Learning Independent, Diverse Binary Hash Functions: Pruning and Locality

Ramin Raziperchikolaei and Miguel Á. Carreira-Perpiñán
Electrical Engineering and Computer Science
University of California, Merced. Merced, CA, USA
Emails: rraziperchikolaei@ucmerced.edu, mcarreira-perpinan@ucmerced.edu

***Abstract*—Information retrieval in large databases of complex objects, such as images, audio or documents, requires approximate search algorithms in practice, in order to return semantically similar objects to a given query in a reasonable time. One practical approach is supervised binary hashing, where each object is mapped onto a small binary vector so that Hamming distances approximate semantic similarities, and the search is done in the binary space more efficiently. Much work has focused on designing objective functions and optimization algorithms for learning b-bit hash functions from a dataset. Recent work has shown that comparable or better results can be obtained by training b hash functions independently from each other and making them cooperate by introducing diversity with ensemble learning techniques. We show that this can be further improved by two techniques: pruning an ensemble of hash functions, and learning local hash functions. We show how it is possible to train our improved algorithms in datasets orders of magnitude larger than those used by most works on supervised binary hashing.**

***Keywords*-binary hashing; ensemble diversity; optimization; information retrieval.**

I. Introduction

Efficient k-nearest neighbor search has many applications in data mining and information retrieval problems. As an example, in image retrieval problems, we are interested in finding similar images to a query by searching a very large database of images. The exact nearest neighbor search takes $\mathcal{O}(ND)$ in both time and space if we have N images in D dimensional space. Since images are usually represented by high-dimensional feature vectors and there are millions of images in the database, the exact search is very slow. For this reason, several efficient and fast algorithms are proposed to solve the same problem approximately. In this paper, we will focus on binary hashing. A binary hash function $\mathbf{h}(\cdot)$, takes a point $\mathbf{x} \in \mathbb{R}^D$ in high dimensional space and maps it to a b-bit binary code $\mathbf{z} \in \{0,1\}^b$. Finding nearest neighbors of a query in binary space is much faster because it uses efficient hardware operations to compute Hamming distances, and because the entire database in binary form takes only bN bits, which may fit in main memory.

While binary hashing helps in accelerating the search speed significantly, it introduces errors: the result of nearest neighbor search in binary space is different from the exact search. The main goal of hashing papers is to make this difference as small as possible. Many recent hashing methods try to *learn* the hash functions by optimizing an objective function defined on the training points, which has two main goals. The first goal is *to learn binary codes that preserve the neighborhood*. This can be achieved by minimizing an objective that returns a small value when similar (dissimilar) points in original space are mapped into nearby (far away) binary codes. The second goal is *to make the hash functions different from each other*, since nothing is gained if they are identical. Exactly how this is incorporated into the optimization depends on each method. In many methods (for example, those based on the well-known Laplacian loss; [1], [2], [3]) this is done by adding orthogonality constraints or penalties.

Most optimization-based hashing papers propose either a new objective function [4], [5], [6] or a new optimization algorithm [7], [8] to optimize the existing objectives. Optimizing these objective functions is difficult and slow because they contain bN binary variables that are coupled together, which makes the objective nondifferentiable. Even with the best binary optimization algorithms (such as mincut [9] and GraphCut [10], possibly combined with the method of auxiliary coordinates [11]), we cannot train the hash functions on more than a few thousand points in a reasonable amount of time.

A recently proposed method, Independent Laplacian Hashing (ILH) [12], takes a very different approach. ILH drops the interaction between the 1-bit hash functions in the objective and replaces it with a diversity-inducing mechanism based on ensemble learning techniques. The 1-bit hash functions are learned *independently*, so the optimization is vastly easier, more efficient, embarrassingly parallel, and can use larger training sets. (In fact, in the single-bit case all binary hashing objective functions can be written in the same mathematical form, namely a quadratic binary problem.) And, as shown in [12], ILH actually beats optimization-based approaches in precision-recall.

In this paper, we propose two important improvements to ILH: in section III, *pruning* a large set of 1-bit hash functions gives a small subset of functions with comparable retrieval quality; and in section IV, learning *local hash functions* introduces further diversity and improves the retrieval quality. Our experiments in section V show that these simple, very efficient methods learn hash functions that can beat almost all the state-of-the-art methods in image retrieval tasks.

