensional codebook features, and vary the number of support vectors from 300 to 3000. KSH is trained on a sampled set of 5000 examples. The results of these tests are summarized in Table 1, which shows that increasing the number of support vectors consistently improves the retrieval performance of KSH. However, even on this small

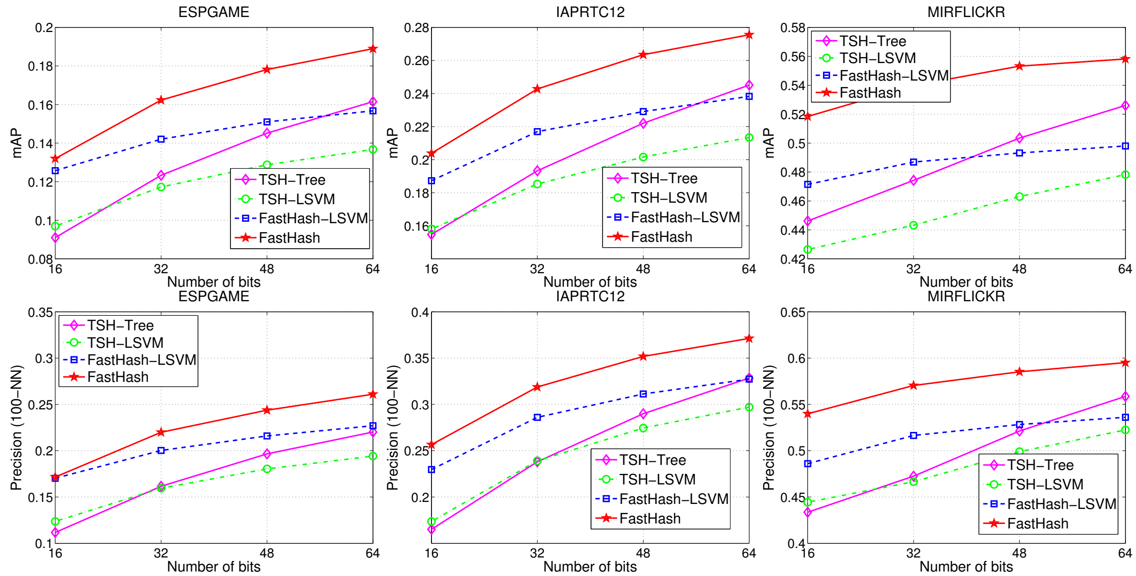

Figure 1: Comparison of various combinations of hash functions and binary inference methods. Note that the proposed FastHash uses decision tree as hash functions. The proposed decision tree hash function performs much better than the linear SVM hash function. Moreover, our FastHash performs much better than TSH when using the same hash function in Step 2.

