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

Abstract

In supervised binary hashing, one wants to learn a function that maps a high-dimensional feature vector to a vector of binary codes, for application to fast image retrieval. This typically results in a difficult optimization problem, nonconvex and nonsmooth, because of the discrete variables involved. Much work has simply relaxed the problem during training, solving a continuous optimization, and truncating the codes a posteriori. This gives reasonable results but is quite suboptimal. Recent work has tried to optimize the objective directly over the binary codes and achieved better results, but the hash function was still learned a posteriori, which remains suboptimal. We propose a general framework for learning hash functions using affinity-based loss functions that uses auxiliary coordinates. This closes the loop and optimizes jointly over the hash functions and the binary codes so that they gradually match each other. The resulting algorithm can be seen as an iterated version of the procedure of optimizing first over the codes and then learning the hash function. Compared to this, our optimization is guaranteed to obtain better hash functions while being not much slower, as demonstrated experimentally in various supervised datasets. In addition, our framework facilitates the design of optimization algorithms for arbitrary types of loss and hash functions.

Information retrieval arises in several applications, most obviously web search. For example, in image retrieval, a user is interested in finding similar images to a query image. Computationally, this essentially involves defining a high-dimensional feature space where each relevant image is represented by a vector, and then finding the closest points (nearest neighbors) to the vector for the query image, according to a suitable distance. For example, one can use features such as SIFT or GIST [23] and the Euclidean distance for this purpose. Finding nearest neighbors in a dataset of N images (where N can be millions), each a vector of dimension D (typically in the hundreds) is slow, since exact algorithms run essentially in time $\mathcal{O}(ND)$ and space $\mathcal{O}(ND)$ (to store the image dataset). In practice, this is approximated, and a successful way to do this is *binary hashing* [12]. 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 nearest neighbor search is then done in the binary space. This now costs $\mathcal{O}(Nb)$ time and space, which is orders of magnitude faster because typically $b < D$ and, crucially, (1) operations with binary vectors (such as computing Hamming distances) are very fast because of hardware support, and (2) the entire dataset can fit in (fast) memory rather than slow memory or disk.

The disadvantage is that the results are inexact, since the neighbors in the binary space will not be identical to the neighbors in the original space. However, the approximation error can be controlled by using sufficiently many bits and by *learning a good hash function*. This has been the topic of much work in recent years. The general approach consists of defining a supervised objective that has a small value for good hash functions and minimizing it. Ideally, such an objective function should be minimal when the neighbors of any given image are the same in both original and binary spaces. Practically, in information retrieval, this is often evaluated using precision and recall. However, this

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

ideal objective cannot be easily optimized over hash functions, and one uses approximate objectives instead. Many such objectives have been proposed in the literature. We focus here on *affinity-based loss functions*, which directly try to preserve the original similarities in the binary space. Specifically, we consider objective functions of the 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 (often through their Hamming distance $\|\mathbf{h}(\mathbf{x}_n) - \mathbf{h}(\mathbf{x}_m)\|$) with the ground-truth value y_{nm} that measures the affinity in the original space between the two images $\mathbf{x}_n$ and $\mathbf{x}_m$ (distance, similarity or other measure of neighborhood; [12]). The sum is often restricted to a subset of image pairs (n, m) (for example, within the k nearest neighbors of each other in the original space), to keep the runtime low. Examples of these objective functions (described below) include models developed for dimension reduction, be they spectral such as Laplacian Eigenmaps [2] and Locally Linear Embedding [24], or nonlinear such as the Elastic Embedding [4] or t-SNE [26]; as well as objective functions designed specifically for binary hashing, such as Supervised Hashing with Kernels (KSH) [19], Binary Reconstructive Embeddings (BRE) [14] or sequential Projection Learning Hashing (SPLH) [29].

If the hash function $\mathbf{h}$ was a continuous function of its input $\mathbf{x}$ and its parameters, one could simply apply the chain rule to compute derivatives over the parameters of $\mathbf{h}$ of the objective function (1) and then apply a nonlinear optimization method such as gradient descent. This would be guaranteed to converge to an optimum under mild conditions (for example, Wolfe conditions on the line search), which would be global if the objective is convex and local otherwise [21]. Hence, optimally learning the function $\mathbf{h}$ would be in principle doable (up to local optima), although it would still be slow because the objective can be quite nonlinear and involve many terms.

