Supervised Hashing Using Graph Cuts and Boosted Decision Trees

By 李欣

Introduction

最常见的 hash 函数是线性感知机函数，优化的过程也因 hash 函数选择的不同而不同，不同的 hash 函数都试图在测试时间以及拟合能力之间寻找平衡。因为优化流程因 hash 函数的选择而不同，因此对于每一个 hash 函数都需要不同的优化技巧。
本文中提出的框架，可以适应任一个定义在数据对上的汉明距离/亲和度的损失函数。这是因为，本文将学习过程分解为两个部分，二进制码推理以及 hash 函数学习。我们可以将任何定义在汉明距离/亲和度上的损失问题用二元二次问题来表示，这样以来不同类型的损失函数都可以统一为同一个优化问题，这极大简化了优化过程。
同时，这样分解问题之后，hash 函数学习过程变成了一个二分类问题。通过简单训练二分类器我们可以学习各种类型的哈希函数，比如感知机、决策树哈希函数等。
许多监督 hash 方法需要复杂的优化过来直接学习到 hash 函数，也因此只能适应于小规模的训练数据集。在我们的方法中，我们提出一个基于块搜索算法的高效**图切割**技术，来应对大规模训练数据情况下的**二进码推理**问题。对于训练数据的高维问题，现有的方法在训练时无法取得令人满意的效果。在本文，我们提出来学习一个**决策树**作为非线性映射的**哈希函数**。决策树仅仅涉及到简单的比较操作，因为就算训练数据的维度高，本文中的方法也一样可以取得比较不错的效果。

Contribution

1、 提出一个相对灵活且高效的哈希框架，这个框架可以将不同类型的损失函数和哈希函数结合起来。将学习过程分解为两个部分，二进制码生成以及哈希函数的学习。这个分解使得哈希函数的学习过程简化为二分类问题。我们发现不同类型的损失函数都可以统一为二元二次问题，因此可以通过求解一个标准的二元二次问题来解决二进码生成问题。
2、 对于二进制码生成，我们提出子模块以及以块搜索方法为基础的图切割技术来解决大规模二进制码生成问题。
3、 提出使用决策树作为 hash 函数用作监督哈希，在处理大规模高维数据时可以取得较好的效果，并且可以进行非线性映射。
4、 本文中所提到的方法在检索准确度上比现有的方法更准确，对于高维数据本方法在训练时的速度可以提高很多。

本文中方法实现的代码：https://bitbucket.org/chhshen/fasthash/（我还没有去实现这个算法，计划最近几日学学 MatLab，去实现这个方法）

Flexible Two-Step Hashing

给定训练数据集 $X=\{x_1,\cdots\cdots,x_n\}\subset R^d$，成对监督信息，亲和力矩阵 Y，其中 $y_{ij}=1$ 表示点 x_i 与点 x_j 相似，$y_{ij}=-1$ 表示点 x_i 与点 x_j 不相似，$y_{ij}=0$ 表示点 x_i 与点 x_j 之间关系未定义。目标是学习一个 hash 函数集合，将数据点映射成二进制哈希码的同时，保证其在原始空间中的相似性没有被破坏。如：现有 m 个哈希函数，每个函数的输出 $h(x)\in\{-1,1\}$.则 m 个哈希函数的输出结果用$\Phi(x)$表示，则：$\Phi(x)=[h1(x),h2(x)\ldots\ldots,hm(x)]$。是一个 m 位二进制向量$\Phi(x)\in\{-1,1\}^m$.

哈希学习过程中，损失函数是定义在数据对之间的汉明距离/亲和度。其中**汉明距离**的定义为：

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

其中$\delta(.)\in\{0,1\}$，即若括号内内容为真则输出为 1，括号中内容为假则输出为 0。更直白的说汉明距离就是用来对比两个相同长度的二进制码，若对应位位码不同则距离加 1，汉明距离就是表示两个二进制码中不同位数的个数。

汉明亲和度：

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

即两个二进制码的内积。

我们通过优化下面一个式子来进行哈希函数的学习：

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

(4)