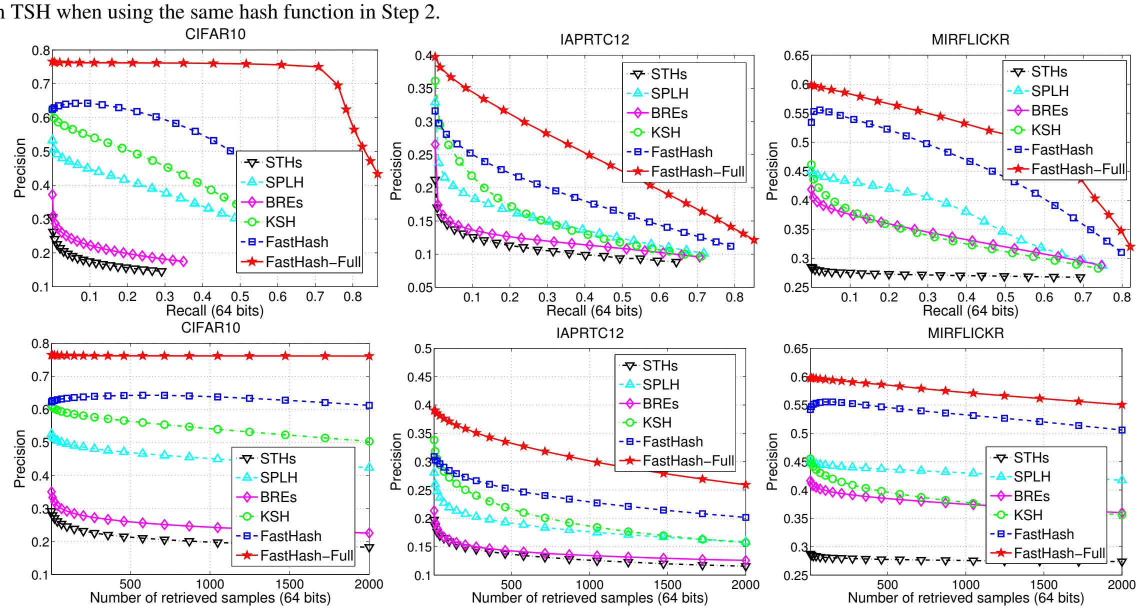

Figure 2: Results on high-dimensional codebook features. The precision and recall curves are given in the first row. The precision curves of the top 2000 retrieved examples are given on the second row. Both our FastHash and FastHash-Full outperform their comparators by a large margin.

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). Our FastHash and FastHash-Full outperform KSH by a large margin both in terms of training speed and and retrieval precision. The results also show that the decision tree hash functions in FastHash are much more efficient for testing (binary encoding) than the kernel function in KSH. Our FastHash is orders of magnitude faster than KSH on training, and thus much better suited to large training sets and high-dimensional data. For the low-dimensional GIST feature, our FastHash also performs much better than KSH in retrieval, see Table 3 for details. The retrieval performance is also plotted in Fig. 2. If not specified, the number of support vectors for KSH is set to 3000.

3.2. Comparison with TSH

The proposed FastHash employs a similar two-step approach to that of TSH [15]. We first compare binary code inference in Step 1: the proposed Block GraphCut (Block-GC) and the spectral method in TSH. The iteration number of Block-GC is set to 2. The results of testing are summarized in Table 2. We construct blocks using Algorithm

Table 3: Results using two types of features: low-dimensional GIST features and the high-dimensional codebook features. Our FastHash and FastHash-Full outperform the comparators by a large margin 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	0.160	0.124	0.084	11740	2	0.148	0.104	0.074
STHs	5000	616	1	0.099	0.092	0.062	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**

1. The average 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 [2, 20] 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. The proposed Block-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 the training set size.

We now provide results comparing different combinations of hash functions (Step 2) and binary code inference methods (Step 1). We evaluate the linear SVM and the proposed decision tree hash functions with different binary code inference methods (Spectral method in TSH and Block-GC in FastHash). The 11200-dimensional codebook features are used here. The retrieval performance is shown in Fig. 1 by varying the number of bits. As expected, the proposed decision tree hash function performs much better than linear SVM hash function. It also shows that our FastHash performs much better than TSH when using the same type of hash function for Step 2 (decision tree or linear SVM hash function), which indicates that the proposed Block-GC method for binary code inference and the stage-wise learning strategy is able to generate high quality binary codes. We also can train RBF-kernel SVM as hashing function in Step 2, however, as the case here, when applied on large training set and high-dimensional data, the training of RBF SVM almost become intractable. Even using the stochastic kernel SVM (BSGD) [25] with a support vector budget, the training and testing cost are still very expensive.

3.3. Comparison on different features

We compare hashing methods on the the low-dimensional (320 or 512) GIST feature and the high-dimensional (11200) codebook feature. We extract GIST features of 320 dimensions for CIFAR10 which contains low resolution images, and 520 dimensions for other datasets. Several state-of-the-art supervised methods are included in this comparison: KSH [16], Supervised Self-Taught Hashing (STHs) [28], and Semi-supervised Hashing (SPLH) [24]. The result is presented in Table 3. The codebook features consistently show better result than GIST features. Comparing methods are trained on a sampled training set (5000 examples). Results show that comparing methods can be efficiently trained on the GIST features. However, when applied on high dimensional features, even on a small training set (5000), their training time dramatically increase. Large matrix multiplication and solving eigenvalue problem on a large matrix may account for the expensive computation in these comparing methods. It would be 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

Table 4: Results of methods with dimension reduction. KSH, SPLH and STHs are trained with PCA feature reduction. Our FastHash outperforms others by a large margin on retrieval performance.

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**

on the whole dataset. FastHash and FastHash-Full outperform others by a large margin both on GIST and codebook features. The training of FastHash is also orders of magnitudes faster than others on the high-dimensional codebook features. The retrieval performance on codebook features is plotted in Fig. 2.