In binary hashing, the optimization is much more difficult, because in addition to the previous issues, the hash function must output binary values, hence the problem is not just generally nonconvex, but also nonsmooth. In view of this, much work has sidestepped the issue and settled on a simple but suboptimal solution. First, one defines the objective function (1) directly on the b-dimensional codes of each image (rather than on the hash function parameters) and optimizes it assuming continuous codes (in $\mathbb{R}^b$). Then, one binarizes the codes for each image. Finally, one learns a hash function given the codes. Optimizing the affinity-based loss function (1) can be done using spectral methods or nonlinear optimization as described above. Binarizing the codes has been done in different ways, from simply rounding them to $\{-1, +1\}$ using zero as threshold [18, 19, 30, 33], to optimally finding a threshold [18], to rotating the continuous codes so that thresholding introduces less error [11, 32]. Finally, learning the hash function for each of the b output bits can be considered as a binary classification problem, where the resulting classifiers collectively give the desired hash function, and can be solved using various machine learning techniques. Several works (e.g. [16, 17, 33]) have used this approach, which does produce reasonable hash functions (in terms of retrieval measures such as precision/recall).

In order to do better, one needs to take into account during the optimization (rather than after the optimization) the fact that the codes are constrained to be binary. This implies attempting directly the discrete optimization of the affinity-based loss function over binary codes. This is a daunting task, since this is usually an NP-complete problem with Nb binary variables altogether, and practical applications could make this number as large as millions or beyond. Recent works have applied alternating optimization (with various refinements) to this, where one optimizes over a usually small subset of binary variables given fixed values for the remaining ones [16, 17], and this did result in very competitive precision/recall compared with the state-of-the-art. This is still slow and future work will likely improve it, but as of now it provides an option to learn better binary codes.

Of the three-step suboptimal approach mentioned (learn continuous codes, binarize them, learn hash function), these works manage to join the first two steps and hence learn binary codes [16, 17]. Then, one learns the hash function given these binary codes. Can we do better? Indeed, in this paper *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.* Hence, by initializing it from binary codes from the previous approach, this algorithm is guaranteed to achieve a lower error and learn better hash functions. Our framework can be seen as an iterated version of the two-step approach: learn binary codes *given the current hash function*, learn hash functions given codes, iterate (note the emphasis).

The key to achieve this in a principled way is to use a recently proposed *method of auxiliary coordinates (MAC)* for optimizing "nested" systems, i.e., consisting of the composition of two or more functions or processing stages. MAC introduces new variables and constraints that cause decoupling between the stages, resulting in the mentioned alternation between learning the hash function and learning the binary codes. Section 1 reviews affinity-based loss functions, section 2 describes our MAC-based proposed framework, section 3 evaluates it in several supervised datasets, using linear and nonlinear hash functions, and section 4 discusses implications of this work.

Related work Although one can construct hash functions without training data [1, 15], we focus on methods that learn the hash function given a training set, since they perform better, and our emphasis is in optimization. The learning can be unsupervised [5, 11], which attempts to preserve distances in the original space, or supervised, which in addition attempts to preserve label similarity. Many objective functions have been proposed to achieve this and we focus on affinity-based ones. These create an affinity matrix for a subset of training points based on their distances (unsupervised) or labels (supervised) and combine it with a loss function [14, 16, 17, 19, 22]. Some methods optimize this directly over the hash function. For example, Binary Reconstructive Embeddings [14] use alternating optimization over the weights of the hash functions. Supervised Hashing with Kernels [19] learns hash functions sequentially by considering the difference between the inner product of the codes and the corresponding element of the affinity matrix. Although many approaches exist, a common theme is to apply a greedy approach where one first finds codes using an affinity-based loss function, and then fits the hash functions to them (usually by training a classifier). The codes can be found by relaxing the problem and binarizing its solution [18, 30, 33], or by approximately solving for the binary codes using some form of alternating optimization (possibly combined with GraphCut), as in two-step hashing [10, 16, 17], or by using relaxation in other ways [19, 22].

1 Nonlinear embedding and affinity-based loss functions for binary hashing

The dimensionality reduction literature has developed a number of objectives of the form (1) (often called "embeddings") where the low-dimensional projection $\mathbf{z}_n \in \mathbb{R}^b$ of each high-dimensional data point $\mathbf{x}_n \in \mathbb{R}^D$ is a free, real-valued parameter. The neighborhood information is encoded in the y_{nm} values (using labels in supervised problems, or distance-based affinities in unsupervised problems). An example is the elastic embedding [4], where $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form:

$$y^+_{nm} \left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2 + \lambda y^-_{nm} \exp\left(-\left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2\right),\ \lambda > 0 \tag{2}$$

