Hashing with Graphs

Wei Liu WLIU@EE.COLUMBIA.EDU
Department of Electrical Engineering, Columbia University, New York, NY 10027, USA

Jun Wang WANGJUN@US.IBM.COM
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA

Sanjiv Kumar SANJIVK@GOOGLE.COM
Google Research, New York, NY 10011, USA

Shih-Fu Chang SFCHANG@EE.COLUMBIA.EDU
Department of Electrical Engineering, Columbia University, New York, NY 10027, USA

Abstract

Hashing is becoming increasingly popular for efficient nearest neighbor search in massive databases. However, learning short codes that yield good search performance is still a challenge. Moreover, in many cases real-world data lives on a low-dimensional manifold, which should be taken into account to capture meaningful nearest neighbors. In this paper, we propose a novel graph-based hashing method which automatically discovers the neighborhood structure inherent in the data to learn appropriate compact codes. To make such an approach computationally feasible, we utilize *Anchor Graphs* to obtain tractable low-rank adjacency matrices. Our formulation allows constant time hashing of a new data point by extrapolating graph Laplacian eigenvectors to eigenfunctions. Finally, we describe a hierarchical threshold learning procedure in which each eigenfunction yields multiple bits, leading to higher search accuracy. Experimental comparison with the other state-of-the-art methods on two large datasets demonstrates the efficacy of the proposed method.

1. Introduction

Nearest neighbor (NN) search is a fundamental problem that arises commonly in computer vision, machine learning, data mining, and information retrieval. Conceptually, searching nearest neighbors of a query $\boldsymbol{q}$ requires scanning all n items in a database $\mathcal{X} = \{\boldsymbol{x}_i \in \mathbb{R}^d\}_{i=1}^n$, which has a linear time complexity $O(n)$. For large n, e.g., millions, exhaustive linear search is extremely expensive. Therefore, many techniques have been proposed in the past for fast approximate nearest neighbor (ANN) search. One classical paradigm to address this problem is based on trees, such as *kd*-tree (Friedman et al., 1977), which provides logarithmic query time $O(\log n)$. However, for high-dimensional data, most tree-based methods suffer significantly with their performance typically reducing to exhaustive linear search.

To overcome this issue, hashing-based methods have attracted considerable attention recently. These methods convert each database item into a code and can provide constant or sub-linear search time. In this paper, we focus on Hamming embeddings of data points, which map data to binary codes. The seminal work on *Locality-Sensitive Hashing* (LSH) (Gionis et al., 1999) uses simple random projections for such mapping. It has been extended to a variety of similarity measures including p-norm distances for $p \in (0, 2]$ (Datar et al., 2004), Mahalanobis distance (Kulis et al., 2009), and kernel similarity (Kulis & Grauman, 2009). Another related technique named *Shift Invariant Kernel Hashing* (SIKH) was proposed in (Raginsky & Lazebnik, 2010). Although enjoying asymptotic theoretical properties, LSH-related methods require long binary codes to achieve good precision. Nonetheless, long codes result in low recall when used for creating a hash lookup table, as the collision probability decreases exponentially with the code length. Hence, one usually needs

Appearing in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.

to set up multiple hash tables to achieve reasonable recall, which leads to longer query time as well as significant increase in storage.

Unlike the data-independent hash generation in LSH-based algorithms, more recent methods have focused on learning *data-dependent hash functions*. They try to learn compact binary codes for all database items, leading to faster query time with much less storage. Several methods such as *Restricted Boltzmann Machines* (RBMs) (or semantic hashing) (Salakhutdinov & Hinton, 2007), *Spectral Hashing* (SH) (Weiss et al., 2009), *Binary Reconstruction Embedding* (BRE) (Kulis & Darrell, 2010), and *Semi-Supervised Hashing* (SSH) (Wang et al., 2010a) have been proposed, but learning short codes that yield high search accuracy, especially in an *unsupervised* setting, is still an open question.

Perhaps the most critical shortcoming of the existing unsupervised hashing methods is the need to specify a *global* distance measure. On the contrary, in many real-world applications data lives on a low-dimensional manifold, which should be taken into account to capture meaningful nearest neighbors. For these, one can only specify *local* distance measures, while the global distances are automatically determined by the underlying manifold. In this work, we propose a graph-based hashing method which automatically discovers the neighborhood structure inherent in the data to learn appropriate compact codes in an unsupervised manner. Our basic idea is motivated by (Weiss et al., 2009) in which the goal is to embed the data in a Hamming space such that the neighbors in the original data space remain neighbors in the Hamming space.

Solving the above problem requires three main steps: (i) building a neighborhood graph using all n points from the database ($O(dn^2)$), (ii) computing r eigenvectors of the graph Laplacian ($O(rn)$), and (iii) extending r eigenvectors to any unseen data point ($O(rn)$). Unfortunately, step (i) is intractable for offline training while step (iii) is infeasible for online hashing given very large n. To avoid these bottlenecks, (Weiss et al., 2009) made a strong assumption that data is uniformly distributed. This leads to a simple analytical eigenfunction solution of 1-D Laplacians, but the manifold structure of the original data is almost ignored, substantially weakening the basic theme of that work.

On the contrary, in this paper, we propose a novel unsupervised hashing approach named *Anchor Graph Hashing* (AGH) to address both of the above bottlenecks. We build an approximate neighborhood graph using *Anchor Graphs* (Liu et al., 2010), in which the similarity between a pair of data points is measured with respect to a small number of anchors (typically a few hundred). The resulting graph is built in $O(n)$ time and is sufficiently sparse with performance approaching to the true kNN graph as the number of anchors increases. Because of the low-rank property of an Anchor Graph's adjacency matrix, our approach can solve the graph Laplacian eigenvectors in linear time. One critical requirement to make graph-based hashing practical is the ability to generate hash codes for unseen points. This is known as *out-of-sample extension* in the literature. In this work, we show that the eigenvectors of the Anchor Graph Laplacian can be extended to the generalized eigenfunctions in constant time, thus leading to fast code generation.

