Optimizing Affinity-Based Binary Hashing Using Auxiliary Coordinates

Ramin Raziperchikolaei RRAZIPERCHIKOLAEI@UCMERCED.EDU
Miguel Á. Carreira-Perpiñán MCARREIRA-PERPINAN@UCMERCED.EDU
Electrical Engineering and Computer Science, School of Engineering, University of California, Merced

1. Introduction

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 to the vector for the query image, according to a suitable distance (Shakhnarovich et al., 2006). 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 $\mathcal{O}(ND)$. This can be approximated by *binary hashing* (Grauman & Fergus, 2013). 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*. The general approach consists of defining a supervised objective that has a small value for good hash functions and minimizing it. We focus here on *affinity-based loss functions*, which directly try to preserve the original similarities in the binary space and has the following the form

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

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 or similarity measures; Grauman & Fergus, 2013). The sum is often restricted to a subset of image pairs (n, m) to keep the runtime low.

In binary hashing, the optimization is difficult, because the hash function must output binary values, hence *the problem is not just generally nonconvex, but also nonsmooth*. One can compute the gradients of the objective function wrt the parameters of the hash function, but this is not useful because they are zero nearly everywhere. Most hashing approaches propose 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.

Of the three-step suboptimal approach mentioned (learn continuous codes, binarize them, learn hash function), recent works manage to join the first two steps and hence learn binary codes. Then, one learns the hash function given these binary codes. Can we do better? Indeed, in this work *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. In fact, our framework can be seen as an iterated, corrected version of the two-step approach: learn binary codes *given the current hash function*, learn hash functions given codes, iterate. The key to achieve this 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.

2. Nonlinear embedding and affinity-based loss functions for binary hashing

The dimensionality reduction literature has developed a number of objective functions 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. A representative example is the elastic embedding (Carreira-Perpiñán, 2010), where $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form:

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

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. Although these models were developed to produce continuous projections, they have been successfully used for binary hashing too by truncating their codes (Weiss et al., 2009) or using the two-step approach of (Lin et al., 2014).

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, for Supervised Hashing with Kernels (KSH) $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form

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

where y_{nm} is 1 if $\mathbf{x}_n$, $\mathbf{x}_m$ are similar and -1 if they are dissimilar. Binary Reconstructive Embeddings (Kulis & Darrell, 2009) 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 (Wang et al., 2012) proposed by Lin et al. (2013) (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.

3. 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)* (Carreira-Perpiñán & Wang, 2012; 2014), 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:

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

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 a penalty method, such as the quadratic-penalty (Nocedal & Wright, 2006). We solve the following minimization problem (unconstrained again, but dependent on μ) while progressively increasing μ, so the constraints are eventually satisfied:

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

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 (described in detail later):

- 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 (Lin et al., 2013; 2014) 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 very similar to the two-step (TSH) approach of Lin et al. (2013), 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 $n = 1, \dots, N$.

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. 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 (Nocedal & Wright, 2006), which was the setting considered in the original MAC papers (Carreira-Perpiñán & Wang, 2012; 2014). 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.

3.1. $\mathbf{h}$ step

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

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

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.

3.2. Z step

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 (Lin et al., 2013), and a GraphCut method (Lin et al., 2014).

In both the quadratic surrogate and the GraphCut method, 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. Here, we only explain the solution using the GraphCut algorithm. In our experiments, we show the results for both quadratic surrogate and GraphCut method.We start by describing the GraphCut method in its original form (which applies to the loss function over binary codes, i.e., the first term in $\mathcal{L}_P$), and then we give our modification to make it work with our $\mathbf{Z}$ step objective (the regularized loss function over binary codes, i.e., the complete $\mathcal{L}_P$).