where the first term tries to project true neighbors (having $y^+_{nm} > 0$) close together, while the second repels all non-neighbors' projections (having $y^-_{nm} > 0$) from each other. Laplacian Eigenmaps [2] and Locally Linear Embedding [24] result from replacing the second term above with a constraint that fixes the scale of $\mathbf{Z}$, which results in an eigenproblem rather than a nonlinear optimization, but also produces more distorted embeddings. Other objectives exist, such as t-SNE [26], that do not separate into functions of pairs of points. Optimizing nonlinear embeddings is quite challenging, but much progress has been done recently [4, 6, 25, 27, 28, 31]. Although these models were developed to produce continuous projections, they have been successfully used for binary hashing too by truncating their codes [30, 33] or using the two-step approach of [16, 17].

Other loss functions have been developed specifically for hashing, where now $\mathbf{z}_n$ is a b-bit vector (where binary values are in $\{-1, +1\}$). For example (see a longer list in [16]), for Supervised Hashing with Kernels (KSH) $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form

$$(\mathbf{z}_n^T \mathbf{z}_m - b y_{nm})^2 \tag{3}$$

where y_{nm} is 1 if $\mathbf{x}_n$, $\mathbf{x}_m$ are similar and -1 if they are dissimilar. Binary Reconstructive Embeddings [14] uses $(\frac{1}{b}\left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2 - y_{nm})^2$ where $y_{nm} = \frac{1}{2}\left\|\mathbf{x}_n - \mathbf{x}_m\right\|^2$. The exponential variant of SPLH [29] proposed by Lin et al. [16] (which we call eSPLH) uses $\exp(-\frac{1}{b} y_{nm} \mathbf{z}_n^T \mathbf{z}_n)$. Our approach can be applied to any of these loss functions, though we will mostly focus on the KSH loss for simplicity. When the variables $\mathbf{Z}$ are binary, we will call these optimization problems *binary embeddings*, in analogy to the more traditional continuous embeddings for dimension reduction.

2 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)* [7, 8], 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 $\mathbf{z}_n \in \{-1,+1\}^b$ for each point. This transforms the original, unconstrained problem into the following equivalent, constrained problem:

$$\min_{\mathbf{h},\mathbf{Z}} \textstyle\sum_{n=1}^N L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \ \text{ s.t. } \ \mathbf{z}_1 = \mathbf{h}(\mathbf{x}_1), \cdots, \mathbf{z}_N = \mathbf{h}(\mathbf{x}_N). \tag{4}$$

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 a penalty method, such as the quadratic-penalty or augmented Lagrangian [21]. We discuss here the former for simplicity. We solve the following minimization problem (unconstrained again, but dependent on μ) while progressively increasing μ, so the constraints are eventually satisfied:

$$\min \mathcal{L}_P(\mathbf{h}, \mathbf{Z}; \mu) = \sum_{n,m=1}^N L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \ + \mu \sum_{n=1}^N \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 \quad \text{s.t.} \quad \begin{matrix} \mathbf{z}_1, \dots, \mathbf{z}_N \in \\ \{-1,+1\}^b. \end{matrix} \tag{5}$$

$\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$ is proportional to the Hamming distance between the binary vectors $\mathbf{z}_n$ and $\mathbf{h}(\mathbf{x}_n)$.

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

Z step 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 [16, 17] with some modifications.

h step Optimize the hash function $\mathbf{h}$ given the binary codes $\mathbf{Z}$. This simply means training b binary classifiers using $\mathbf{X}$ as inputs and $\mathbf{Z}$ as labels.

This is very similar to the two-step (TSH) approach of Lin et al. [16], 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 $n = 1, \dots, N$. The supplementary material gives the overall MAC algorithm to learn a hash function by optimizing an affinity-based loss function.

Theoretical results We can prove the following under the assumption that the $\mathbf{Z}$ and $\mathbf{h}$ steps are exact (suppl. mat.). 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. Besides, the interval $[\mu_1, \mu_2]$ is itself partitioned in a finite set of intervals so that the minimizer changes only at interval boundaries. Hence, theoretically the algorithm needs only run for a finite set of μ values (although this set can still be very big). In practice, we increase μ more aggressively to reduce the runtime.

This is very different from the quadratic-penalty methods in continuous optimization [21], which was the setting considered in the original MAC papers [7, 8]. There, the minimizer varies continuously with μ, which must be driven to infinity to converge to a stationary point, and in so doing it gives rise to ill-conditioning and slow convergence.