One interesting characteristic of the proposed hashing method AGH is that it tends to capture semantic neighborhoods. In other words, data points that are close in the Hamming space, produced by AGH, tend to share similar semantic labels. This is because for many real-world applications close-by points on a manifold tend to share similar labels, and AGH is derived using a neighborhood graph which reveals the underlying manifold, especially at large scale. The key characteristic of AGH is validated by extensive experiments carried out on two datasets, where AGH outperforms exhaustive linear scan in the input space with the commonly used ℓ_2 distance. In the remainder of this paper, we present AGH in Section 2, analyze it in Section 3, show experimental results in Section 4, and conclude the work in Section 5.

2. Anchor Graph Hashing (AGH)

2.1. Formulation

The goal in this paper is to learn binary codes such that neighbors in the input space are mapped to similar codes in the Hamming space. Suppose, $A_{ij} \geq 0$ is the similarity between a data pair $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ in the input space. Then, similar to Spectral Hashing (SH) (Weiss et al., 2009), our method seeks an r-bit Hamming embedding $Y \in \{1, -1\}^{n \times r}$ for n points in the database by minimizing[1]

$$\begin{aligned} &\min_{Y} \quad \frac{1}{2} \sum_{i,j=1}^{n} \|Y_i - Y_j\|^2 A_{ij} = \mathrm{tr}(Y^\top L Y) \\ &\text{s.t.} \quad Y \in \{1, -1\}^{n \times r},\ \mathbf{1}^\top Y = 0,\ Y^\top Y = nI_{r \times r} \quad (1) \end{aligned}$$

where Y_i is the i^{th} row of Y representing the r-bit code for point $\boldsymbol{x}_i$, A is the $n \times n$ similarity matrix, and $D = \mathrm{diag}(A\mathbf{1})$ with $\mathbf{1} = [1, \cdots, 1]^\top \in \mathbb{R}^n$. The graph

[1]Converting -1/1 codes to 0/1 codes is a trivial shift and scaling operation.

Laplacian is defined as $L = D - A$. The constraint $\mathbf{1}^\top Y = 0$ is imposed to maximize the information of each bit, which occurs when each bit leads to balanced partitioning of the data. Another constraint $Y^\top Y = nI_{r\times r}$ forces r bits to be mutually uncorrelated in order to minimize redundancy among bits.

The above problem is an integer program, equivalent to balanced graph partitioning even for a single bit. This is known to be NP-hard. To make eq. (1) tractable, one can apply *spectral relaxation* (Shi & Malik, 2000) to drop the integer constraint and allow $Y \in \mathbb{R}^{n\times r}$. With this, the solution Y is given by r eigenvectors of length $\sqrt{n}$ corresponding to r smallest eigenvalues (ignoring eigenvalue 0) of the graph Laplacian L. Y thereby forms an r-dimensional spectral embedding in analogy to *Laplacian Eigenmap* (Belkin & Niyogi, 2003). Note that the excluded bottom most eigenvector associated with eigenvalue 0 is $\mathbf{1}$ if the underlying graph is connected. Since all the remaining eigenvectors are orthogonal to it, $\mathbf{1}^\top Y = 0$ holds. An approximate solution given by $\mathrm{sgn}(Y)$ yields the final desired hash codes, forming a Hamming embedding from $\mathbb{R}^d$ to $\{1, -1\}^r$.

Although conceptually simple, the main bottleneck in the above formulation is computation. The cost of building the underlying graph and the associated Laplacian is $O(dn^2)$, which is intractable for large n. To avoid the computational bottleneck, unlike the restrictive assumption of a separable uniform data distribution made by SH, in this work, we propose a more general approach based on Anchor Graphs. The basic idea is to directly approximate the sparse neighborhood graph and the associated adjacency matrix as described next.

2.2. Anchor Graphs

An Anchor Graph uses a small set of m points called *anchors* to approximate the data neighborhood structure (Liu et al., 2010). Similarities of all n database points are measured with respect to these m anchors, and the true adjacency (or similarity) matrix A is approximated using these similarities. First, K-means clustering is performed on n data points to obtain m ($m \ll n$) cluster centers $\mathcal{U} = \{\boldsymbol{u}_j \in \mathbb{R}^d\}_{j=1}^m$ that act as anchor points. In practice, running K-means on a small subsample of the database with very few iterations (less than 10) is sufficient. This makes clustering very fast, thus speeding up training significantly[2]. Next, the Anchor Graph defines the *truncated* similarities Z_{ij}'s between all n data points and m anchors as,

$$Z_{ij} = \begin{cases} \dfrac{\exp(-\mathcal{D}^2(\boldsymbol{x}_i, \boldsymbol{u}_j)/t)}{\sum_{j' \in \langle i\rangle} \exp(-\mathcal{D}^2(\boldsymbol{x}_i, \boldsymbol{u}_{j'})/t)}, & \forall j \in \langle i\rangle \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where $\langle i\rangle \subset [1:m]$ denotes the indices of s ($s \ll m$) nearest anchors of point $\boldsymbol{x}_i$ in $\mathcal{U}$ according to a distance function $\mathcal{D}()$ such as ℓ_2 distance, and t denotes the bandwidth parameter. Note that the matrix $Z \in \mathbb{R}^{n\times m}$ is highly sparse. Each row Z_i contains only s nonzero entries which sum to 1.

