OPTIMIZING AFFINITY-BASED BINARY HASHING USING AUXILIARY COORDINATES

Ramin Raziperchikolaei and **Miguel Á. Carreira-Perpiñán.** EECS, UC Merced, USA

ICML@NYC
International Conference on Machine Learning

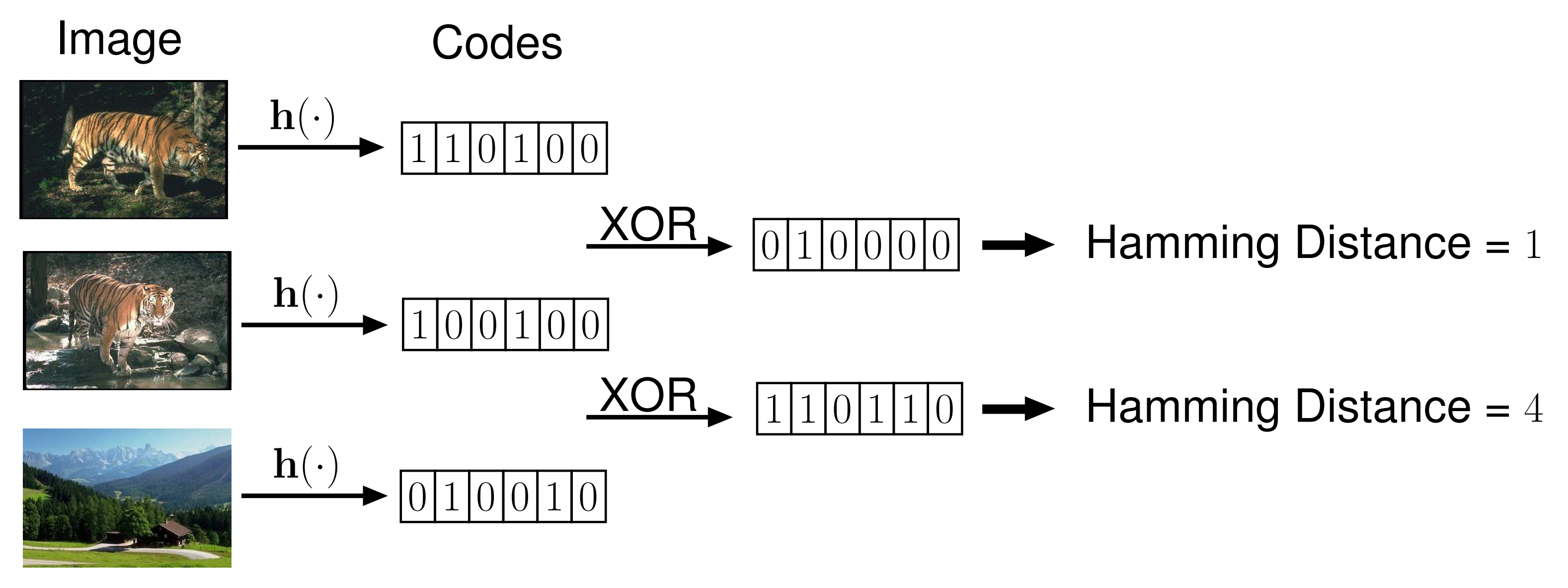
1 Binary hash functions for fast image retrieval

In K nearest neighbors problem, there are N training points in D-dimensional space (usually $D > 100$) $\mathbf{x}_i \in \mathbb{R}^D, i = 1, \dots, N$. The goal is to find the K nearest neighbors of a query point $\mathbf{x}_q \in \mathbb{R}^D$.

- Exact search in the original space is $\mathcal{O}(ND)$ in time and space.

A binary hash function $\mathbf{h}$ takes as input a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$ and maps it to an b-bit vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{0,1\}^b$. The search is done in this low-dimensional, binary space.

- The main goal is preserving the neighborhood, i.e., assign (dis)similar codes to (dis)similar patterns.



Finding K nearest neighbors in Hamming space is more efficient:

- Time and space complexities would be $\mathcal{O}(Nb)$ instead of $\mathcal{O}(ND)$.
- Hamming Distance can be computed efficiently and fast using hardware operations.