2.1 h step: optimization over the parameters of the hash function, given the binary codes

Given the binary codes $\mathbf{z}_1, \dots, \mathbf{z}_N$, since $\mathbf{h}$ does not appear in the first term of $\mathcal{L}_P$, this simply involves finding a hash function $\mathbf{h}$ that minimizes

$$\min_{\mathbf{h}} \textstyle\sum_{n=1}^N \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 = \sum_{i=1}^b \min_{h_i} \sum_{n=1}^N (z_{ni} - h_i(\mathbf{x}_n))^2$$

where $z_{ni} \in \{-1,+1\}$ is the ith bit of the binary vector $\mathbf{z}_n$. Hence, we can find b one-bit hash functions in parallel and concatenate them into the b-bit hash function. Each of these is a binary classification problem using the number of misclassified patterns as loss. This allows us to use a regular classifier for $\mathbf{h}$, and even to use a simpler surrogate loss (such as the hinge loss), since this will also enforce the constraints eventually (as μ increases). For example, we can fit an SVM by optimizing the margin plus the slack and using a high penalty for misclassified patterns. We discuss other classifiers in the experiments.

2.2 Z step: optimization over the binary codes, given the hash function

Although the MAC technique has significantly simplified the original problem, the step over $\mathbf{Z}$ is still complex. This involves finding the binary codes given the hash function $\mathbf{h}$, and it is an NP-complete problem in Nb binary variables. Fortunately, some recent works have proposed practical approaches for this problem based on alternating optimization: a quadratic surrogate method [16], and a GraphCut method [17]. In both methods, the starting point is to apply alternating optimization over the ith bit of all points given the remaining bits are fixed for all points (for $i = 1, \dots, b$), and to solve the optimization over the ith bit approximately. This would correspond to the first step in the two-step hashing of Lin et al. [16]. These methods, in their original form, can be applied to the loss function over binary codes, i.e., the first term in $\mathcal{L}_P$. Here, we explain briefly our modification to these methods to make them work with our $\mathbf{Z}$ step objective (the regularized loss function over codes, i.e., the complete $\mathcal{L}_P$). The full explanation can be found in the supplementary material.

Solution using a quadratic surrogate method [16] This is based on the fact that any loss function that depends on the Hamming distance of two binary variables can be equivalently written as a quadratic function of those two binary variables. We can then write the first term in $\mathcal{L}_P$ as a binary quadratic problem using a certain matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ (computed using the fixed bits), and the second term (on μ) is also quadratic. The optimization for the ith bit can then be equivalently written as

$$\min_{\mathbf{z}_{(i)}} \mathbf{z}_{(i)}^T \mathbf{A} \mathbf{z}_{(i)} + \mu \left\| \mathbf{z}_{(i)} - \mathbf{h}_i(\mathbf{X}) \right\|^2 \quad \text{s.t.} \quad \mathbf{z}_{(i)} \in \{-1, +1\}^N \tag{6}$$

where $\mathbf{h}_i(\mathbf{X}) = (h_i(\mathbf{x}_1), \dots, h_i(\mathbf{x}_N))^T$ and $\mathbf{z}_{(i)}$ are vectors of length N (one bit per data point). This is still an NP-complete problem (except in special cases), and we approximate it by relaxing it to a continuous quadratic program (QP) over $\mathbf{z}_{(i)} \in [-1, 1]^N$, minimizing it using L-BFGS-B [34] and binarizing its solution.

Solution using a GraphCut algorithm [17] To optimize $\mathcal{L}_P$ over the ith bit of each image (given all the other bits are fixed), we have to minimize the NP-complete problem of eq. (6) over N bits. We can apply the GraphCut algorithm [3], as proposed by the FastHash algorithm of Lin et al. [17]. This proceeds as follows. First, we assign all the data points to different, possibly overlapping groups (blocks). Then, we minimize the objective function over the binary codes of the same block, while all the other binary codes are fixed, then proceed with the next block, etc. (that is, we do alternating optimization of the bits over the blocks). Specifically, to optimize over the bits in block $\mathcal{B}$, ignoring the constants, we can rewrite equation (6) in the standard form for the GraphCut algorithm as:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \sum_{n \in \mathcal{B}} \sum_{m \in \mathcal{B}} v_{nm} z_{ni} z_{mi} + \sum_{n \in \mathcal{B}} u_{nm} z_{ni}$$

where $v_{nm} = a_{nm}$, $u_{nm} = 2 \sum_{m \notin \mathcal{B}} a_{nm} z_{mi} - \mu h_i(\mathbf{x}_n)$. To minimize the objective function using the GraphCut algorithm, the blocks have to define a submodular function. In our case, this can be easily achieved by putting points with the same label in one block ([17] give a simple proof of this).