Derived from random walks across data points and anchors, the Anchor Graph provides a powerful approximation to the adjacency matrix A as $\hat{A} = Z\Lambda^{-1}Z^\top$ where $\Lambda = \mathrm{diag}(Z^\top \mathbf{1}) \in \mathbb{R}^{m\times m}$ (Liu et al., 2010). The approximate adjacency matrix has three key properties: 1) $\hat{A}$ is nonnegative and sparse since Z is very sparse; 2) $\hat{A}$ is low-rank (its rank is at most m), so an Anchor Graph does not compute $\hat{A}$ explicitly but instead keeps its low-rank form; 3) $\hat{A}$ is a doubly stochastic matrix, i.e., has unit row and column sums, so the resulting graph Laplacian is $L = I - \hat{A}$. Properties 2) and 3) are critical, which allow efficient eigenfunction extensions of graph Laplacians, as shown in the next subsection. The memory cost of an Anchor Graph is $O(sn)$ for storing Z, and the time cost is $O(dmnT + dmn)$ in which $O(dmnT)$ originates from K-means clustering with T iterations. Since $m \ll n$, the cost for constructing an Anchor Graph is linear in n, which is far more efficient than constructing a kNN graph that has a quadratic cost $O(dn^2)$.

The graph Laplacian of the Anchor Graph is $L = I - \hat{A}$, so the required r graph Laplacian eigenvectors are also eigenvectors of $\hat{A}$ but associated with the r largest eigenvalues (ignoring eigenvalue 1 which corresponds to eigenvalue 0 of L). One can easily solve the eigenvectors of $\hat{A}$ by utilizing its low-rank property. Specifically, we solve the eigenvalue system of a small $m \times m$ matrix $M = \Lambda^{-1/2}Z^\top Z\Lambda^{-1/2}$, resulting in r ($< m$) eigenvector-eigenvalue pairs $\{(\boldsymbol{v}_k, \sigma_k)\}_{k=1}^r$ where $1 > \sigma_1 \geq \cdots \geq \sigma_r > 0$. After expressing $V = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_r] \in \mathbb{R}^{m\times r}$ (V is column-orthonormal) and $\Sigma = \mathrm{diag}(\sigma_1, \cdots, \sigma_r) \in \mathbb{R}^{r\times r}$, we obtain the desired spectral embedding matrix Y as

$$Y = \sqrt{n}Z\Lambda^{-1/2}V\Sigma^{-1/2} = ZW \tag{3}$$

which satisfies $\mathbf{1}^\top Y = 0$ and $Y^\top Y = nI_{r\times r}$. It is interesting to find out that hashing with Anchor Graphs can be interpreted as first nonlinearly transforming each input point $\boldsymbol{x}_i$ to Z_i by computing its sparse similarities to anchor points and second linearly projecting Z_i onto the vectors in $W = \sqrt{n}\Lambda^{-1/2}V\Sigma^{-1/2} = [\boldsymbol{w}_1, \cdots, \boldsymbol{w}_r] \in \mathbb{R}^{m\times r}$ where $\boldsymbol{w}_k = \sqrt{n/\sigma_k}\Lambda^{-1/2}\boldsymbol{v}_k$.

[2] Instead of K-means, one can alternatively use any other efficient clustering methods.

2.3. Eigenfunction Generalization

The procedure given in eq. (3) generates codes only for those points that are available during training. But, for the purpose of hashing, one needs to learn a general hash function $h : \mathbb{R}^d \mapsto \{1, -1\}$ which can take any arbitrary point as input. For this, one needs to generalize the eigenvectors of the Anchor Graph Laplacian to the eigenfunctions $\{\phi_k : \mathbb{R}^d \mapsto \mathbb{R}\}_{k=1}^r$ such that the hash functions can be simply defined as $h_k(\boldsymbol{x}) = \mathrm{sgn}(\phi_k(\boldsymbol{x}))$ $(k = 1, \cdots, r)$. We create the "out-of-sample" extension of the Anchor Graph Laplacian eigenvectors Y to their corresponding eigenfunctions using the Nyström method (Williams & Seeger, 2001)(Bengio et al., 2004). Theorem 1 below gives an analytical form to each eigenfunction ϕ_k.

Theorem 1. *Given m anchor points $\mathcal{U} = \{\boldsymbol{u}_j\}_{j=1}^m$ and any sample $\boldsymbol{x}$, define a feature map $\boldsymbol{z} : \mathbb{R}^d \mapsto \mathbb{R}^m$ as follows*

$$\boldsymbol{z}(\boldsymbol{x}) = \frac{\left[\delta_1 \exp(-\frac{\mathcal{D}^2(\boldsymbol{x},\boldsymbol{u}_1)}{t}), \cdots, \delta_m \exp(-\frac{\mathcal{D}^2(\boldsymbol{x},\boldsymbol{u}_m)}{t})\right]^\top}{\sum_{j=1}^m \delta_j \exp(-\frac{\mathcal{D}^2(\boldsymbol{x},\boldsymbol{u}_j)}{t})}, \tag{4}$$

where $\delta_j \in \{1, 0\}$ and $\delta_j = 1$ if and only if anchor $\boldsymbol{u}_j$ is one of s nearest anchors of sample $\boldsymbol{x}$ in $\mathcal{U}$ according to the distance function $\mathcal{D}()$. Then the Nyström eigenfunction extended from the Anchor Graph Laplacian eigenvector $\boldsymbol{y}_k = Z\boldsymbol{w}_k$ is

$$\phi_k(\boldsymbol{x}) = \boldsymbol{w}_k^\top \boldsymbol{z}(\boldsymbol{x}). \tag{5}$$

Proof. First, we check that ϕ_k and $\boldsymbol{y}_k$ overlap on all training samples. If $\boldsymbol{x}_i$ is in the training set, then $Z_i^\top = \boldsymbol{z}(\boldsymbol{x}_i)$ and thus $\phi_k(\boldsymbol{x}_i) = \boldsymbol{w}_k^\top Z_i^\top = Z_i \boldsymbol{w}_k = Y_{ik}$.