$\delta(y_{ij}\neq 0)\in\{0,1\}$用来判断两个数据点之间的关系是否已经定义了。L(.)是损失函数，来衡量二进制码和真实值y_{ij}之间的差距。然而大多数方法都是通过直接优化上式来学习到 hash 函数的参数。这就意味着优化过程和 hash 函数的使用息息相关，若要更换其他类型的 hash 函数就会变得非常麻烦。此外，该式的优化并不是很简单。

如 KSH 采用的损失函数：

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

(5)

其中$\delta(y_{ij}\neq 0)$是用来防止未定义的数据点之间的关系影响训练目标。在 KSH 中使用的 hash 函数为核函数：

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

其中，$\chi' = \{x'_1, x'_2, \ldots, x'_Q\}$是 Q 的支持向量，KSH 通过直接解决（5）式中的优化问题来学习到哈希函数。如果我们选用其它形式的哈希函数，那么 KSH 的优化函数就不一定适应了。

下面简要介绍本文所提出的框架，该框架可以将各种形式的损失函数和哈希函数组合起来。根据 [26] 的启发，我们将学习过程分解为两个部分，二进制码的生成以及哈希函数的学习。下面引入辅助变量$z_{r,i} \epsilon \{-1,1\}$作为第 r 个哈希函数对于 x_i 的输出。

$$z_{r,i} = h_r(xi)$$

显然$z_{r,i}$表示第 i 个输入的数据点的第 r 位二进制。有了这个辅助变量，（4）式可以化简为两个子问题：

$$\min_{\mathbf{Z}} \sum_{i=1}^{n}\sum_{j=1}^{n} \delta(y_{ij} \neq 0) L(\boldsymbol{z}_i, \boldsymbol{z}_j; y_{ij}), \tag{8a}$$

$$\text{s.t.} \quad \mathbf{Z} \in \{-1, 1\}^{m \times n}; \tag{8b}$$

and,

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

其中 Z 为 n 个数据点的 m 位二进制码，z_i是第 i 个数据点的二进制码。这样，（4）式的 hash 学习问题就转化为两个相对简单的任务（8）和（9）。显然，（8）式是为了解决二进制码问题，（9）式为了解决简单二分类问题。

我们在上一位的基础上一次一个地调整位。我们是交叉执行这两个步骤的，并不是完全将其分开的。当解决了一位之后，通过运用哈希函数来更新二进制码。这种按位进行优化的策略简化了优化过程，并且一个哈希函数学习错误可以在下一位学习中得到弥补。

本文的方法被称为 FastHash，算法如下：

Algorithm 1: FastHash (flexible two-step hashing)

Input: training data points: $\{\boldsymbol{x}_1, \ldots \boldsymbol{x}_n\}$; affinity matrix: $\mathbf{Y}$; bit length: m.

Output: hash functions: $\Phi = [h_1, \ldots, h_m]$.

1 Initialization: construct blocks:$\{\mathcal{B}_1, \mathcal{B}_2, \ldots\}$ for Block GraphCut, Algorithm 2 shows an example;

2 **for** $r = 1, \ldots, m$ **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: $\{x_1, ...x_n\}$; affinity matrix: $\mathbf{Y}$.
Output: blocks:$\{\mathcal{B}_1, \mathcal{B}_2, ...\}$.

$\mathcal{V} \leftarrow \{x_1, ..., x_n\}$; $t = 0$;
repeat
  $t = t + 1$; $\mathcal{B}_t \leftarrow \emptyset$;
  Randomly selected x_i from $\mathcal{V}$;
  Initialize $\mathcal{U}$ as the joint set of $\mathcal{V}$ and similar examples of x_i ;
  for *each* x_j *in* $\mathcal{U}$ **do**
    if x_j *is not dissimilar with any examples in* $\mathcal{B}_t$ **then**
      add x_j to $\mathcal{B}_t$;
      remove x_j from $\mathcal{V}$;
until $\mathcal{V} = \emptyset$;