3 Experiments

We have tested our framework with several combinations of loss function, hash function, number of bits, datasets, and comparing with several state-of-the-art hashing methods (see suppl. mat.). We report a representative subset to show the flexibility of the approach. We use the KSH (3) [19] and eSPLH [29] loss functions. We test quadratic surrogate and GraphCut methods for the $\mathbf{Z}$ step in MAC. As hash functions (for each bit), we use linear SVMs (trained with LIBLINEAR; [9]) and kernel SVMs (with 500 basis functions).

We use the following labeled datasets: (1) CIFAR [13] contains 60 000 images in 10 classes. We use $D = 320$ GIST features [23] from each image. We use 58 000 images for training and 2 000 for test. (2) Infinite MNIST [20]. We generated, using elastic deformations of the original MNIST handwritten digit dataset, 1 000 000 images for training and 2 000 for test, in 10 classes. We represent each image by a $D = 784$ vector of raw pixels. Because of the computational cost of affinity-based methods, previous work has used training sets limited to a few thousand points [14, 16, 19, 22]. We train the hash functions in a subset of 10 000 points of the training set, and report precision and recall by searching for a test query on the entire dataset (the base set).

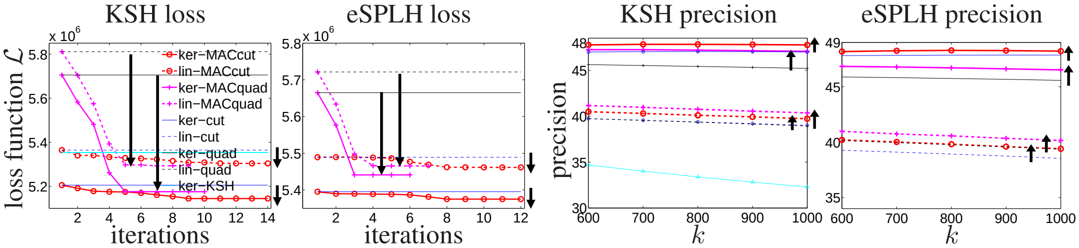

Figure 1: Loss function $\mathcal{L}$ and precision for k retrieved points for KSH and eSPLH loss functions on CIFAR dataset, using $b = 48$ bits.

We report precision (and precision/recall in the suppl. mat.) for the test set queries using as ground truth (set of true neighbors in original space) all the training points with the same label. The retrieved set contains the k nearest neighbors of the query point in the Hamming space. We report precision for different values of k to test the robustness of different algorithms.

The main comparison point are the quadratic surrogate and GraphCut methods of Lin et al. [16, 17], which we denote in this section as *quad* and *cut*, respectively, regardless of the hash function that fits the resulting codes. Correspondingly, we denote the MAC version of these as *MACquad* and *MACcut*, respectively. We use the following schedule for the penalty parameter μ in the MAC algorithm (regardless of the hash function type or dataset). We initialize $\mathbf{Z}$ with $\mu = 0$, i.e., the result of *quad* or *cut*. Starting from $\mu_1 = 0.3$ (*MACcut*) or 0.01 (*MACquad*), we multiply μ by 1.4 after each iteration ($\mathbf{Z}$ and $\mathbf{h}$ step).

Our experiments show our MAC algorithm indeed finds hash functions with a significantly and consistently lower objective value than rounding or two-step approaches (in particular, *cut* and *quad*); and that it outperforms other state-of-the-art algorithms on different datasets, with *MACcut* beating *MACquad* most of the time. The improvement in precision makes using MAC well worth the relatively small extra runtime and minimal additional implementation effort it requires. In all our plots, the vertical arrows indicate the improvement of *MACcut* over *cut* and of *MACquad* over *quad*.

The MAC algorithm finds better optima *The goal of this paper is not to introduce a new affinity-based loss or hash function, but to describe a generic framework to construct algorithms that optimize a given combination thereof.* We illustrate its effectiveness here with the CIFAR dataset, with different sizes of retrieved neighbor sets, and using 16 to 48 bits. We optimize two loss functions (KSH from eq. (3) and eSPLH), and two hash functions (linear and kernel SVM). In all cases, the MAC algorithm achieves a better hash function both in terms of the loss and of the precision/recall. We compare 4 ways of optimizing the loss function: *quad* [16], *cut* [17], *MACquad* and *MACcut*.