The Anchor Graph's adjacency matrix $\hat{A} = Z\Lambda^{-1}Z^\top$ is positive semidefinite, with each entry defined as $\hat{A}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{z}^\top(\boldsymbol{x}_i)\Lambda^{-1}\boldsymbol{z}(\boldsymbol{x}_j)$. For any unseen sample $\boldsymbol{x}$, the Nyström method extends $\boldsymbol{y}_k$ to $\phi_k(\boldsymbol{x})$ as the weighted summation over n entries of $\boldsymbol{y}_k$: $\phi_k(\boldsymbol{x}) = \sum_{i=1}^n \hat{A}(\boldsymbol{x}, \boldsymbol{x}_i) Y_{ik}/\sigma_k$. Since $\boldsymbol{w}_k = \sqrt{n/\sigma_k}\Lambda^{-1/2}\boldsymbol{v}_k$ and $M\boldsymbol{v}_k = \sigma_k \boldsymbol{v}_k$, we can show that

$$\begin{aligned}
\phi_k(\boldsymbol{x}) &= \frac{1}{\sigma_k}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1}[\boldsymbol{z}(\boldsymbol{x}_1), \cdots, \boldsymbol{z}(\boldsymbol{x}_n)]\boldsymbol{y}_k \\
&= \frac{1}{\sigma_k}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1}Z^\top\boldsymbol{y}_k = \frac{1}{\sigma_k}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1}Z^\top Z\boldsymbol{w}_k \\
&= \frac{1}{\sigma_k}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1}Z^\top Z\sqrt{\frac{n}{\sigma_k}}\Lambda^{-1/2}\boldsymbol{v}_k \\
&= \sqrt{\frac{n}{\sigma_k^3}}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1/2}\left(\Lambda^{-1/2}Z^\top Z\Lambda^{-1/2}\boldsymbol{v}_k\right) \\
&= \sqrt{\frac{n}{\sigma_k^3}}\boldsymbol{z}^\top(\boldsymbol{x})\Lambda^{-1/2}\left(M\boldsymbol{v}_k\right) \\
&= \boldsymbol{z}^\top(\boldsymbol{x})\left(\sqrt{\frac{n}{\sigma_k}}\Lambda^{-1/2}\boldsymbol{v}_k\right) = \boldsymbol{z}^\top(\boldsymbol{x})\boldsymbol{w}_k = \boldsymbol{w}_k^\top\boldsymbol{z}(\boldsymbol{x}).
\end{aligned}$$

□

Following Theorem 1, the hash functions used in the proposed Anchor Graph Hashing (AGH) are designed as:

$$h_k(\boldsymbol{x}) = \mathrm{sgn}(\boldsymbol{w}_k^\top \boldsymbol{z}(\boldsymbol{x})), \ k = 1, \cdots, r. \tag{6}$$

In addition to the time for Anchor Graph construction, AGH needs $O(m^2n + srn)$ time for solving r graph Laplacian eigenvectors retained in the spectral embedding matrix Y, and $O(rn)$ time for compressing Y into binary codes. Under the online search scenario, AGH needs to save the binary codes $\mathrm{sgn}(Y)$ of n training samples, m anchors $\mathcal{U}$, and the projection matrix W in memory. Hashing any test sample $\boldsymbol{x}$ only costs $O(dm + sr)$ time which is dominated by the construction of a sparse vector $\boldsymbol{z}(\boldsymbol{x})$.

Remarks. 1) Though the graph Laplacian eigenvectors of the Anchor Graph are not as accurate as those of an exact neighborhood graph, e.g., kNN graph, they provide good performance when used for hashing. Exact neighborhood graph construction is infeasible at large scale. Even if one could get r graph Laplacian eigenvectors of the exact graph, the cost of calculating their Nyström extensions to a novel sample is $O(rn)$, which is still infeasible for online hashing requirement. 2) Free from any restrictive data distribution assumption, AGH solves Anchor Graph Laplacian eigenvectors in linear time and extends them to eigenfunctions in constant time (depends only on constants m and s).

2.4. Hierarchical Hashing

To generate r-bit codes, we use r graph Laplacian eigenvectors, but not all eigenvectors are equally suitable for hashing especially when r increases. From a geometric point of view, the intrinsic dimension of data manifolds is usually low, so a low-dimensional spectral embedding containing the lower graph Laplacian eigenvectors is desirable. Moreover, (Shi & Malik, 2000) discussed that the error made in converting the real-valued eigenvector $\boldsymbol{y}_k$ to the optimal integer solution $\boldsymbol{y}_k^* \in \{1, -1\}^n$ accumulates rapidly as k increases. In this subsection, we propose a simple hierarchical scheme that gives the priority to the lower graph Laplacian eigenvectors by revisiting them to generate multiple bits.

To illustrate the basic idea, let us look at a toy example shown in Fig. 1. To generate the first bit, the graph Laplacian eigenvector $\boldsymbol{y}$ partitions the graph by the red

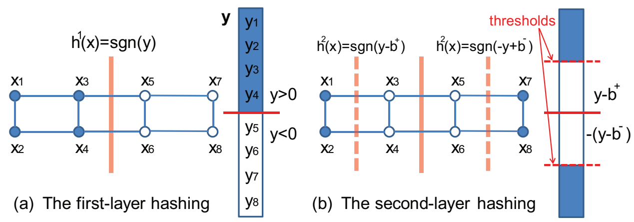

Figure 1. Hierarchical hashing on a data graph. $x_1, \cdots, x_8$ are data points and $\boldsymbol{y}$ is a graph Laplacian eigenvector. The data points of filled circles take '1' hash bit and the others take '-1' hash bit. The entries with dark color in $\boldsymbol{y}$ are positive and the others are negative. (a) The first-layer hash function h^1 uses threshold 0 ; (b) the second-layer hash functions h^2 use thresholds b^+ and b^-.

line using threshold zero. Due to thresholding, there is always a possibility that neighboring points close to the boundary (i.e., threshold) are hashed to different bits (e.g., points x_3 and x_5). To address this issue, we conduct hierarchical hashing of two layers in which the second-layer hashing tries to correct the boundary errors caused by the previous hashing. Intuitively, we form the second layer by further dividing each partition created by the first layer. In other words, the positive and negative entries in $\boldsymbol{y}$ are thresholded at b^+ and b^-, respectively. Hence, the hash bits at the second layer are generated by $\mathrm{sgn}(y_i - b^+)$ when $y_i > 0$ and $\mathrm{sgn}(-y_i + b^-)$ otherwise. Fig. 1(b) shows that x_3 and x_5 are hashed to the same bit at the second layer. Next we describe how one can learn the optimal thresholds for the second-layer hashing.