Solution using a GraphCut algorithm (Lin et al., 2014) To optimize over the ith bit (given all the other bits are fixed), we have to minimize the following objective function (see (Lin et al., 2013) for details):

$$\min_{\mathbf{z}_{(i)}} \sum_{n,m=1}^{N} \frac{1}{2} z_{ni} z_{mi} a_{nm} + \mu \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2.$$

where z_{ni} is the ith bit of the nth point and $a_{nm} \in \mathbb{R}$ is constant. In general, this is an NP-complete problem over N bits (the ith bit for each image), with the form of a quadratic function on binary variables. We can apply the GraphCut algorithm (Boykov & Kolmogorov, 2003; 2004; Kolmogorov & Zabih, 2003), as proposed by the FastHash algorithm of Lin et al. (2014). 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 $\mathcal{B}$, we can rewrite equation (3.2) as:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \textstyle\sum_{n,m\in\mathcal{B}} a_{nm} z_{ni} z_{mi} + 2 \sum_{n\in\mathcal{B}, m\notin\mathcal{B}} a_{nm} z_{ni} z_{mi} - \mu \sum_{n\in\mathcal{B}} z_{ni} h_i(\mathbf{x}_n).$$

We then rewrite this equation in the standard form for the GraphCut algorithm:

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

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. This can be easily achieved by putting points with the same label in one block.

Unlike in the quadratic surrogate method, using the GraphCut algorithm with alternating optimization on blocks defining submodular functions is guaranteed to find a $\mathbf{Z}$ that has a lower or equal objective value that the initial one, and therefore to decrease monotonically $\mathcal{L}_P$.

4. Experiments

In this section, we report a small subset of our experiments. More results can be found in the main paper (Raziperchikolaei & Carreira-Perpiñán, 2016). We use the KSH (3) and eSPLH as the loss functions, quadratic surrogate and GraphCut methods for the $\mathbf{Z}$ step in MAC, and linear and kernel SVMs as the hash functions. Here, we only show the results on the CIFAR (Krizhevsky, 2009) dataset that contains 60 000 images in 10 classes. We use $D = 320$ GIST features (Oliva & Torralba, 2001) from each image.

The main comparison points are the quadratic surrogate and GraphCut methods of Lin et al. (2013; 2014), which

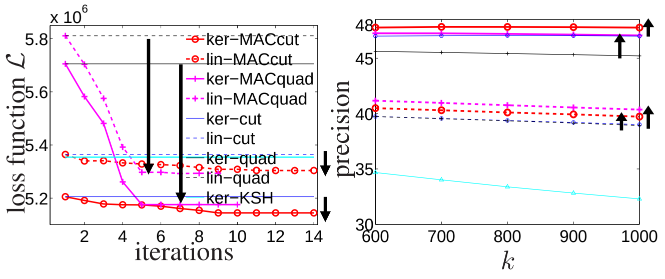

Figure 1. KSH on CIFAR dataset, using $b = 48$ bits.

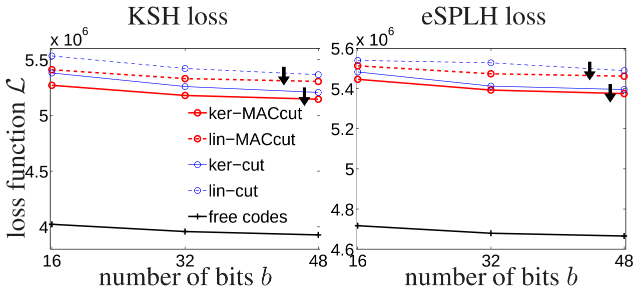

Figure 2. Like fig. 1 but showing the value of the error function $E(\mathbf{Z})$ of eq. (6) for the "free" binary codes, and for the codes produced by the hash functions learned by *cut* (the two-step method) and *MACcut*, with linear and kernel hash functions.

we denote in this section as *quad* and *cut*, respectively. Correspondingly, we denote the MAC version of these as *MACquad* and *MACcut*, respectively.

The MAC algorithm finds better optima *Our goal 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.

Fig. 1(left) shows the KSH loss function for all the methods 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. 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 within the same algorithm. 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.

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

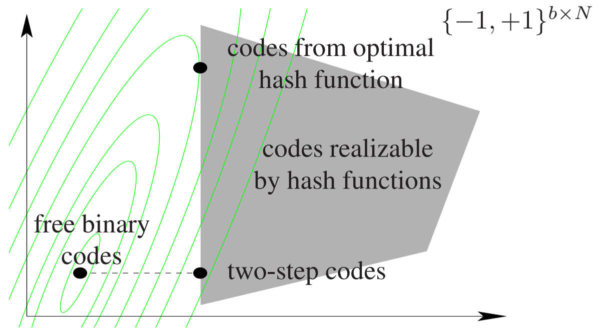
$\{-1,+1\}^{b\times N}$
codes from optimal hash function
codes realizable by hash functions
free binary codes
two-step codes

Figure 3. Illustration of free codes, two-step codes and optimal codes realizable by a hash function, in the space $\{-1,+1\}^{b\times N}$.