For each point $\mathbf{x}_n$ in the training set, we use $\kappa_+ = 100$ positive and $\kappa_- = 500$ negative neighbors, chosen at random to have the same or a different label as $\mathbf{x}_n$, respectively. Fig. 1 (panels 1 and 3) shows the KSH loss function for all the methods (including the original KSH method in [19]) over iterations of the MAC algorithm (KSH, *quad* and *cut* do not iterate), as well as precision and recall. 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, as well as the original KSH algorithm (cyan), in all cases: type of hash function (linear: dashed lines, kernel: solid lines) and number of bits $b = 16$ to 48 (suppl. mat.). Hence, applying MAC is always beneficial. Reducing the loss nearly always translates into better precision and recall (with a larger gain for linear than for kernel hash functions, usually). 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 within the same algorithm.

We usually find *cut* outperforms *quad* (in agreement with [17]), and correspondingly *MACcut* outperforms *MACquad*. Interestingly, *MACquad* and *MACcut* end up being very similar even though they started very differently. This suggests it is not crucial which of the two methods to use in the MAC $\mathbf{Z}$ step, although we still prefer *cut*, because it usually produces somewhat better optima. Finally, fig. 1 (panels 2 and 4) also shows the *MACcut* results using the eSPLH loss. All settings are as in the first KSH experiment. As before, *MACcut* outperforms *cut* in both loss function and precision/recall using either a linear or a kernel SVM.

Why does MAC learn better hash functions? In both the two-step and MAC approaches, the starting point are the “free” binary codes obtained by minimizing the loss over the codes without

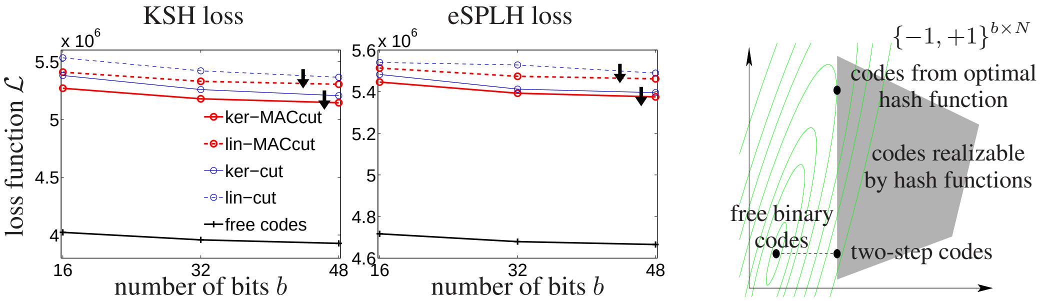

Figure 2: *Panels 1–2*: like fig. 1 but showing the value of the error function $E(\mathbf{Z})$ of eq. (7) for the "free" binary codes, and for the codes produced by the hash functions learned by *cut* (the two-step method) and *MACcut*, with linear and kernel hash functions. *Panel 3*: illustration of free codes, two-step codes and optimal codes realizable by a hash function, in the space $\{-1,+1\}^{b\times N}$.

them being the output of a particular hash function. That is, minimizing (4) without the "$\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$" constraints:

$$\min_{\mathbf{Z}} E(\mathbf{Z}) = \textstyle\sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\, y_{nm}), \qquad \mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1,+1\}^b. \tag{7}$$

The resulting free codes try to achieve good precision/recall independently of whether a hash function can actually produce such codes. Constraining the codes to be realizable by a specific family of hash functions (say, linear), means the loss $E(\mathbf{Z})$ will be larger than for free codes. How difficult is it for a hash function to produce the free codes? Fig. 2 (panels 1–2) plots the loss function for the free codes, the two-step codes from *cut*, and the codes from *MACcut*, for both linear and kernel hash functions in the same experiment as in fig. 1. It is clear that the free codes have a very low loss $E(\mathbf{Z})$, which is far from what a kernel function can produce, and even farther from what a linear function can produce. Both of these are relatively smooth functions that cannot represent the presumably complex structure of the free codes. This could be improved by using a very flexible hash function (e.g. using a kernel function with many centers), which could better approximate the free codes, but 1) a very flexible function would likely not generalize well, and 2) we require fast hash functions for fast retrieval anyway. Given our linear or kernel hash functions, what the two-step *cut* optimization does is fit the hash function directly to the free codes. This is not guaranteed to find the best hash function in terms of the original problem (1), and indeed it produces a pretty suboptimal function. In contrast, MAC gradually optimizes both the codes and the hash function so they eventually match, and finds a better hash function for the original problem (although it is still not guaranteed to find the globally optimal function of problem (1), which is NP-complete).