A. Related work

There are two main approaches to achieve binary hash functions: (1) data independent approach where there is no training (such as considering the hash functions as the random hyperplanes [13], [14]), and (2) data dependent approach that learns the hash functions by minimizing an objective function using the training points (it performs better than the first approach). The data dependent methods can be either supervised, where semantic similarity determines similar and dissimilar pairs of points [5], [7], or unsupervised, where the distance between the points in the original space is the similarity measure [15], [16], [17]. Here we focus on supervised data dependent methods.

Several affinity-based objective functions have been proposed in the literature. They have two main goals: (1) preserving the neighborhood by creating an affinity matrix of the points based on their semantic similarity, and (2) making the hash functions different from each other. Most hashing methods define a loss function that couples all the 1-bit hash functions and use optimization techniques to achieve the mentioned goals [5], [4], [6]. Binary reconstructive embedding [4] uses direct alternating optimization over the weights of hash functions to minimize the objective function. Supervised hashing with kernels [5] ignores the binary codes, optimizes the continuous objective function, and truncates the results to get the binary codes. To optimize the objective functions better, two-step methods are proposed [1], [18], [7]. They define the objective function on binary codes, learn the codes using alternating optimization, and finally fit the hash functions (classifiers) on the binary codes. More recently, methods are proposed that try to optimize the binary codes and hash functions jointly [3], [8]. Hashing with auxiliary coordinates [8] defines the new binary coordinates as the output of the hash functions and uses quadratic penalty methods to define an objective function over both binary codes and hash functions, which can be later optimized in an alternation between the codes and functions.

Bagging has been recently used to improve the performance of binary hash functions learned with PCA as the number of bits increases [19]. The motivation of using ensemble learning diversity techniques in ILH [12] is different: to uncouple the optimization in supervised binary hashing.

II. Independent Laplacian Hashing (ILH)

Independent Laplacian Hashing (ILH) [12] is the first supervised hashing method that uses diversity techniques to make the hash functions differ. Consider the problem of learning a b-bit hash function, which maps a given input point $\mathbf{x} \in \mathbb{R}^D$ to a b-bit vector in $\{-1,+1\}^b$, given a training set containing supervisory information in the form of pairs of similar, dissimilar or indifferent points. Traditionally, one would define an objective function of the b-bit hash function containing constraints or penalties to make the b bits differ. *The fundamental idea in ILH is to define b independent objective functions each operating on a 1-bit hash function, and make them differ via diversity-inducing techniques from ensemble learning, such as using a different training subset for each bit*. Specifically, each 1-bit objective function has the form (equivalent to a Laplacian loss):

$$\min_{\mathbf{h}} \mathbf{h}(\mathbf{X})\,\mathbf{A}\,\mathbf{h}(\mathbf{X})^T = \sum_{n,m=1}^{N} a_{nm}\, h(\mathbf{x}_n)\, h(\mathbf{x}_m) \qquad (1)$$

where $\mathbf{h}(\mathbf{X}) = (h(\mathbf{x}_1), \dots, h(\mathbf{x}_n)) \in \{-1,+1\}^N$ is a row vector of N bits, $\mathbf{A} = (a_{nm}) \in \mathbb{R}^{N\times N}$ is the affinity matrix for the training set $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N) \in \mathbb{R}^{D\times N}$, and the minimization is over the parameters of $\mathbf{h}$. For example, $\mathbf{h}$ can be a thresholded linear function, and the affinity a_{mn} can be $+1$, -1 or 0 for similar, dissimilar or indifferent points $\mathbf{x}_n$ and $\mathbf{x}_m$. If the training sets differ across hash functions, then so do the affinity matrices and we learn different functions.

Optimizing the objective (1) b times independently to learn the b hash functions has several advantages over optimizing an objective functions that couple all b hash functions: (1) the large binary optimization over bN variables separates into b independent optimizations over only N variables, which can be solved faster and more accurately; (2) training is embarrassingly parallel; and (3) model selection (to choose the best number of bits) becomes easier. ILH uses the two-step optimization to minimize (1).