Step1: Binary Code Inference

当求解第 r 位二进制码时，前 r-1 位已经确定了，则二进制码推理问题可以表示为：

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

其中 z(r)表示 n 个训练数据点的第 r 位取值向量。l_r表示产生第 r 位二进制码的损失函数，其取值取决于前一位：

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

其中，z_i^{r-1}表示第 r 个数据点的前 r-1 位二进制码。
根据以下命题，我们可以将任意以汉明距离/亲和度为基础的损失函数写成一个标准的二元二次问题。

Proposition1：

对于任意一个定义在一对二进制输入变量 z1,z2∈ {−1,1}的损失函数 l(z1,z2), 且 l(-1,-1)=l(1,1),l(1,-1)=l(-1,1)。我们可以定义一个与 l(z1,z2)相等的二次函数 g(z1,z2)。有如下等式：

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

其中，l^{11}, l^{-11}是常数，且$l^{11} = l(1,1), l^{-11} = l(-1,1)$。
这个命题可以通过穷举所有类型的输入来得到证明。而事实上只有两种类型的输入。当 z1=1,z2=1 的时候：

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

当 z1=-1,z2=1 时：

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

从上面的等式可以得出结论，即函数 l(~.~)与函数 g(~.~)对任意的输入都有相同的输出。
基于这个命题，则（10）式可以化简为：

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

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

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

其中$a_{i,j}$是一个常数，以上优化问题是一个没有限制的二元二次问题，可以用矩阵的形式来表示：

$$
\begin{aligned}
&\min_{\boldsymbol{z}_{(r)}} \boldsymbol{z}_{(r)}^{\top} \mathbf{A} \boldsymbol{z}_{(r)}, \qquad (14a) \\
&\text{s.t. } \boldsymbol{z}_{(r)} \in \{-1,1\}^n. \qquad (14b)
\end{aligned}
$$

其中矩阵 A 中的(i,j)个元素就是（13b）中的 $a_{i,j}$。这样，（10）式原始求解每一位的优化问题，就可以写成（13a）式中的二元二次问题(BQP)。

Spectral method for binary inference

为了解决（14a）中的 BPQ 问题来获得一个二进制码，首先介绍了一个介于谱松弛（Spectral Relaxation）方法，然后介绍一个基于高效图切割的方法来进行大规模的二进制码生成。
谱松弛方法会降低二元约束，优化的问题变成：

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

上式的优化解释矩阵 A 最小特征值对应的特征向量。为了获得更好的结果值，我们可以解决如下优化问题（14a 式的 relaxed 形式）：

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

我们使用（23 式）的谱松弛结果$z_{(r)}^0$作为一个初始化值，并且使用了 LBFGS-B solver[33]解决了上述问题。然后将解决方案设置为 0 以输出最终的二进制代码。

Block GraphCut for binary code inference

已经证明使用谱方法可以解决（14a）中的二进制码推理问题，但是在求解特征值的时候，如果训练集过大，该方法就不太可能产生较好的效果。这里，我们就进一步提出了子模块化公式和一种基于高效图割的块搜索方法来解决大规模推理问题。块搜索方法可以比谱方法取得更好的结果。

具体来说，我们首先将数据点分成若干块，然后迭代优化这些块，直至收敛。在每次迭代中，我们随机挑选一个块，然后针对（更新）该块的相应二进制变量进行优化，调整其余变量。换句话说，当对一个块进行优化时，只有与目标块的数据点对应的二进制变量才会被更新；而对于目标块中没有涉及的变量，它们的值保持不变。 显然每个区块更新都会严格降低目标。

形式上，让 B 表示一个数据点块。 我们希望为块 B 的相应二进制变量进行优化。我们用$\hat{z}_r$表示在 r 位中没有涉及目标块的二进制代码。 首先我们重写（13a）中的目标，将目标块的变量与其他变量分开。 （13a）中的目标可以改写为：