Fig. 2 (right) shows this 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 $\{\mathbf{Z} \in \{-1,+1\}^{b\times N}\colon\ \mathbf{Z} = \mathbf{h}(\mathbf{X})$ 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 $\mathbf{h}$.

Runtime The runtime per iteration for our 10 000-point training sets with $b = 48$ bits and $\kappa_+ = 100$ and $\kappa_- = 500$ neighbors in a laptop is 2' for both *MACcut* and *MACquad*. They stop after 10–20 iterations. Each iteration is comparable to a single *cut* or *quad* run, since the $\mathbf{Z}$ step dominates the computation. The iterations after the first one are faster because they are warm-started.

Comparison with binary hashing methods Fig. 3 shows results on CIFAR and Infinite MNIST. We create affinities y_{nm} for all methods using the dataset labels as before, with $\kappa_+ = 100$ similar neighbors and $\kappa_- = 500$ dissimilar neighbors. We compare *MACquad* and *MACcut* with Two-Step Hashing (*quad*) [16], FastHash (*cut*) [17], Hashing with Kernels (KSH) [19], Iterative Quantization (ITQ) [11], Binary Reconstructive Embeddings (BRE) [14] and Self-Taught Hashing (STH) [33]. *MACquad*, *MACcut*, *quad* and *cut* all use the KSH loss function (3). The results show that *MACcut* (and *MACquad*) generally outperform all other methods, often by a large margin, in nearly all situations (dataset, number of bits, size of retrieved set). In particular, *MACcut* and *MACquad* are the only ones to beat ITQ, as long as one uses sufficiently many bits.

4 Discussion

The two-step approach of Two-Step Hashing [16] and FastHash [17] is a significant advance in finding good codes for binary hashing, but it also causes a maladjustment between the codes and the

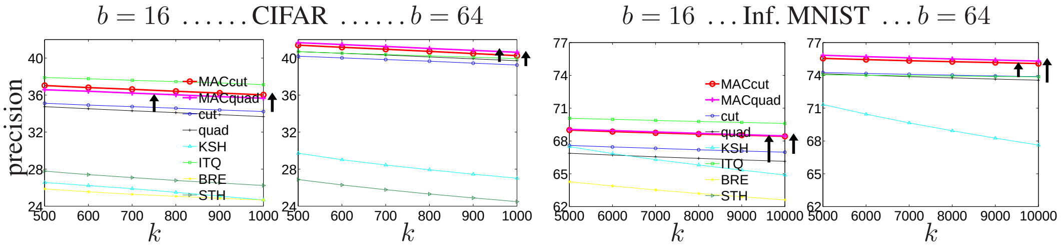

Figure 3: Comparison with binary hashing methods on CIFAR (left) and Infinite MNIST (right), using a linear hash function, using $b = 16$ to 64 bits (suppl. mat.). Each plot shows the precision for k retrieved points, for a range of k.

hash function, since the codes were learned without knowledge of what kind of hash function would use them. Ignoring the interaction between the loss and the hash function limits the quality of the results. For example, a linear hash function will have a harder time than a nonlinear one at learning such codes. In our algorithm, this tradeoff is enforced gradually (as μ increases) in the $\mathbf{Z}$ step as a regularization term (eq. (5)): it finds the best codes according to the loss function, but makes sure they are close to being realizable by the current hash function. Our experiments demonstrate that significant, consistent gains are achieved in both the loss function value and the precision/recall in image retrieval over the two-step approach. Note that the objective (5) is not an ad-hoc combination of a loss over the hash function and a loss over the codes; it follows by applying MAC to the well-defined top-level problem (1), and it solves it in the limit of large μ (up to local optima).

What is the best type of hash function to use? The answer to this is not unique, as it depends on application-specific factors: quality of the codes produced (to retrieve the correct images), time to compute the codes on high-dimensional data (since, after all, the reason to use binary hashing is to speed up retrieval), ease of implementation within a given hardware architecture and software libraries, etc. Our MAC framework facilitates this choice considerably, because training different types of hash functions simply involves reusing an existing classification algorithm within the $\mathbf{h}$ step, with no changes to the $\mathbf{Z}$ step.

5 Conclusion

We have proposed a general framework for optimizing binary hashing using affinity-based loss functions. It improves over previous, two-step approaches based on learning binary codes first and then learning the hash function. Instead, it optimizes jointly over the binary codes and the hash function in alternation, so that the binary codes eventually match the hash function, resulting in a better local optimum of the affinity-based loss. This was possible by introducing auxiliary variables that conditionally decouple the codes from the hash function, and gradually enforcing the corresponding constraints. Our framework makes it easy to design an optimization algorithm for a new choice of loss function or hash function: one simply reuses existing software that optimizes each in isolation. The resulting algorithm is not much slower than the two-step approach—it is comparable to iterating the latter a few times—and well worth the improvement in precision/recall.