3.4. Comparison with dimension reduction

A possible way to reduce the training cost on high-dimensional data is to apply dimension reduction. For comparing methods: KSH, SPLH and STHs, here we reduce the original 11200-dimensional codebook features to 500 dimensions by applying PCA. We also compare to CCA+ITQ [8] which combines ITQ with the supervised dimensional reduction. Our FastHash still use the original high-dimensional features. The result is summarized in Table 4. After dimension reduction, most comparing methods can be trained on the whole training set within 24 hours (except KSH on CIFAR10). However it still much slower than our FastHash. The retrieval performance of most methods get improved with more training data. Our FastHash still significantly outperforms all others. The proposed 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 method with dimensional reduction.

3.5. Comparison with unsupervised methods

We compare to some popular unsupervised hashing methods: LSH [7], ITQ [8], Anchor Graph Hashing (AGH) [17], Spherical Hashing (SPHER) [10], MDSH [26]. The retrieval performance is shown in Fig. 3. Unsupervised methods perform poorly for preserving label based similarity. Our FastHash outperforms others by a large margin.

Table 5: Performance of our FastHash on more features (22400 dimensions) and more bits (1024 bits). It 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 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

3.6. 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. The result is shown in Table 5. It shows that our FastHash can be efficiently trained on high-dimensional features with large bit length. The training and binary coding time (Test time) of FastHash is only linearly increased with the bit length.

3.7. Large dataset: SUN397

The challenging SUN397 dataset is a collection of more than $100,000$ scene images from 397 categories. 11200-dimensional codebook features are extracted on this dataset. We compare with a number of supervised and unsupervised methods. The depth for decision trees is set to 6. The result is presented in Table 6 Supervised methods: KSH, BREs, SPLH and STHs are trained to 64 bits on a subset of 10K examples. However, even on this sampled training set and only run to 64 bits, the training of these methods are already impractically slow. It would be almost intractable for the whole training set and long bit length. Short length of bits are not able to achieve good performance on this challenging dataset. In contrast, our method can be efficiently trained to large bit length (1024 bits) on the whole training set (more than $100,000$ training examples). FastH-N is our FastHash using weighted sampling of examples (5000 examples) for tree node splitting. FastH-N may consume more training time due to less pruning based on minimum node size. Both of our FastHash and FastH-N outperform other methods by a large margin on retrieval performance.

For memory usage, many of the comparing methods require a large amount of memory for large matrix multiplication. In contrast, the decision tree learning in our method

Table 6: Results on the large image dataset SUN397 using 11200-dimensional codebook features. Our FastHash can be efficiently trained to large bit length (1024 bits) on this large training set. Both of our FastHash and FastH-N outperform other methods by a large margin on retrieval performance.

Method	#Train	Bits	Train time	Test time	Precision	MAP	Method	#Train	Bits	Train time	Test time	Precision	MAP
						SUN397							
KSH	10000	64	57045	463	0.034	0.023	ITQ	100417	1024	1686	127	0.030	0.021
BREs	10000	64	105240	23	0.019	0.013	SPHER	100417	1024	35954	121	0.039	0.024
SPLH	10000	64	27552	14	0.022	0.015	LSH	100417	1024	99	99	0.028	0.019
STHs	10000	64	22914	14	0.010	0.008	CCA+ITQ	100417	1024	15580	127	0.120	0.081
CCA+ITQ	100417	512	7484	66	0.113	0.076	**FastH**	100417	1024	62076	536	**0.165**	**0.163**
FastH	100417	512	29624	302	0.149	0.142	**FastH-N**	100417	1024	71203	749	**0.177**	**0.184**

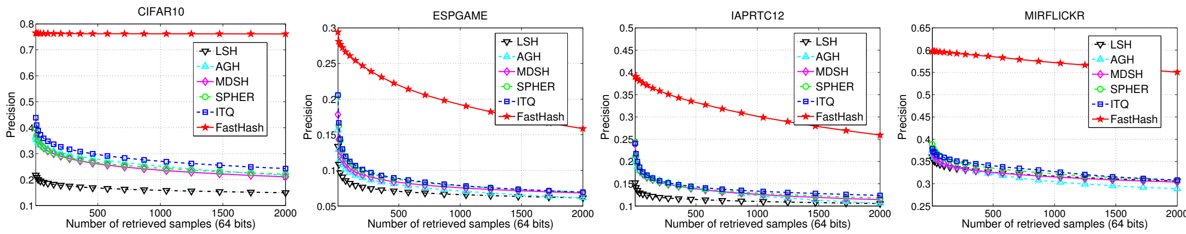

Figure 3: The retrieval precision results of unsupervised methods. Unsupervised methods perform poorly for preserving label based similarity. Our FastHash outperform others by a large margin.