Importantly, the quadratic form of the objective in (1) is not restrictive. As shown in [12], *any affinity-based objective function (where each term depends on a pair of points) that uses a 1-bit hash function can be written as a quadratic function*. This is not true if using $b > 1$ bits, and indeed many different such objective functions have been proposed.

Various diversity-inducing mechanisms are possible [12], including using different initializations for the optimization, using different subsets of features of $\mathbf{x}$, and using different and disjoint subsets of training points picked randomly (referred to as ILHt in [12]). The latter works best and is the one we use in this paper; we will refer to it as ILH.

III. Pruning a set of hash functions: ILH-prune

Given a set of b 1-bit hash functions, the goal is to select a subset of s hash functions which performs comparably well in a retrieval task. The resulting s-bit codes are, of course, faster to search than the b-bit codes. (Our pruning algorithm is applicable to any set of hash functions, although we will apply it to the result of ILH.) This is similar to pruning a classifier ensemble [20], an effective and widely used technique to produce very good, compact classifiers.

Ideally, we seek the subset of hash functions that maximizes the precision (or other measure such as the F-score) on a given test set of queries. A brute-force search is impractical because there are $\binom{b}{s}$ subsets. We solve this combinatorial problem approximately with a greedy algorithm, *sequential forward selection*, which is considered the most

effective in ensemble learning [21], [20]. Starting with an empty set, it repeatedly adds the hash function that, when combined with the current set, gives highest precision. We stop when we reach s functions, where s is set by the user (e.g. to achieve a desired test runtime), or when we reach a desired percentage of the precision of the entire set of b functions; note it is possible for a smaller subset to exceed this precision. The result is deterministic given the b hash functions (unless there are ties) and defines an order in which the functions should be picked.

Assuming a test set of M query vectors and a base set (where we search) of N vectors, all of dimension D, the computational complexity of pruning is linear on the base set size. Indeed, computing the binary codes for all points is $\mathcal{O}(bD(M+N))$ (assuming linear hash functions), which we need to do even if we do not use pruning. Then, for $i = 1, \dots, b-1$ we have to determine the optimal ith hash function among the remaining $b-i+1$. The required Hamming distances can be updated incrementally, by adding to each distance so far (using the already selected $i-1$ functions) the distance corresponding to the ith bit if using the jth remaining function; this is $\mathcal{O}(MN)$. Then, in order to find the function that gives the best precision, we need to sort the N distances for each of the M queries, which can be done in linear time using Counting Sort (since the distances are in $\{0, \dots, b\}$ and $b \ll N$) [22]; this is $\mathcal{O}(MN)$. This has to be done for each of the $b-i+1$ remaining hash functions, so each step is $\mathcal{O}((b-i+1)MN)$, and the total for the for loop is $\mathcal{O}(b^2MN)$. The grand total is $\mathcal{O}(b^2MN + bD(N+M)) = \mathcal{O}(bN(bM+D))$ if $M \ll N$. Finally, each of the steps above parallelizes embarrassingly over the query set and over the hash functions.

IV. Training local hash functions: ILH-local

One problem with the selection of the training subsets in ILH is that, although disjoint, they will have high overlap over the input space, which will decrease the resulting diversity, particularly as the number of bits increases. We can avoid this by selecting *spatially local subsets*. Specifically, we define the training subset for a given hash function as a training point $\mathbf{x}_n$ (picked at random, the *seed* for that local subset) and its k nearest neighbors. This can be done in $\mathcal{O}(N(D + \log N))$ per hash function (for computing and sorting the distances).

The *locality parameter* k varies in $\{1, \dots, N-1\}$ (where N is the size of the whole training set) and controls how local the hash function is. If k is very small, the neighborhoods are very local and likely disjoint, and the hash function will look like a random hyperplane through that neighborhood. (For $k = 1$, a good hash function would simply be the bisecting hyperplane that assigns a different code to $\mathbf{x}_n$ and its nearest neighbor and maximizes the margin.) The functions will behave similarly to that of random projection algorithms such as LSH [13]. As k increases, the neighborhoods become less local spatially and less disjoint. They become identical for $k = N-1$, at which point there is no diversity, so the precision must drop for large k (though this will likely not happen unless $k \gg N/b$).