The step over the hash function is essentially a solved problem if using a classifier, since this can be learned in an accurate and scalable way using machine learning techniques. The most difficult and time-consuming part in our approach is the optimization over the binary codes, which is NP-complete and involves many binary variables and terms in the objective. Although some techniques exist [16, 17] that produce practical results, designing algorithms that reliably find good local optima and scale to large training sets is an important topic of future research.

Another direction for future work involves learning more sophisticated hash functions that go beyond mapping image features onto output binary codes using simple classifiers such as SVMs. This is possible because the optimization over the hash function parameters is confined to the $\mathbf{h}$ step and takes the form of a supervised classification problem, so we can apply an array of techniques from machine learning and computer vision. For example, it may be possible to learn image features that work better with hashing than standard features such as SIFT, or to learn transformations of the input to which the binary codes should be invariant, such as translation, rotation or alignment.

Acknowledgments Work supported by NSF award IIS–1423515.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. ACM*, 51(1):117–122, Jan. 2008.

[2] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, June 2003.

[3] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE PAMI*, 26(9):1124–1137, Sept. 2004.

[4] M. Carreira-Perpiñán. The elastic embedding algorithm for dimensionality reduction. *ICML*, 2010.

[5] M. Carreira-Perpiñán and R. Raziperchikolaei. Hashing with binary autoencoders. *CVPR*, 2015.

[6] M. Carreira-Perpiñán and M. Vladymyrov. A fast, universal algorithm to learn parametric nonlinear embeddings. *NIPS*, 2015.

[7] M. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. arXiv:1212.5921 [cs.LG], Dec. 24 2012.

[8] M. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. *AISTATS*, 2014.

[9] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874, Aug. 2008.

[10] T. Ge, K. He, and J. Sun. Graph cuts for supervised binary coding. *ECCV*, 2014.

[11] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A Procrustean approach to learning binary codes for large-scale image retrieval. *IEEE PAMI*, 35(12):2916–2929, Dec. 2013.

[12] K. Grauman and R. Fergus. Learning binary hash codes for large-scale image search. In R. Cipolla, S. Battiato, and G. Farinella, editors, *Machine Learning for Computer Vision*, pages 49–87. Springer-Verlag, 2013.

[13] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, U. Toronto, 2009.

[14] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. *NIPS*, 2009.

[15] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *IEEE PAMI*, 34(6):1092–1104, 2012.

[16] G. Lin, C. Shen, D. Suter, and A. van den Hengel. A general two-step approach to learning-based hashing. *ICCV*, 2013.

[17] G. Lin, C. Shen, Q. Shi, A. van den Hengel, and D. Suter. Fast supervised hashing with decision trees for high-dimensional data. *CVPR*, 2014.

[18] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. *ICML*, 2011.

[19] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. *CVPR*, 2012.

[20] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 301–320. MIT Press, 2007.

[21] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, second edition, 2006.

[22] M. Norouzi and D. Fleet. Minimal loss hashing for compact binary codes. *ICML*, 2011.

[23] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Computer Vision*, 42(3):145–175, May 2001.

[24] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, Dec. 22 2000.

[25] L. J. P. van der Maaten. Barnes-Hut-SNE. In *Int. Conf. Learning Representations (ICLR 2013)*, 2013.

[26] L. J. P. van der Maaten and G. E. Hinton. Visualizing data using t-SNE. *JMLR*, 9:2579–2605, Nov. 2008.

[27] M. Vladymyrov and M. Carreira-Perpiñán. Partial-Hessian strategies for fast learning of nonlinear embeddings. *ICML*, 2012.

[28] M. Vladymyrov and M. Carreira-Perpiñán. Linear-time training of nonlinear low-dimensional embeddings. *AISTATS*, 2014.

[29] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for large scale search. *IEEE PAMI*, 2012.

[30] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. *NIPS*, 2009.

[31] Z. Yang, J. Peltonen, and S. Kaski. Scalable optimization for neighbor embedding for visualization. *ICML*, 2013.

[32] S. X. Yu and J. Shi. Multiclass spectral clustering. *ICCV*, 2003.

[33] D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. *SIGIR*, 2010.

[34] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: FORTRAN subroutines for large-scale bound-constrained optimization. *ACM Trans. Mathematical Software*, 23(4):550–560, 1997.