Suppose that $N = 10^9$, $D = 500$ and $b = 64$

Search in	Space	Time
original space	2 TB	1 hour
Hamming space	8 GB	10 seconds

2 Affinity-based objective functions

Hashing papers first define a supervised objective that has a small value for good hash functions. They then try to minimize this objective function.
We focus here on *affinity-based loss functions*, which directly try to preserve the original similarities in the binary space.

$$\min \mathcal{L}(\mathbf{h}) = \textstyle\sum_{n,m=1}^{N} L(\mathbf{h}(\mathbf{x}_n), \mathbf{h}(\mathbf{x}_m);\ y_{nm})$$

where $\mathbf{x}_i \in \mathbb{R}^D$ is the i-th input data, $\mathbf{h}$ is the parameters of the hash function, $L(\cdot)$ is a loss function that compares the codes for two images with the ground-truth value y_{nm} that measures the affinity in the original space between the two images $\mathbf{x}_n$ and $\mathbf{x}_m$.
Examples of the loss function $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$:

KSH: $(\mathbf{z}_n^T\mathbf{z}_m - b y_{nm})^2$ BRE: $(\frac{1}{b}\left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2 - y_{nm})^2$ eSPLH: $\exp(-\frac{1}{b} y_{nm}\mathbf{z}_n^T\mathbf{z}_n)$

If the hash function $\mathbf{h}$ was a continuous function, one could compute derivatives over the parameters of $\mathbf{h}$ and then apply a nonlinear optimization method.

In binary hashing, optimization is much more difficult:

- the hash function must output binary values, hence the problem is not just generally nonconvex, but also nonsmooth.
- While the gradients of the objective function do exist wrt $\mathbf{W}$, they are zero nearly everywhere.

3 Optimization using two-step approach

Most hashing papers follow a simple but suboptimal approach:

- Define the objective function directly on the b-dimensional codes of each image (instead of the hash functions) and optimizes it. This is an NP-complete problem with Nb binary variables. This can be solved approximately.
- Learn the hash function given the codes, by training several classifiers.

The main issue of this approach is that it does not consider the relation between the binary codes and the hash function in optimizing the codes (the first step).

4 Optimization using auxiliary coordinates

We show that all elements of the problem (binary codes and hash function) can be incorporated in a single algorithm that optimizes jointly over them.

We use the method of auxiliary coordinates (MAC), a generic approach to optimize nested functions. First, we introduce auxiliary coordinates $\mathbf{z}_n \in \{-1,+1\}^b$ as the output of $\mathbf{h}(\mathbf{x}_n)$ and convert the problem for $\mathcal{L}(\mathbf{h})$ into an equivalent constrained problem:

$$\mathcal{L}_c(\mathbf{h}, \mathbf{Z}) = \textstyle\sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \text{ s.t. } \mathbf{z}_1 = \mathbf{h}(\mathbf{x}_1), \cdots, \mathbf{z}_N = \mathbf{h}(\mathbf{x}_N)$$

Now we apply the quadratic-penalty method:

$$\mathcal{L}_P(\mathbf{h}, \mathbf{Z}; \mu) = \textstyle\sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) + \mu \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\right\|^2$$

where $\mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1,+1\}^b$. We start with a small μ and increase it slowly. To optimize $\mathcal{L}_P(\mathbf{h}, \mathbf{Z}; \mu)$ we apply alternating optimization:

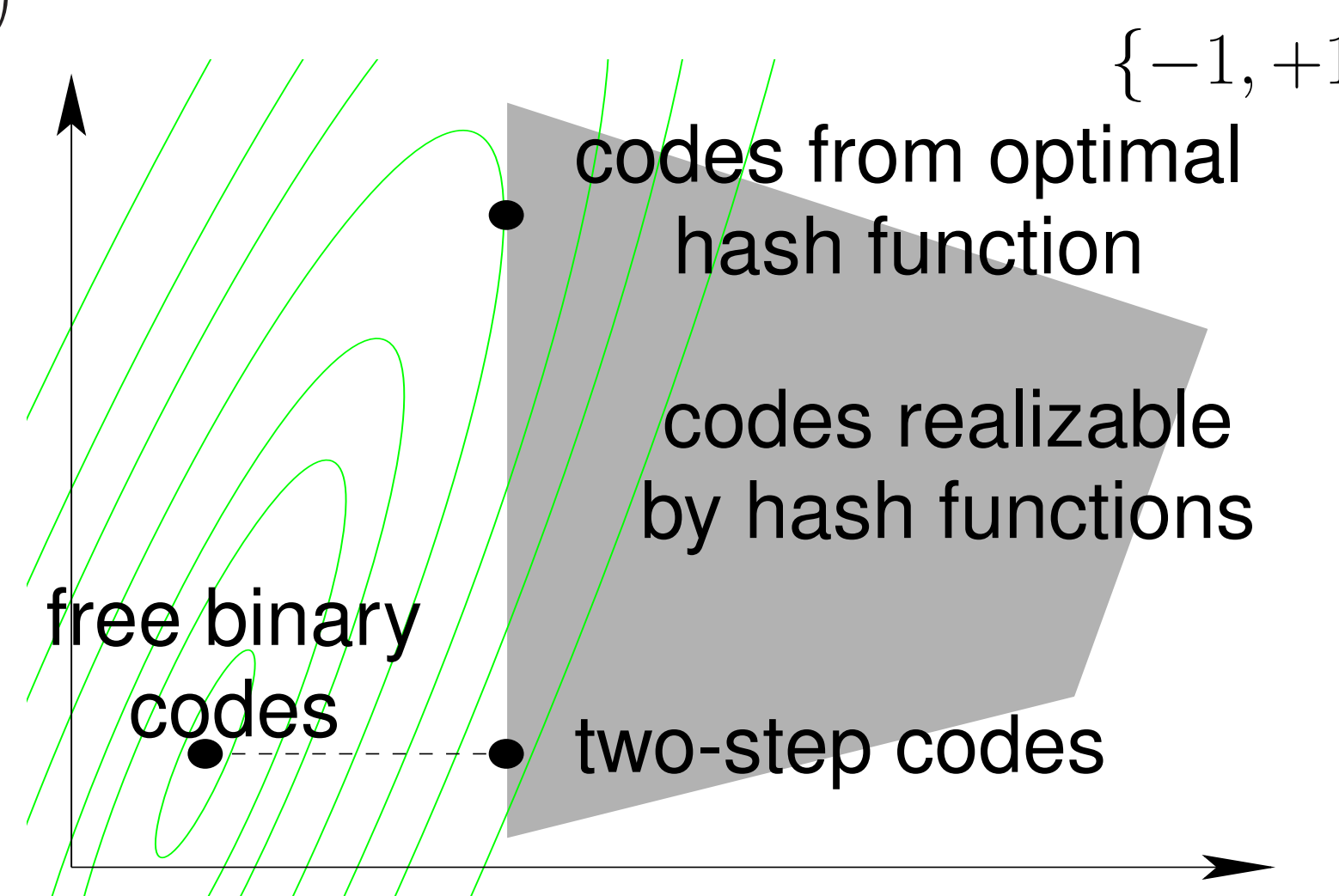
- Optimization over $\mathbf{Z}$ given $\mathbf{h}$. This is an NP-complete problem over bN binary variables and can be seen as a *regularized binary embedding*.
- Optimization over $\mathbf{h}$ given $\mathbf{Z}$: $\min_{\mathbf{h}} \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\right\|^2$. This is equivalent to training b binary classifiers with data $(\mathbf{X}, \mathbf{Z})$.

The $\mathbf{Z}$-step is still complex. Some recent works have proposed practical approaches for this: (1) apply alternating optimization over the i-th bit of all points given the remaining bits are fixed. This gives a binary quadratic problem. (2) Solve this quadratic problem approximately (using methods like GraphCut).

The two-step approach (TSH) corresponds to optimizing $\mathcal{L}_P$ for $\mu \to 0^+$. In practice, we start from a very small value of μ (hence, initialize MAC from the result of TSH).

Advantages of optimizing affinity-based objectives using MAC:

- It optimizes jointly over the binary codes and the hash function in alternation resulting in a better local optimum of the affinity-based loss.
- It performs better than previous, two-step approaches in both optimization and information retrieval measures like precision and recall.
- Our framework makes it easy to design an optimization algorithm for a new choice of loss function or hash function.