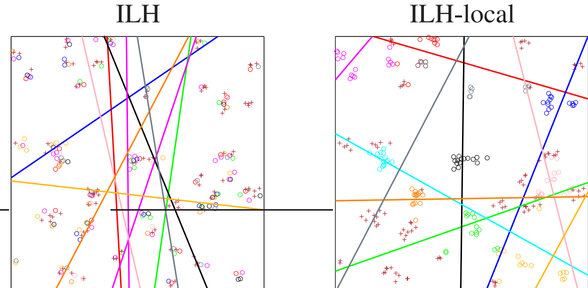

Figure 1. Learning local hash functions improves the diversity and neighborhood preservation. Data points marked as $\circ$ were used to train the hash functions (they are colored according to the hash function they train). Some points may have used for training different hash functions in ILH-local. Data points marked as $+$ were not used for training.

There are other ways to define local subsets (e.g. running a clustering algorithm such as k-means), but ours is nonparametric, requires no training and has no local optima. Likewise, other definitions of local neighborhood of a point $\mathbf{x}_n$ are possible, such as the points within a distance ϵ of $\mathbf{x}_n$, or even just the kth nearest neighbor of $\mathbf{x}_n$. Better sampling for the seeds could be done with k-centres or k-means++ [23], which improves the chances to get more local and disjoint subsets at the cost of a larger computation.

Fig. 1 illustrates this with a dataset of 250 2D points sampled from a Gaussian mixture with 50 components. We learn $b = 10$ hash functions. With ILH, some of the 1-bit hash functions are very similar to each other, which results in some small code regions with few or no data points at all. Pruning this set of functions would remove redundant ones, but also reduce the number of bits and code regions. ILH-local learns diverse hash functions that lie in different parts of the input space and produce more uniform code regions that better preserve neighborhoods. For example, note the clusters of points in the upper corners of each figure. ILH-local preserves the neighborhood by separating those clusters from each other and assigning them different binary codes, while ILH assigns the same binary code to several clusters, which destroys the similarity. Ideally we would like that a cluster be partitioned only by local hash functions, while being contained on a single region of nonlocal functions. This would preserve distances better for points in the cluster.

V. Experiments [1]

We show that our proposed methods ILH-prune (prunes a given set of hash functions) and ILH-local (learns local hash functions) improve the retrieval quality. We use three different datasets in our experiments to evaluate different methods: (1) CIFAR [24] dataset contains 60 000 points in

[1] Matlab code for the algorithms can be found in the authors' web pages.

10 classes. We consider $58\,000$ images for training and $2\,000$ images for test. Each image is represented by $D = 320$ SIFT features [25]. (2) Infinite MNIST [26]. 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 (3) Flickr [27] dataset contains $1\,000\,000$ points, each of them represented by $D = 150$ MPEG-7 edge histogram features. We randomly select $2\,000$ points for the test and consider the rest as the training set.

Most hashing papers use a small subset of training set to train the hash functions [4], [5], [18], [3]. In our experiments all the hash functions are trained using training subsets of $5\,000$ points that are selected randomly from the dataset (except ILH-local that picks local subsets). To evaluate the methods, we search the entire dataset to find the nearest neighbors of a query. We consider linear hash functions for all the methods. ILH-local and ILH-prune use linear SVMs (trained with LIBLINEAR [28]) as the 1-bit hash functions.

For CIFAR and Infinite MNIST, the positive neighbors of a point $\mathbf{x}$ are all the points with the same label as the label of $\mathbf{x}$. For Flickr, $K = 10\,000$ nearest neighbors of each point in original space are considered as the positive neighbors. The rest of the points are considered as the negative neighbors. The affinity matrix $\mathbf{A}$ contains 100 positive and 100 negative neighbors per training point, chosen randomly from all the positive and negative neighbors of that point.

To report the precision and recall, we define the ground-truth and retrieved sets. The ground-truth set of each query contains all the positive neighbors of that query among the points in the training set. The retrieved set contains the r nearest neighbors of the query in the Hamming space.

A. Pruning the learned hash functions

We investigate the effect of pruning a set of hash functions learned with ILH on the CIFAR and infinite MNIST datasets.

