Optimizing Affinity-Based Binary Hashing Using Auxiliary Coordinates

Ramin Raziperchikolaei
EECS, University of California, Merced
rraziperchikolaei@ucmerced.edu

Miguel Á. Carreira-Perpiñán
EECS, University of California, Merced
mcarreira-perpinan@ucmerced.edu

Introduction. In image retrieval, a user is interested in finding similar images to a query image. Finding the exact nearest neighbors in a dataset of N images, each a vector of dimension D, takes $\mathcal{O}(ND)$ in both time and space, which is slow. In practice, this is approximated, and a successful way to do this is *binary hashing* [2]. Here, given a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$, the hash function $\mathbf{h}$ maps it to a b-bit vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{-1, +1\}^b$, and the search is then done in the binary space. This now costs $\mathcal{O}(Nb)$ in time and space, which is orders of magnitude faster because typically $b < D$ and, crucially, (1) operations with binary vectors are very fast because of hardware support, and (2) the entire dataset can fit in (fast) memory rather than slow memory or disk.

The main goal of binary hashing is to preserve the neigborhood: similar (dissimilar) points in the original space have to be mapped into the similar (dissimilar) binary codes. To achieve this, the general approach consists of defining a supervised objective over the parameters of the hash function and then minimizing it. Optimization of this objective is very difficult because the hash function must output binary values, hence the problem is not just generally nonconvex, but also nonsmooth. Most hashing papers use a two-step approach to minimize the objective. First, one defines the objective function directly on the b-dimensional codes of each image (rather than on the hash function parameters). Then, one learns a hash function given the codes. This is a suboptimal approach because it ignores the hash functions while it learns the binary codes. Here, 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. This leads to learning better hash functions and achieving lower error.

Objective function. We focus here on *affinity-based loss functions*, which have been used in most hashing papers. The objective function has the following form

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

where $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$ is the high-dimensional dataset of feature vectors, $\min_{\mathbf{h}}$ means minimizing over the parameters of the hash function $\mathbf{h}$ (e.g. over the weights of a linear SVM), and $L(\cdot)$ 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 $\mathbf{x}_m$. An example of the loss functions is $L(\mathbf{z}_n, \mathbf{z}_m, y_{nm}) = (\mathbf{z}_n^T \mathbf{z}_m - b y_{nm})^2$ where y_{nm} is 1 if $\mathbf{x}_n$, $\mathbf{x}_m$ are similar and -1 if they are dissimilar. We propose a general approach that works with different kinds of the loss functions.

Our approach: Learning codes and hash functions using auxiliary coordinates. The optimization of the loss $\mathcal{L}(\mathbf{h})$ in eq. (1) is difficult because of the thresholded hash function, which appears as the argument of the loss function L. We use the recently proposed *method of auxiliary coordinates (MAC)* [1], which is a meta-algorithm to construct optimization algorithms for nested functions. This proceeds in 3 stages. First, we introduce new variables (the "auxiliary coordinates") as equality constraints into the problem, with the goal of unnesting the function. We can achieve this by introducing one binary vector $\mathbf{z}_n \in \{-1, +1\}$ for each point. This transforms the original, unconstrained problem into the following, constrained problem:

$$\min_{\mathbf{h},\mathbf{Z}} \textstyle\sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \ \text{ s.t. } \ \mathbf{z}_1 = \mathbf{h}(\mathbf{x}_1), \cdots, \mathbf{z}_N = \mathbf{h}(\mathbf{x}_N) \tag{2}$$

which is seen to be equivalent to (1) by eliminating $\mathbf{Z}$. We recognize as the objective function the "embedding" form of the loss function, except that the "free" parameters $\mathbf{z}_n$ are in fact constrained to be the deterministic outputs of the hash function $\mathbf{h}$.

Second, we solve the constrained problem using the quadratic-penalty (or augmented Lagrangian) method. We solve the following minimization problem (unconstrained again, but dependent on μ) while progressively increasing μ, so the constraints are eventually satisfied:

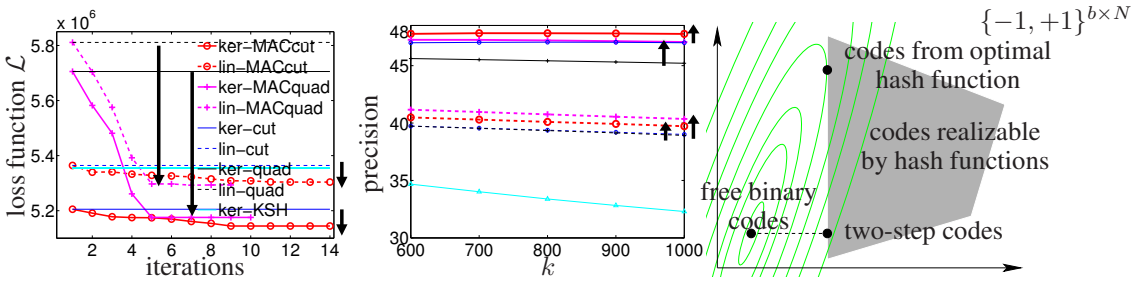

Figure 1: *Panels 1–2*: Loss function $\mathcal{L}$ and precision for k retrieved points on CIFAR dataset, using $b = 48$ bits. *Panel 3*: Illustration of free codes, two-step codes and optimal codes