$\{-1,+1\}^{b\times N}$
codes from optimal hash function
codes realizable by hash functions
free binary codes
two-step codes

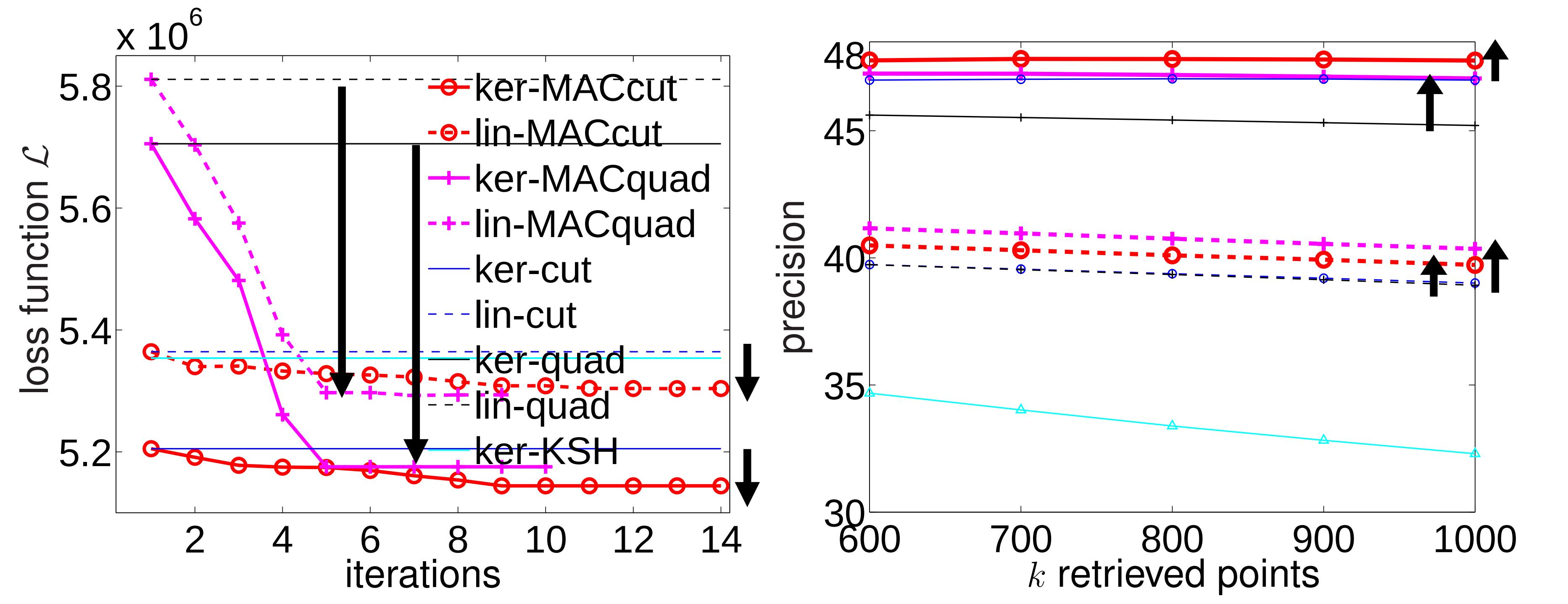
This figure shows the space of all possible binary codes and the faesible set for linear hash functions. The contours corresponds to $\mathcal{L}_c$ defined only on codes. The two-step method projects the free codes into the feasible set.
MAC optimizes the codes and functions jointly to find a better local optima.