Precision as a function of number of bits b. In fig. 3 (left), we compare the pruning method with four different methods: (1) tPCA: we run PCA on the training set and truncate it to get the binary codes, (2) LSH [13]: hash functions are random hyperplanes that are achieved independently, (3) KSHcut [7]: it is a state-of-the-art method that its hash functions are coupled in the objective function (can not train them independently), (4) ILH [12]: the $b = 200$ 1-bit hash functions are trained independently by solving a binary quadratic problem. For ILH and LSH, we show 5 different random ordering of the hash functions. ILH-prune is our new algorithm: we start from the hash functions of ILH, we order them using the sequential forward selection pruning algorithm, and we report the precisions. We see that the ILH-prune always performs better than all the other methods. Pruning beats other methods with a large margin when we use smaller number of bits ($b < 32$). The reason is that as we increase the number of bits, the set of remained hash functions becomes smaller, and the algorithm has less options to choose from. Furthermore, for large number of bits, ILH and ILH-prune have a lot of hash functions in common (for $b = 200$ they are exactly the same).

Effect of the number of 1-bit functions on pruning. To investigate this, we first run ILH to learn a set of 200 hash functions. Then, from this set, we create subsets containing s hash functions randomly, where $s = 25, 50, 100, 150$ and 200. Finally, we run pruning algorithm on each of these subsets and report precision in fig. 3 (right). The figure shows that The curve for $s = 200$ is always above all the other curves (with $s < 200$). This means that by pruning a larger set of 1-bit hash functions, we can reach to a higher precision using a smaller number of bits.

How does the test set size affect the pruning. To explore this, we learn a set of 200 hash functions using ILH on CIFAR dataset. Then we prune this set using test sets of different sizes.

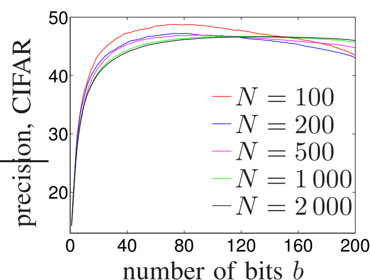

Figure 2. Pruning and test set size N.

In figure 2, we change the size of the test set from 100 to $2\,000$ and report the precision as a function of number of bits. For a test set with $N \leq 500$ points, the maximum precision is achieved by using less than 100 bits. For a large test sets (with $N = 2\,000$ points in the figure), the precision always increases as we increase the number of bits. Since in reality we expect an infinite test set, having the largest possible set of hash functions should give us the best results. The disadvantage of using a large number of hash functions is its computational cost. *Pruning helps to achieve a reasonable precision with a much less number of the hash function than the total number of functions.*

Comparison with other hashing methods. Figure 5 compares our pruning algorithm with the following hashing methods on CIFAR and infinite MNIST datasets: ILH [12], Hashing with kernels (KSH) [5], KSHcut (FastHash) [7], Binary Reconstructive Embeddings (BRE) [4], Self-Taught Hashing (STH) [1], Spectral Hashing (SH) [2], Iterative Quantization (ITQ) [16], and Locality-Sensitive Hashing (LSH) [13]. We show results for $b = 16$ and 32 where ILH-prune has access to a pool of 200 1-bit hash functions. We can see that *pruning is very useful and performs well* for these number of bits. Methods like KSH and ITQ can beat ILH for small number of bits in these two datasets, but pruning improves the results of ILH significantly and becomes the best method.

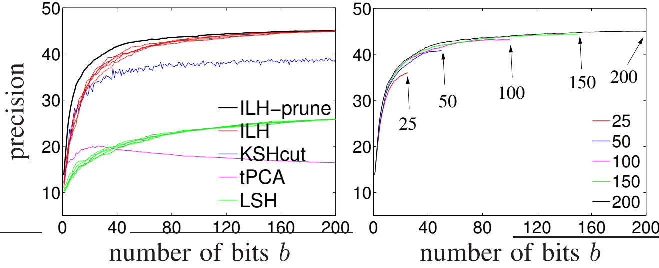

Figure 3. Precision as a function of number of bits in CIFAR dataset. *Left*: Comparing ILH-prune with other hashing methods. *Right*: Effect of increasing the number of 1-bit hash functions in pruning.