$\min \mathcal{L}_P(\mathbf{h}, \mathbf{Z}; \mu) = \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) + \mu \sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$ s.t. $\mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1,+1\}^b$.

Third, we apply alternating optimization over the binary codes $\mathbf{Z}$ and the hash function parameters $\mathbf{h}$. This results in iterating the following two steps:

- Optimize the binary codes $\mathbf{z}_1, \dots, \mathbf{z}_N$ given $\mathbf{h}$ (hence, given the output binary codes $\mathbf{h}(\mathbf{x}_1), \dots, \mathbf{h}(\mathbf{x}_N)$ for each of the N images). This can be seen as a *regularized binary embedding*, because the projections $\mathbf{Z}$ are encouraged to be close to the hash function outputs $\mathbf{h}(\mathbf{X})$. Here, we try two different approaches [3, 4] with some modifications.
- Optimize the hash function $\mathbf{h}$ given binary codes $\mathbf{Z}$. This reduces to training b binary classifiers using $\mathbf{X}$ as inputs and $\mathbf{Z}$ as targets.

This is similar to the two-step (TSH) approach of Lin et al. [3], except that the latter learns the codes $\mathbf{Z}$ in isolation, rather than given the current hash function, so iterating the two-step approach would change nothing, and it does not optimize the loss $\mathcal{L}$. More precisely, 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), and increase μ slowly while optimizing $\mathcal{L}_P$, until the equality constraints are satisfied, i.e., $\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$ for $n = 1, \dots, N$.

Why does MAC learn better hash functions? Consider $E(\mathbf{Z})$ as the objective in eq (2) without the "$\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$" constraints: the loss over the codes without them being the output of a particular hash function. In both the two-step and MAC approaches, the starting point are the "free" binary codes obtained by minimizing $E(\mathbf{Z})$. Fig. 1 (right) shows why MAC learns better hash functions conceptually. It shows the space of all possible binary codes, the contours of $E(\mathbf{Z})$ (green) and the set of codes that can be produced by (say) linear hash functions $\mathbf{h}$ (gray), which is the feasible set $\{\mathbf{Z} \in \{-1,+1\}^{b\times N}$: $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ for linear $\mathbf{h}\}$. The two-step codes "project" the free codes onto the feasible set, but these are not the codes for the optimal hash function $\mathbf{h}$. MAC gradually optimizes both the codes and the hash function so they eventually match, and finds a better hash function.

The MAC algorithm finds better optima Our goal is not to introduce a new loss or hash function, but to describe a generic framework to construct algorithms that optimize a given combination thereof. We optimize KSH loss function and two hash functions (linear and kernel SVM) on CIFAR dataset. We compare our iterative optimization algorithm (*MACquad* and *MACcut*) with the two-step methdos (*quad* [3], *cut* [4]). Fig. 1 shows the value of loss function over iterations of the MAC algorithm (*quad* and *cut* do not iterate), as well as precision. It is clear that *MACcut* (red lines) and *MACquad* (magenta lines) reduce the loss function more than *cut* (blue lines) and *quad* (black lines), respectively. Hence, applying MAC is always beneficial. Reducing the loss nearly always translates into better precision and recall. The gain of *MACcut*/*MACquad* over *cut*/*quad* is significant, often comparable to the gain obtained by changing from the linear to the kernel hash function.

References

[1] M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pages 10–19, Apr. 22–25 2014.

[2] K. Grauman and R. Fergus. Learning binary hash codes for large-scale image search. *Machine Learning for Computer Vision*, pages 49–87. Springer-Verlag, 2013.

[3] G. Lin, C. Shen, D. Suter, and A. van den Hengel. A general two-step approach to learning-based hashing. In *Proc. 14th Int. Conf. Computer Vision (ICCV'13)*, pages 2552–2559, Sydney, Australia, Dec. 1–8 2013.

[4] G. Lin, C. Shen, Q. Shi, A. van den Hengel, and D. Suter. Fast supervised hashing with decision trees for high-dimensional data. In *CVPR*, pages 1971–1978, Columbus, OH, June 23–28 2014.