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

$$\min_{\mathbf{Z}} E(\mathbf{Z}) = \sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}),\ \begin{cases} \mathbf{z}_1,\dots,\mathbf{z}_N \\ \in \{-1,+1\}^b. \end{cases} \quad (6)$$

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 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)).

Fig. 3 illustrates 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 $\mathbf{h}$.

References

Boykov, Yuri and Kolmogorov, Vladimir. Computing geodesics and minimal surfaces via graph cuts. In *Proc. 9th Int. Conf. Computer Vision (ICCV'03)*, pp. 26–33, Nice, France, October 14–17 2003.

Boykov, Yuri and Kolmogorov, Vladimir. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, September 2004.

Carreira-Perpiñán, Miguel Á. The elastic embedding algorithm for dimensionality reduction. In Fürnkranz, Johannes and Joachims, Thorsten (eds.), *Proc. of the 27th Int. Conf. Machine Learning (ICML 2010)*, pp. 167–174, Haifa, Israel, June 21–25 2010.

Carreira-Perpiñán, Miguel Á. and Wang, Weiran. Distributed optimization of deeply nested systems. arXiv:1212.5921 [cs.LG], December 24 2012.

Carreira-Perpiñán, Miguel Á. and Wang, Weiran. Distributed optimization of deeply nested systems. In Kaski, Samuel and Corander, Jukka (eds.), *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pp. 10–19, Reykjavik, Iceland, April 22–25 2014.

Grauman, Kristen and Fergus, Rob. Learning binary hash codes for large-scale image search. In Cipolla, R., Battiato, S., and Farinella, G. (eds.), *Machine Learning for Computer Vision*, pp. 49–87. Springer-Verlag, 2013.

Kolmogorov, Vladimir and Zabih, Ramin. What energy functions can be minimized via graph cuts? *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(2):147–159, February 2003.

Krizhevsky, Alex. Learning multiple layers of features from tiny images. Master's thesis, Dept. of Computer Science, University of Toronto, April 8 2009.

Kulis, Brian and Darrell, Trevor. Learning to hash with binary reconstructive embeddings. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 22, pp. 1042–1050. MIT Press, Cambridge, MA, 2009.

Lin, Guosheng, Shen, Chunhua, Suter, David, and van den Hengel, Anton. A general two-step approach to learning-based hashing. In *Proc. 14th Int. Conf. Computer Vision (ICCV'13)*, pp. 2552–2559, Sydney, Australia, December 1–8 2013.

Lin, Guosheng, Shen, Chunhua, Shi, Qinfeng, van den Hengel, Anton, and Suter, David. Fast supervised hashing with decision trees for high-dimensional data. In *Proc. of the 2014 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'14)*, pp. 1971–1978, Columbus, OH, June 23–28 2014.

Nocedal, Jorge and Wright, Stephen J. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

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

Raziperchikolaei, Ramin and Carreira-Perpiñán, Miguel Á. Optimizing affinity-based binary hashing using auxiliary coordinates. arXiv:1501.05352 [cs.LG], February 5 2016.

Shakhnarovich, Gregory, Indyk, Piotr, and Darrell, Trevor (eds.). *Nearest-Neighbor Methods in Learning and Vision*. Neural Information Processing Series. MIT Press, Cambridge, MA, 2006.

Wang, Jun, Kumar, Sanjiv, and Chang, Shih-Fu. Semi-supervised hashing for large scale search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(12): 2393–2406, December 2012.

Weiss, Yair, Torralba, Antonio, and Fergus, Rob. Spectral hashing. In Koller, Daphne, Bengio, Yoshua, Schuurmans, Dale, Bottou, Leon, and Culotta, Aron (eds.), *Advances in Neural Information Processing Systems (NIPS)*, volume 21, pp. 1753–1760. MIT Press, Cambridge, MA, 2009.