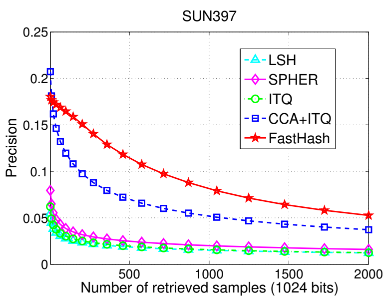

Figure 4: The precision curve of top 2000 retrieved examples on large image dataset SUN397 using 1024 bits. Here we compare with those methods which can be efficiently trained up to 1024 bits on the whole training set. Our FastHash outperforms others by a large margin.

only involves the simple comparison operation on quantized feature data (256 bins), thus FastHash only consumes less than 7GB for training, which shows that our method can be easily applied for large-scale training.

4. Conclusion

We have proposed an efficient supervised hashing method, which uses decision tree based hash functions and GraphCut based binary code inference. Our comprehensive experiments show the advantages of our method on retrieval performance and fast training for high-dimensional data, which indicates its practical significance on many potential applications like large-scale image retrieval.

References

[1] R. Appel, T. Fuchs, P. Dollár, and P. Perona. Quickly boosting decision trees-pruning underachieving features early. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2013. 3

[2] J. Besag. On the statistical analysis of dirty pictures. *J. of the Royal Stat. Society.*, 1986. 3, 6

[3] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE T. Pattern Analysis Mach. Intelli. (TPAMI)*, 2001. 2

[4] A. Coates and A. Ng. The importance of encoding versus training with sparse coding and vector quantization. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2011. 1, 4

[5] 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. (CVPR)*, 2013. 1

[6] 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. 3

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. Int. Conf. Very Large Data Bases (VLDB)*, 1999. 1, 7

[8] 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. (TPAMI)*, 2012. 1, 7

[9] 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. (CVPR)*, 2009. 4

[10] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In *Proc. IEEE Conf. Comp. Vis. Pattern Recogn. (CVPR)*, 2012. 7

[11] M. J. Huiskes and M. S. Lew. The MIR-Flickr retrieval evaluation. In *Proc. ACM Int. Conf. Multimedia Info. Retrieval*, 2008. 4

[12] R. Kiros and C. Szepesvári. Deep representations and codes for image auto-annotation. In *Proc. Adv. Neural Info. Process. Syst. (NIPS)*, 2012. 1, 4

[13] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In *Proc. Adv. Neural Info. Process. Syst. (NIPS)*, 2009. 1, 4

[14] X. Li, G. Lin, C. Shen, A. van den Hengel, and A. Dick. Learning hash functions using column generation. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2013. 1

[15] 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. (ICCV)*, 2013. 1, 2, 4, 5

[16] W. Liu, J. Wang, R. Ji, Y. Jiang, and S. Chang. Supervised hashing with kernels. In *Proc. IEEE Conf. Comp. Vis. Pattern Recogn. (CVPR)*, 2012. 1, 2, 4, 6

[17] W. Liu, J. Wang, S. Kumar, and S. F. Chang. Hashing with graphs. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2011. 7

[18] M. Rastegari, A. Farhadi, and D. Forsyth. Attribute discovery via predictable discriminative binary codes. In *Proc. Eur. Conf. Comp. Vis. (ECCV)*, 2012. 2

[19] C. Rother, V. Kolmogorov, V. Lempitsky, and M. Szummer. Optimizing binary MRFs via extended roof duality. In *Proc. IEEE Conf. Comp. Vis. Pattern Recogn. (CVPR)*, 2007. 3

[20] M. Schmidt. http://www.di.ens.fr/~mschmidt/Software/UGM.html. 3, 6

[21] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *Proc. Int. Conf. Comp. Vis. (ICCV)*, 2003. 2

[22] 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. (CVPR)*, 2013. 1

[23] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Proc. IEEE Conf. Comp. Vis. Pattern Recogn. (CVPR)*, 2008. 2

[24] J. Wang, S. Kumar, and S. Chang. Semi-supervised hashing for large scale search. *IEEE T. Pattern Analysis Mach. Intelli. (TPAMI)*, 2012. 6

[25] 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 (JMLR)*, 2012. 6

[26] Y. Weiss, R. Fergus, and A. Torralba. Multidimensional spectral hashing. In *Proc. Eur. Conf. Comp. Vis. (ECCV)*, 2012. 1, 7

[27] 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. (CVPR)*, 2010. 4

[28] D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. In *Proc. Annual ACM SIGIR Conf.*, 2010. 6