B. Learning local hash functions

Effect of the locality parameter on ILH-local. As explained in section IV, to train each of the hash functions, we consider a random point (seed) and its k-nearest neighbors as the training set, where k is the locality parameter. To see the effect of the locality parameter k, we first select 200 seeds randomly and fix them. Then, we change the size of the training subsets around the seeds and report precisions.

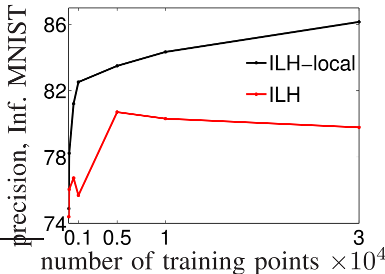

Figure 4. ILH-local vs ILH.

Fig. 4 shows the precision as a function of the size of the training subsets (and the locality parameter k). First, consider only the result of ILH-local. The figure illustrates that for a very small k, ILH-local performs poorly. This is because there is not much information in a very small subset of points and 1-bit hash functions perform like random projections. As we increase the number of points, the precision starts increasing. Here, we increase the number of points up to 30 000. It should be mentioned that at some point the precision starts decreasing as we increase the number of points because the hash functions start losing the diversity. Now, we compare the result of ILH-local with ILH. We see that ILH-local always performs better than ILH and increasing the size of training subsets makes the gap between them larger. This happens because *ILH-local keeps the diversity between the hash functions even when we use a large subset of points.*

Comparison with other hashing methods. We also compare ILH-local with several state-of-the art hashing methods in fig. 6 on infinite MNIST and Flickr datasets. We add these two unsupervised methods to the list of our competitors in the Flickr dataset: Binary Autoencoder (BA) [17] and thresholded PCA (tPCA). Locality shows its effectiveness as we increase the number of bits, so we report precision using $b = 128$ and 200. The figure shows that ILH-local performs better than all the other methods specifically using $b = 200$. This happens because for large b the 1-bit hash functions of all the methods (except ILH-local) will get similar to each other and lose the diversity.

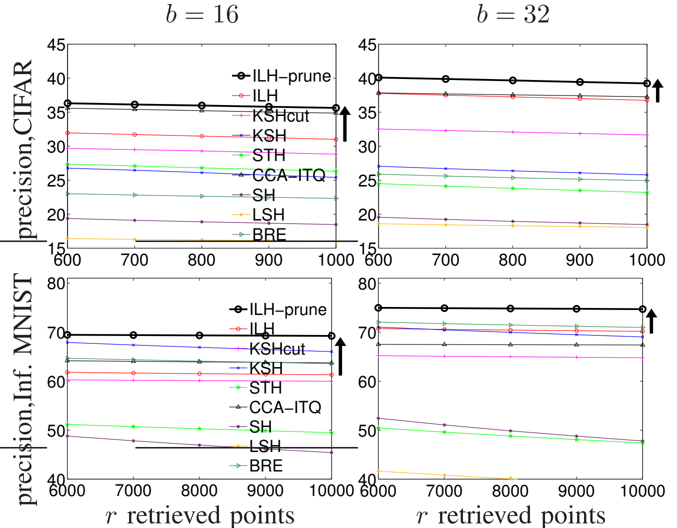

Figure 5. Comparing ILH-prune with binary hashing methods on CIFAR and infinite MNIST datasets. The arrow shows the improvement over ILH.

VI. Discussion

Optimizing affinity-based objective functions is difficult and slow. This is mainly because the objective function is nonsmooth: it contains a large number of binary variables (the bN binary codes) that are coupled and a larger number of pairwise terms ($\mathcal{O}(N^2)$). These difficulties limit the number of training points that can be used for learning the hash functions to a few thousand. FastHash [7] is the only exception; it uses the GraphCut algorithm to optimize the objective on a large training set with around 100 000 points.

Independent Laplacian Hashing [12], with the pruning and locality improvements proposed in this paper, makes supervised binary hashing scalable to much larger datasets. It can be trained easily on datasets with millions of points. The main reason is the ability of ILH to train independent hash functions on disjoint subsets of points. To see this better, consider the following example. Assume that FastHash is able to train b binary hash functions on a subset of N training points in t seconds. The average time for learning a 1-bit hash function of FastHash is the same as ILH (t/b seconds in this example). As a result, ILH, using b disjoint subsets, can train b hash functions in the same amount of time as FastHash *but using b times more training points than FastHash*. Also, ILH can learn the hash functions in parallel.