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j} z_{r,i} z_{r,j} \tag{25a}$$
$$= \sum_{i\in\mathcal{B}}\sum_{j\in\mathcal{B}} a_{i,j} z_{r,i} z_{r,j} + \sum_{i\in\mathcal{B}}\sum_{j\notin\mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j}$$
$$+ \sum_{i\notin\mathcal{B}}\sum_{j\in\mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j} + \sum_{i\notin\mathcal{B}}\sum_{j\notin\mathcal{B}} a_{i,j} \hat{z}_{r,i} \hat{z}_{r,j} \tag{25b}$$
$$= \sum_{i\in\mathcal{B}}\sum_{j\in\mathcal{B}} a_{i,j} z_{r,i} z_{r,j} + 2\sum_{i\in\mathcal{B}}\sum_{j\notin\mathcal{B}} a_{i,j} z_{r,i} \hat{z}_{r,j}$$
$$+ \sum_{i\notin\mathcal{B}}\sum_{j\notin\mathcal{B}} a_{i,j} \hat{z}_{r,i} \hat{z}_{r,j}. \tag{25c}$$

在对一个块进行优化时，那些不涉及目标块的变量将被视为常量；因此^ zr 被视为常量。通过去除常量部分，一个块的优化是：

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

我们的目标是优化$z_{r,B}$，它是涉及目标块 B 的变量的向量。将常数$a_{i,j}$通过其在（13b）中的定义代入，上述优化被写为：

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

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

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

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: $\boldsymbol{z}_r$.

```
repeat
    Randomly permute all blocks;
    for each B_i do
        Solve the inference in (27a) on B_i using graph cuts;
until max iteration is reached;
```

这里的u_i, v_{ij}都是常数。构建块的关键是确保（27a）这个块是子模块化的，因此我们能够应用高效的图形切割方法。 我们将此称为 Block GraphCut（Block-GC），如算法 3 所示。具体来说，在我们的哈希问题中，通过利用相似性信息，我们可以轻松构建块以满足子模块性要求。我们假设损失函数满足 以下条件：

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

这直观地意味着对于两个相似的数据点，为一个位分配相同的二进制值的损失小于分配不同的二进制值。 由于损失函数总是鼓励两个相似的数据点具有相似的二进制代码，所以这个条件当然可以满足。

下面这个命题定义如何来构建这个块：

Proposition 2

∀i, j ∈ B，如果y_{ij}>=0，那么在（27a）中的优化问题就是子模块问题。换句话说，对块中的

任一个数据点如果它和块中的其他数据点都相似，那么（27a）就是一个子模块。
这个命题也很容易得到证明。
只要满足命题 2 中的条件，块就可以以多种方式构造。在算法 2 中示出了简单的贪婪方法。
注意，一个块可以与另一个块重叠，并且所有块的联合需要覆盖所有 n 个变量。 如果一个块只包含一个变量，则 Block-GC 将成为 ICM 方法（[35]，[36]），一次优化一个变量。

Step 2 Hash Function Learning

第二步是解决学习一个散列函数的二元分类问题。 将第一步中获得的二进制代码用作分类标签。 任何二元分类器（例如决策树，神经网络）和任何先进的大规模训练技术都可以直接应用于散列函数学习。 对于第 r 位，分类问题是：

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

通常情况下，上述问题中的 0-1 损失函数会被替换成凸代理损失（convex surrogate loss），在学习哈希函数一位后，通过应用学习哈希函数来更新二进制代码。因此，学习哈希函数能够影响下一位的学习。

提升树作为 hash 函数

具有非线性映射的散列函数的决策树可能是一个很好的选择。 与核方法相比，决策树只涉及简单的评估操作; 因此它们对于测试更有效率，特别是对于高维数据。 我们将一个哈希函数定义为树的线性组合：

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

其中 Q 为所构建的树的数目，T(.)∈ $\{-1,1\}$表示一个有二进制输出的树。
我们训练增强分类器来学习一个散列函数的加权系数和树。 第 r 个哈希函数的分类问题写为：

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

我们应用 Adaboost 来解决问题。在每次增强迭代中，学习决策树及其加权系数。二叉决策树的每个节点都是决策树。培训一个残端（stump）是找到一个特征维度和阈值，使加权分类错误最小化。从这个角度来看，我们同时进行特征选择和哈希函数学习。我们可以很容易地使用现有的文献中提供的高效决策树学习技术。