We propose to optimize the two thresholds b^+ and b^- from the perspective of balanced graph partitioning. Let us form a thresholded vector $\begin{bmatrix} \boldsymbol{y}^+ - b^+\mathbf{1}^+ \\ -\boldsymbol{y}^- + b^-\mathbf{1}^- \end{bmatrix}$ whose sign gives a hash bit for each training sample during the second-layer hashing. Two vectors $\boldsymbol{y}^+$ of length n^+ and $\boldsymbol{y}^-$ of length n^- correspond to the positive and negative entries in $\boldsymbol{y}$, respectively. Two constant vectors $\mathbf{1}^+$ and $\mathbf{1}^-$ contain n^+ and n^- 1 entries accordingly ($n^+ + n^- = n$). Similar to the first layer, we would like to find such thresholds that minimize the cut value of the graph Laplacian with the target thresholded vector while maintaining a balanced partitioning, i.e.,

$$\min_{b^+,b^-} \Gamma(b^+,b^-) = \begin{bmatrix} \boldsymbol{y}^+ - b^+\mathbf{1}^+ \\ -\boldsymbol{y}^- + b^-\mathbf{1}^- \end{bmatrix}^\top L \begin{bmatrix} \boldsymbol{y}^+ - b^+\mathbf{1}^+ \\ -\boldsymbol{y}^- + b^-\mathbf{1}^- \end{bmatrix}$$
$$\text{s.t. } \mathbf{1}^\top \begin{bmatrix} \boldsymbol{y}^+ - b^+\mathbf{1}^+ \\ -\boldsymbol{y}^- + b^-\mathbf{1}^- \end{bmatrix} = 0. \tag{7}$$

Defining vector $\overline{\boldsymbol{y}} = [(\boldsymbol{y}^+)^\top, -(\boldsymbol{y}^-)^\top]^\top$ and arranging L into $\begin{bmatrix} L_{+.} \\ L_{-.} \end{bmatrix} = \begin{bmatrix} L_{++} & L_{+-} \\ L_{-+} & L_{--} \end{bmatrix}$ corresponding to the positive and negative entries in $\boldsymbol{y}$, we optimize b^+ and b^- by zeroing the derivatives of the objective in eq. (7). After simple algebraic manipulation, one can show that

$$b^+ + b^- = \frac{(\mathbf{1}^+)^\top L_{+.}\overline{\boldsymbol{y}}}{(\mathbf{1}^+)^\top L_{++}\mathbf{1}^+} \equiv \beta. \tag{8}$$

On the other hand, combining the fact that $\mathbf{1}^\top \boldsymbol{y} = 0$ with the constraint in eq. (7) leads to:

$$n^+b^+ - (n - n^+)b^- = (\mathbf{1}^+)^\top \boldsymbol{y}^+ - (\mathbf{1}^-)^\top \boldsymbol{y}^- = 2(\mathbf{1}^+)^\top \boldsymbol{y}^+. \tag{9}$$

We use the Anchor Graph's adjacency matrix $\hat{A} = Z\Lambda^{-1}Z^\top$ for the computations involving the graph Laplacian L. Suppose, $\boldsymbol{y}$ is an eigenvector of $\hat{A}$ with eigenvalue σ such that $\hat{A}\boldsymbol{y} = \sigma\boldsymbol{y}$. Then, we have $\hat{A}_{++}\boldsymbol{y}^+ + \hat{A}_{+-}\boldsymbol{y}^- = \sigma\boldsymbol{y}^+$. Thus, from eq. (8),

$$\begin{aligned} \beta = \frac{(\mathbf{1}^+)^\top L_{+.}\overline{\boldsymbol{y}}}{(\mathbf{1}^+)^\top L_{++}\mathbf{1}^+} &= \frac{(\mathbf{1}^+)^\top \left((I - \hat{A}_{++})\boldsymbol{y}^+ + \hat{A}_{+-}\boldsymbol{y}^-\right)}{(\mathbf{1}^+)^\top (I - \hat{A}_{++})\mathbf{1}^+} \\ &= \frac{(\mathbf{1}^+)^\top (\boldsymbol{y}^+ - \hat{A}_{++}\boldsymbol{y}^+ + \sigma\boldsymbol{y}^+ - \hat{A}_{++}\boldsymbol{y}^+)}{n^+ - (\mathbf{1}^+)^\top \hat{A}_{++}\mathbf{1}^+} \\ &= \frac{(\sigma+1)(\mathbf{1}^+)^\top \boldsymbol{y}^+ - 2(\mathbf{1}^+)^\top \hat{A}_{++}\boldsymbol{y}^+}{n^+ - (\mathbf{1}^+)^\top \hat{A}_{++}\mathbf{1}^+} \\ &= \frac{(\sigma+1)(\mathbf{1}^+)^\top \boldsymbol{y}^+ - 2(Z_+^\top \mathbf{1}^+)^\top \Lambda^{-1}(Z_+^\top \boldsymbol{y}^+)}{n^+ - (Z_+^\top \mathbf{1}^+)^\top \Lambda^{-1}(Z_+^\top \mathbf{1}^+)}, \end{aligned} \tag{10}$$

where $Z_+ \in \mathbb{R}^{n^+ \times m}$ is the sub-matrix of $Z = \begin{bmatrix} Z_+ \\ Z_- \end{bmatrix}$ corresponding to $\boldsymbol{y}^+$. By putting eq. (8)-(10) together, we solve the target thresholds as

$$\begin{cases} b^+ = \dfrac{2(\mathbf{1}^+)^\top \boldsymbol{y}^+ + (n - n^+)\beta}{n} \\ b^- = \dfrac{-2(\mathbf{1}^+)^\top \boldsymbol{y}^+ + n^+\beta}{n}, \end{cases} \tag{11}$$

which requires $O(mn^+)$ time.