VII. Conclusion

The most common approach to learn supervised binary hash functions is to optimize an objective function that tries to preserve neighbors in Hamming space and make the 1-bit hash functions differ. This is a very difficult optimization that scales poorly to large datasets. A different approach that works as well or better is to train the 1-bit hash functions

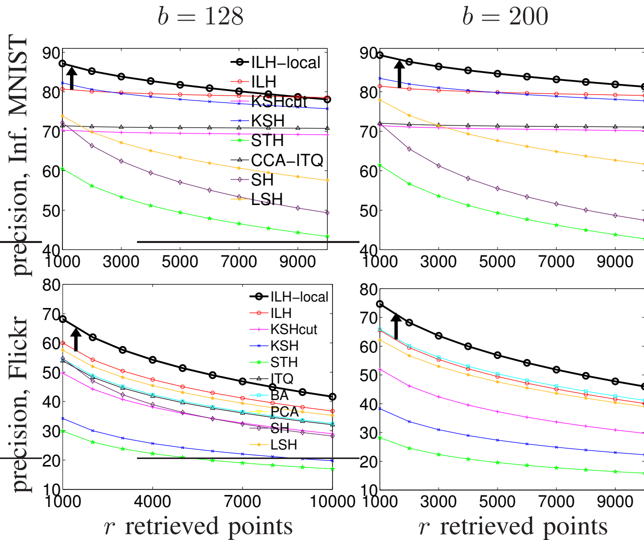

Figure 6. Comparing ILH-local with binary hashing methods on infinite MNIST and Flickr datasets. The arrow shows the improvement over ILH.

independently to preserve neighbors but make them diverse by training them on different data subsets, as done by ILH. We have proposed two methods to improve ILH even further. Our first method prunes redundant hash functions of ILH and achieves better retrieval results using a smaller number of bits. Our second method forces the hash functions of ILH to be spatially local, which leads to more diversity among the hash functions and better results. These methods allow us to learn binary hashing on millions of training points easily and achieve a considerably better retrieval.

VIII. Acknowledgments

Work supported by NSF award IIS–1423515.

References

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

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

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

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

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

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

[7] 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*, 2014.

[8] R. Raziperchikolaei and M. Á. Carreira-Perpiñán, "Optimizing affinity-based binary hashing using auxiliary coordinates," in *NIPS*, 2016.

[9] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *PAMI*, 2004.

[10] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *PAMI*, 2001.

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

[12] M. Á. Carreira-Perpiñán and R. Raziperchikolaei, "An ensemble diversity approach to supervised binary hashing," in *NIPS*, 2016.

[13] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Comm. ACM*, 2008.

[14] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing," *PAMI*, 2012.

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

[16] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A Procrustean approach to learning binary codes for large-scale image retrieval," *PAMI*, 2013.

[17] M. Á. Carreira-Perpiñán and R. Raziperchikolaei, "Hashing with binary autoencoders," in *CVPR*, 2015.

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

[19] C. Leng, J. Cheng, T. Yuan, X. Bai, and H. Lu, "Learning binary codes with bagging PCA," in *ECML*, 2014.

[20] L. I. Kuncheva, *Combining Pattern Classifiers: Methods and Algorithms*, John Wiley & Sons, 2014.

[21] D. D. Margineantu and T. G. Dietterich, "Pruning adaptive boosting," in *ICML*, 1997.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2009.

[23] D. Arthur and S. Vassilvitskii, "`k-means++`: The advantages of careful seeding," in *SODA 2007*, 2007.

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

[25] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *Int. J. Computer Vision*, 2001.

[26] G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," in *Large Scale Kernel Machines*, 2007.

[27] M. J. Huiskes, B. Thomee, and M. S. Lew, "New trends and ideas in visual concept detection: The MIR Flickr Retrieval Evaluation Initiative," in *Proc. ACM Int. Conf. Multimedia Information Retrieval*, 2010.

[28] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *J. Machine Learning Research*, 2008.