Experiment

为了验证本文提出的框架的有点及可用性，本文在实验部分在几个较大的图数据集上进行大量的实验。标准的选择包括训练时间、二进制编码时间以及检索的准确度。为了检验本文的方法在不同情景下的有效性，实验使用了不同二进制码生成方法，采用了大量不同的损失函数，hash 函数。

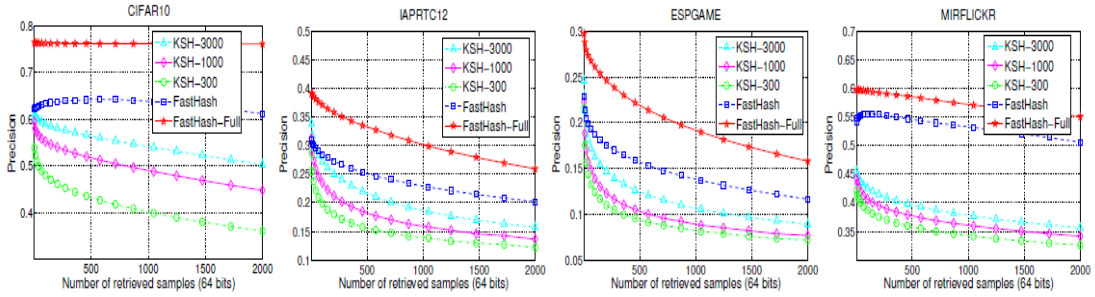

Fig. 2 – Comparison of KSH and our FastHash on all datasets. The number after "KSH" is the number of support vectors. Both of our FastHash and FastHash-Full significantly outperform KSH.

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**

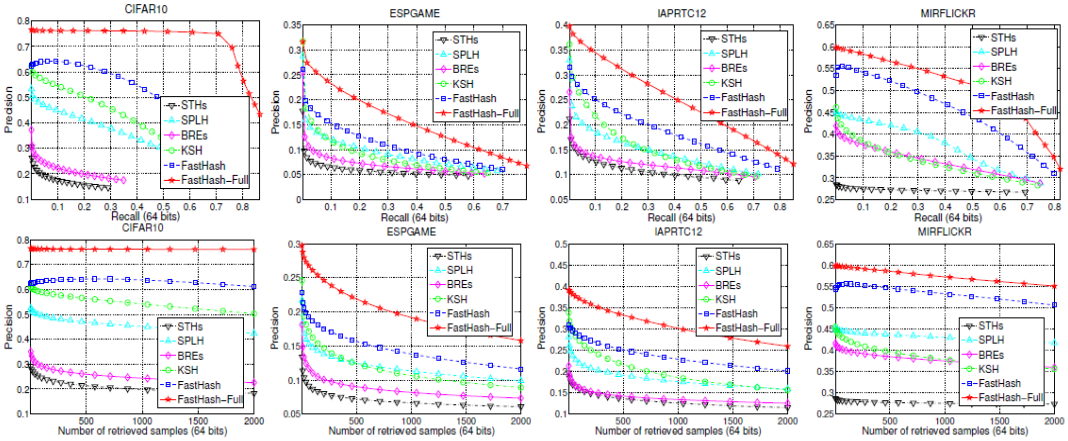

Fig. 3 – Results on high-dimensional codebook features. Our FastHash significantly outperform others.

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

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

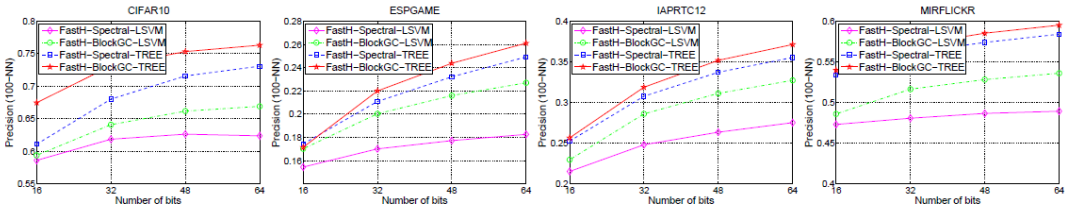

Fig. 4 – Comparison of combinations of hash functions and binary inference methods. Decision tree hash functions perform much better than linear SVM. The proposed Block-GC performs much better than the spectral method.