Now we give the two-layer hash functions for AGH to yield an r-bit code using the first $r/2$ graph Laplacian eigenvectors of the Anchor Graph. Conditioned on the outputs of the first-layer hash functions $\{h_k^{(1)}(\boldsymbol{x}) = \mathrm{sgn}\left(\boldsymbol{w}_k^\top \boldsymbol{z}(\boldsymbol{x})\right)\}_{k=1}^{r/2}$, the second-layer hash functions are generated dynamically as follows for $k = 1, \cdots, r/2$,

$$h_k^{(2)}(\boldsymbol{x}) = \begin{cases} \mathrm{sgn}\left(\boldsymbol{w}_k^\top \boldsymbol{z}(\boldsymbol{x}) - b_k^+\right) & \text{if} \quad h_k^{(1)}(\boldsymbol{x}) = 1 \\ \mathrm{sgn}\left(-\boldsymbol{w}_k^\top \boldsymbol{z}(\boldsymbol{x}) + b_k^-\right) & \text{if} \quad h_k^{(1)}(\boldsymbol{x}) = -1 \end{cases} \tag{12}$$

in which (b_k^+, b_k^-) are calculated from each eigenvector $\boldsymbol{y}_k = Z\boldsymbol{w}_k$. Compared to r one-layer hash functions $\{h_k^{(1)}\}_{k=1}^r$, the proposed two-layer hash functions for r bits actually use the $r/2$ lower eigenvectors twice. Hence, they avoid using the higher eigenvectors which can potentially be of low quality for partitioning and hashing. The experiments conducted in Section 4 reveal that with the same number of bits, AGH using two-layer hash functions achieves comparable precision but much higher recall than using one-layer hash functions alone (see Fig. 2(c)(d)). Of course, one can extend hierarchical hashing to more than two layers. However, the accuracy of the resulting hash functions will depend on whether repeatedly partitioning the existing eigenvectors gives more informative bits than those from picking new eigenvectors.

3. Analysis

For the same budget of r bits, we analyze two hashing algorithms which are proposed in Section 2 and both based on Anchor Graphs with the fixed construction parameters m and s. For convenience, we name AGH with r one-layer hash functions $\{h_k^{(1)}\}_{k=1}^r$ *1-AGH*, and AGH with r two-layer hash functions $\{h_k^{(1)}, h_k^{(2)}\}_{k=1}^{r/2}$ *2-AGH*, respectively.

Below we give space and time complexities of 1-AGH and 2-AGH.

`Space Complexity:` $O((d+s+r)n)$ in the training phase and $O(rn)$ (binary bits) in the test phase for both of 1-AGH and 2-AGH.

`Time Complexity:` $O(dmnT+dmn+m^2n+(s+1)rn)$ for 1-AGH and $O(dmnT+dmn+m^2n+(s/2+m/2+1)rn)$ for 2-AGH in the training phase; $O(dm+sr)$ for both in the test phase.

To summarize, 1-AGH and 2-AGH both have linear training time and constant query time.

4. Experimental Results

4.1. Methods and Evaluation Protocols

We evaluate the proposed graph-based unsupervised hashing, both single-layer AGH (1-AGH) and two-layer AGH (2-AGH), on two benchmark datasets: **MNIST** (70K) and **NUS-WIDE** (270K). Their performance is compared against other popular unsupervised hashing methods including Locality-Sensitive Hashing (LSH), PCA Hashing (PCAH), Unsupervised Sequential Projection Learning for Hashing (USPLH) (Wang et al., 2010a), Spectral Hashing (SH), Kernelized Locality-Sensitive Hashing (KLSH), and Shift-Invariant Kernel Hashing (SIKH). These methods cover both linear (LSH, PCAH and USPLH) and nonlinear (SH, KLSH and SIKH) hashing paradigms. Our AGH methods are nonlinear. We also compare against a supervised hashing method BRE which is trained by sampling a few similar and dissimilar data pairs. We sample 1,000 training points from each dataset, and for each point use ℓ_2 distance to find its top/bottom 2% NNs as similar/dissimilar pairs on **MNIST** and its top/bottom 1% NNs as similar/dissimilar pairs on **NUS-WIDE**, respectively. To run KLSH, we sample 300 training points to form the empirical kernel map and use the same Gaussian kernel as for SIKH. To run our methods 1-AGH and 2-AGH, we fix the graph construction parameters to $m = 300, s = 2$ on **MNIST** and $m = 300, s = 5$ on **NUS-WIDE**, respectively. We adopt ℓ_2 distance for the distance function $\mathcal{D}()$ in defining the matrix Z. In addition, we run K-means clustering with $T = 5$ iterations to find anchors on each dataset. All our experiments are run on a workstation with 2.53 GHz Intel Xeon CPU and 10GB RAM.

We follow two search procedures, i.e., hash lookup and Hamming ranking, for consistent evaluations across two datasets. Hash lookup emphasizes more on search speed since it has constant query time. However, when using many hash bits and a single hash table, hash lookup often fails because the Hamming space becomes increasingly sparse and very few samples fall in the same hash bucket. Hence, similar to (Weiss et al., 2009), we search within a Hamming radius 2 to retrieve potential neighbors for each query. Hamming ranking measures the search quality by ranking database points according to their Hamming distances to the query. Even though the complexity of Hamming ranking is linear, it is usually very fast in practice.