5 Theoretical results

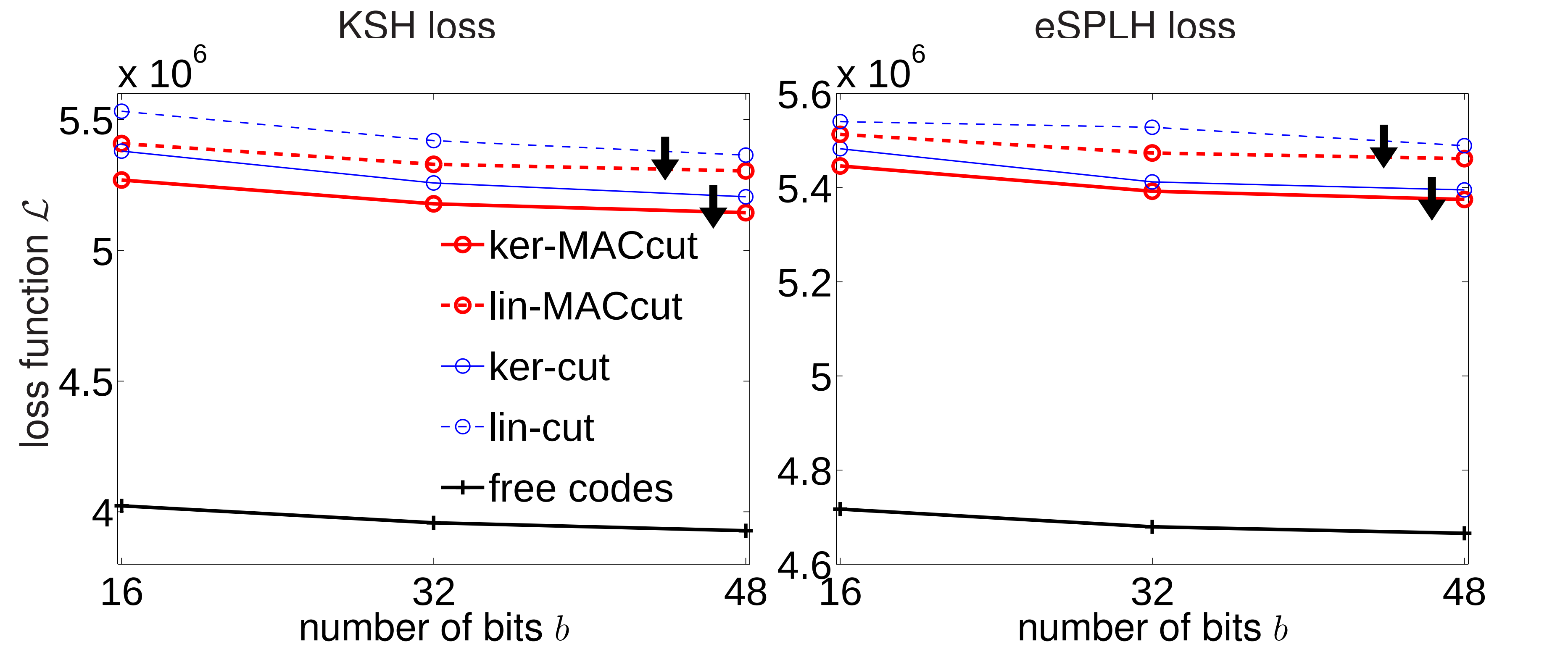
We can prove the following under the assumption that the $\mathbf{Z}$ and $\mathbf{h}$ steps are exact:

1. The MAC algorithm stops after a finite number of iterations, when $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ in the $\mathbf{Z}$ step, since then the constraints are satisfied and no more changes will occur to $\mathbf{Z}$ or $\mathbf{h}$.
2. The path over the continuous penalty parameter $\mu \in [0,\infty)$ is in fact discrete: the minimizer $(\mathbf{h}, \mathbf{Z})$ of $\mathcal{L}_P$ for $\mu \in [0, \mu_1]$ is identical to the minimizer for $\mu = 0$, and the minimizer for $\mu \in [\mu_2, \infty)$ is identical to the minimizer for $\mu \to \infty$, where $0 < \mu_1 < \mu_2 < \infty$. Hence, it suffices to take an initial μ no smaller than μ_1 and keep increasing it until the algorithm stops. In practice, we increase μ more aggressively to reduce the runtime.

6 Experiments



We compare our proposed method MAC with the two step methods that use quadratic surrogate and GraphCut methods in the optimization over codes. We denote these methods as cut and quad. The MAC version of them is called MACcut and MACquad.
We use a subset of 10 000 points of CIFAR dataset and we use two types of hash functions (linear and kernel SVMs). MAC finds hash functions with significantly lower objective function values than the two-step approaches. Reducing the loss nearly always translates into better precision and recall.



MAC learns better hash functions. We achieve free codes by minimizing $\mathcal{L}_c$ over the binary codes $\mathbf{Z}$ without any constraint. Free codes are the starting point of both cut and MACcut. Free codes always achieve lower error than the cut and MACcut.
MAC achieves lower error than the cut using both linear and kernel hash function and using different loss functions. MAC gradually optimizes both the codes and the hash function so they eventually match, and finds a better hash function for the original problem.

Work supported by NSF award IIS–1423515