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

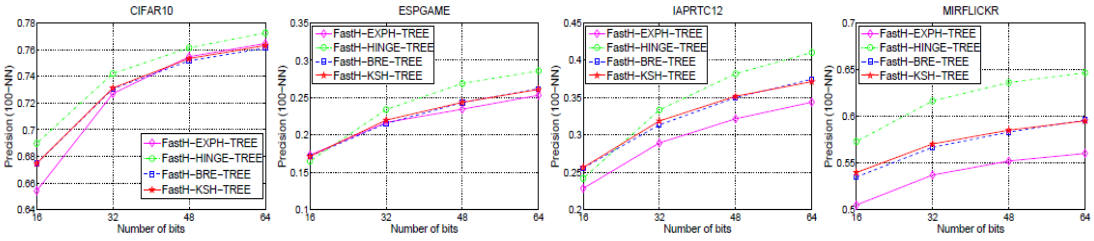

Fig. 5 – Comparison of using different loss functions with decision tree hash functions. Using the Hinge loss (FastH-Hinge) achieves the best result.

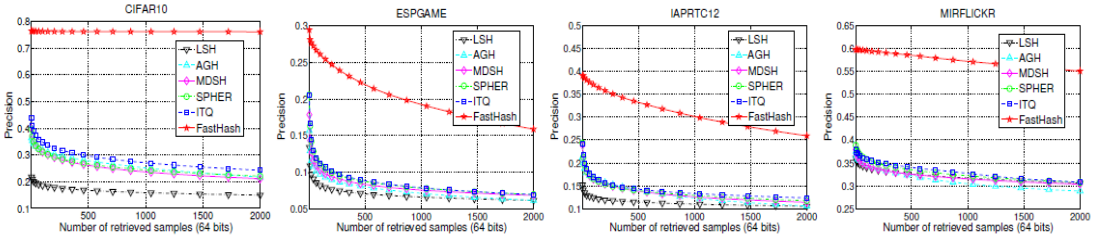

Fig. 6 – Comparison with a few unsupervised hashing methods. Unsupervised methods perform poorly for preserving label based similarity. Our FastHash performs significantly better.

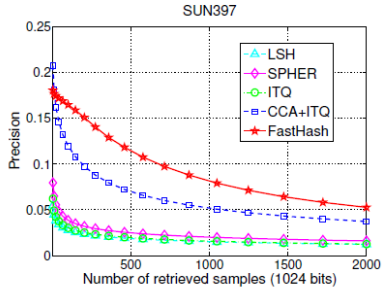

Fig. 7 – The top-2000 precision curve on large dataset SUN397 (1024 bits). Our FastHash performs the best.

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**

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**

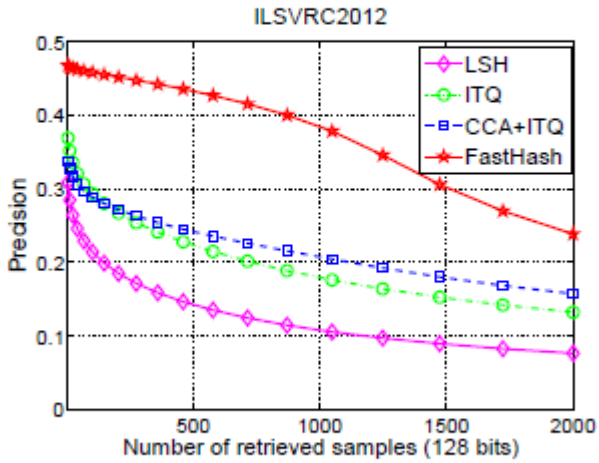

Fig. 8 – The top-2000 precision curve on large dataset ILSVRC2012 (128 bits). Our FastHash outperforms others.

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	

Conclusion：

本文已经表明，可以将各种损失函数和散列函数放置在统一的监督哈希学习框架中。 利用所提出的二进制推理算法 Block GraphCut 和学习决策树哈希函数，可以有效地对大规模和高维数据进行训练，并达到较高的测试精度，这对于大规模图像检索等许多应用具有实际意义。

Reference：

[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.