4.2. Datasets

The well-known **MNIST** dataset[3] consists of 784-dimensional 70,000 samples associated with digits from '0' to '9'. We split this dataset into two subsets: a training set containing 69,000 samples and a query set of 1,000 samples. Because this dataset is fully annotated, we define true neighbors as semantic neighbors based on the associated digit labels.

The second dataset **NUS-WIDE**[4] contains around 270,000 web images associated with 81 ground truth concept tags. Each image is represented by an ℓ_2 normalized 1024-dimensional sparse-coding feature vector (Wang et al., 2010b). Unlike **MNIST**, each image in

[3] http://yann.lecun.com/exdb/mnist/

[4] http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm

Table 1. Hamming ranking performance on **MNIST** and **NUS-WIDE**. r denotes the number of hash bits used in hashing algorithms, and also the number of eigenfunctions used in SE ℓ_2 linear scan. The K-means execution time is 20.1 sec and 105.5 sec for training AGH on **MNIST** and **NUS-WIDE**, respectively. All training and test time is recorded in sec.

Method	**MNIST (70K)**				**NUS-WIDE (270K)**			
	MAP		**Train Time**	**Test Time**	**MP**		**Train Time**	**Test Time**
	$r = 24$	$r = 48$	$r = 48$	$r = 48$	$r = 24$	$r = 48$	$r = 48$	$r = 48$
ℓ_2 Scan	0.4125		–		0.4523		–	
SE ℓ_2 Scan	0.5269	0.3909	–	–	0.4866	0.4775	–	–
LSH	0.1613	0.2196	1.8	2.1×10^{-5}	0.3196	0.2844	8.5	1.0×10^{-5}
PCAH	0.2596	0.2242	4.5	2.2×10^{-5}	0.3643	0.3450	18.8	1.3×10^{-5}
USPLH	0.4699	**0.4930**	163.2	2.3×10^{-5}	0.4269	0.4322	834.7	1.3×10^{-5}
SH	0.2699	0.2453	4.9	4.9×10^{-5}	0.3609	0.3420	25.1	4.1×10^{-5}
KLSH	0.2555	0.3049	2.9	5.3×10^{-5}	0.4232	0.4157	8.7	4.9×10^{-5}
SIKH	0.1947	0.1972	0.4	1.3×10^{-5}	0.3270	0.3094	2.0	1.1×10^{-5}
1-AGH	**0.4997**	0.3971	22.9	5.3×10^{-5}	**0.4762**	**0.4761**	115.2	4.4×10^{-5}
2-AGH	**0.6738**	**0.6410**	23.2	6.5×10^{-5}	**0.4699**	**0.4779**	118.1	5.3×10^{-5}
BRE	0.2638	0.3090	57.9	6.7×10^{-5}	0.4100	0.4229	1247.4	8.3×10^{-5}

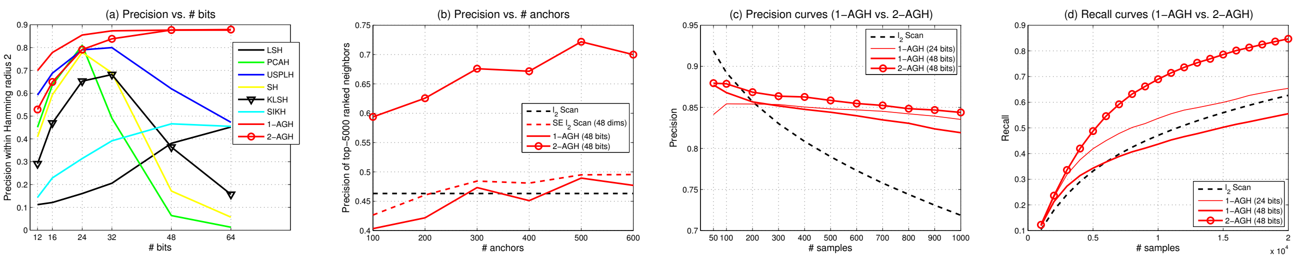

Figure 2. Results on **MNIST**. (a) Precision within Hamming radius 2 using hash lookup and the varying number of hash bits (r); (b) Hamming ranking precision of top-5000 ranked neighbors using the varying number of anchors (m); (c) Hamming ranking precision curves; (d) Hamming ranking recall curves.

NUS-WIDE contains multiple semantic labels (tags). The true neighbors are defined based on whether two images share *at least one* common tag. For evaluation, we consider 21 most frequent tags, such as 'animal', 'buildings', 'person', etc., each of which has abundant relevant images ranging from 5,000 to 30,000. We sample uniformly 100 images from each of the selected 21 tags to form a query set of 2,100 images with the rest serving as the training set.

4.3. Results

Table 1 shows the Hamming ranking performance measured by Mean Average Precision (MAP), training time, and test time for different hashing methods on **MNIST**. We also report MAP for ℓ_2 linear scan in the original input space and ℓ_2 linear scan in the spectral embedding (SE) space, namely SE ℓ_2 linear scan whose binary version is 1-AGH. From this table it is clear that SE ℓ_2 scan gives better precision than ℓ_2 scan for $r = 24$. This shows that spectral embedding is capturing the semantic neighborhoods by learning the intrinsic manifold structure of the data. Increasing r leads to poorer MAP performance, indicating the intrinsic manifold dimension to be around 24. 2-AGH performs significantly better than the other hashing methods and even better than ℓ_2 linear scan and SE ℓ_2 linear scan. Note that the results from both ℓ_2 and SE ℓ_2 linear scans are provided to show the advantage of taking the manifold view in AGH. Such linear scans are not scalable NN search methods.

In terms of training time, while 1-AGH and 2-AGH need more time than the most hashing methods, they are faster than USPLH and BRE. Most of the training time in AGH is spent on the K-means step. By using a subsampled dataset, instead of the whole database, one can further speed up K-means significantly. The test time of AGH methods is comparable to the other nonlinear hashing methods. Table 1 shows a similar trend on the **NUS-WIDE** dataset. As computing MAP is slow on this larger dataset, we show Mean Precision (MP) of top-5000 returned neighbors.

Fig. 2(a) and Fig. 3(a) show the precision curves using hash lookup within Hamming radius 2. Due to increased sparsity of the Hamming space with more bits, precision for the most hashing methods drops significantly when longer codes are used. However, both 1-AGH and 2-AGH do not suffer from this common drawback and provide higher precision when using more than 24 bits for both datasets. We also

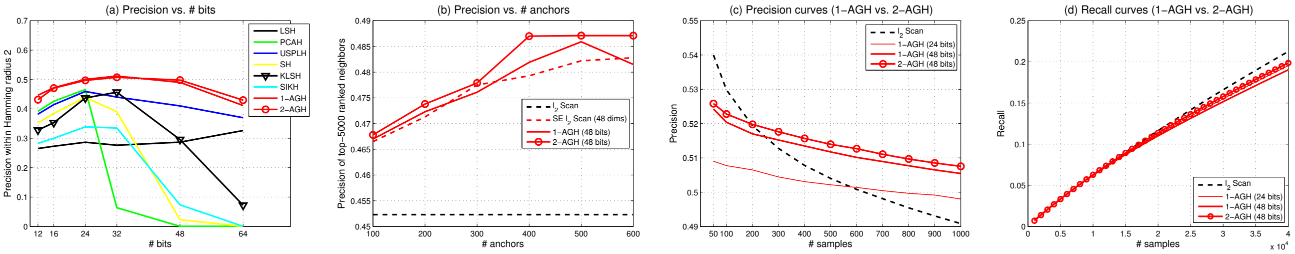

Figure 3. Results on **NUS-WIDE**. (a) Hash lookup precision within Hamming radius 2 using the varying number of hash bits (r); (b) Hamming ranking precision of top-5000 ranked neighbors using the varying number of anchors (m); (c) Hamming ranking precision curves; (d) Hamming ranking recall curves.

plot the Hamming ranking precision of top-5000 returned neighbors with an increasing number of anchors ($100 \leq m \leq 600$) in Fig. 2(b) and Fig. 3(b) (except these two, all the results are reported under $m = 300$), from which one can observe that 2-AGH consistently provides superior precision performance compared to ℓ_2 linear scan, SE ℓ_2 linear scan, and 1-AGH. The gains are more significant on **MNIST**.

Finally, overall better performance of 2-AGH over 1-AGH implies that the higher eigenfunctions of the Anchor Graph Laplacian are not as good as the lower ones when used to create hash bits. 2-AGH reuses the lower eigenfunctions and gives higher search accuracy (see Fig. 2(c)(d) and Fig. 3(c)(d)).

5. Conclusion

We have proposed a scalable graph-based unsupervised hashing approach which respects the underlying manifold structure of the data to return meaningful nearest neighbors. We further showed that Anchor Graphs can overcome the computationally prohibitive step of building graph Laplacians by approximating the adjacency matrix with a low-rank matrix. The hash functions are learned by thresholding the lower eigenfunctions of the Anchor Graph Laplacian in a hierarchical fashion. Experimental comparison showed significant performance gains over the state-of-the-art hashing methods in retrieving semantically similar neighbors. In the future, we would like to investigate if any theoretical guarantees could be provided on retrieval accuracy of our approach.

Acknowledgments

This work is supported in part by NSF Awards # CNS-07-16203, # CNS-07-51078, and Office of Naval Research.

References

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

Bengio, Y., Delalleau, O., Roux, N. Le, Paiement, J.-F., Vincent, P., and Ouimet, M. Learning eigenfunctions links spectral embedding and kernel pca. *Neural Computation*, 16(10):2197–2219, 2004.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, 2004.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Mathematical Software*, 3(3):209–226, 1977.

Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proc. VLDB*, 1999.

Kulis, B. and Darrell, T. Learning to hash with binary reconstructive embeddings. In *NIPS 22*, 2010.

Kulis, B. and Grauman, K. Kernelized locality-sensitive hashing for scalable image search. In *Proc. ICCV*, 2009.

Kulis, B., Jain, P., and Grauman, K. Fast similarity search for learned metrics. *IEEE Trans. on PAMI*, 31(12):2143–2157, 2009.

Liu, W., He, J., and Chang, S.-F. Large graph construction for scalable semi-supervised learning. In *Proc. ICML*, 2010.

Raginsky, M. and Lazebnik, S. Locality-sensitive binary codes from shift-invariant kernels. In *NIPS 22*, 2010.

Salakhutdinov, R. R. and Hinton, G. E. Learning a nonlinear embedding by preserving class neighbourhood structure. In *Proc. AISTATS*, 2007.

Shi, J. and Malik, J. Normalized cuts and image segmentation. *IEEE Trans. on PAMI*, 22(8):888–905, 2000.

Wang, J., Kumar, S., and Chang, S.-F. Sequential projection learning for hashing with compact codes. In *Proc. ICML*, 2010a.

Wang, J., Yang, J., Yu, K., Lv, F., Huang, T., and Gong, Y. Locality-constrained linear coding for image classification. In *Proc. CVPR*, 2010b.

Weiss, Y., Torralba, A., and Fergus, R. Spectral hashing. In *NIPS 21*, 2009.

Williams, C. K. I. and Seeger, M. Using the nyström method to speed up kernel machines. In *NIPS 13*, 2001.