Auto-Encoding Twin-Bottleneck Hashing

Yuming Shen∗†, Jie Qin∗†, Jiaxin Chen∗, Mengyang Yu, Li Liu, Fan Zhu, Fumin Shen, and Ling Shao

1Inception Institute of Artificial Intelligence (IIAI), Abu Dhabi, UAE
2Center for Future Media, University of Electronic Science and Technology of China, China

ymcidence@gmail.com

Abstract

Conventional unsupervised hashing methods usually take advantage of similarity graphs, which are either pre-computed in the high-dimensional space or obtained from random anchor points. On the one hand, existing methods uncouple the procedures of hash function learning and graph construction. On the other hand, graphs empirically built upon original data could introduce biased prior knowledge of data relevance, leading to sub-optimal retrieval performance. In this paper, we tackle the above problems by proposing an efficient and adaptive code-driven graph, which is updated by decoding in the context of an auto-encoder. Specifically, we introduce into our framework twin bottlenecks (i.e., latent variables) that exchange crucial information collaboratively. One bottleneck (i.e., binary codes) conveys the high-level intrinsic data structure captured by the code-driven graph to the other (i.e., continuous variables for low-level detail information), which in turn propagates the updated network feedback for the encoder to learn more discriminative binary codes. The auto-encoding learning objective literally rewards the code-driven graph to learn an optimal encoder. Moreover, the proposed model can be simply optimized by gradient descent without violating the binary constraints. Experiments on benchmarked datasets clearly show the superiority of our framework over the state-of-the-art hashing methods. Our source code can be found at https://github.com/ymcidence/TBH.

1. Introduction

Approximate Nearest Neighbour (ANN) search has attracted ever-increasing attention in the era of big data. Thanks to the extremely low costs for computing Hamming distances, binary coding/hashing has been appreciated as an efficient solution to ANN search. Similar to other feature learning schemes, hashing techniques can be typically subdivided into supervised and unsupervised ones. Supervised hashing [11, 23, 25, 39, 45, 48], which highly depends on labels, is not always preferable since large-scale data annotations are unaffordable. Conversely, unsupervised hashing [12, 16, 15, 20, 33, 47, 46], provides a cost-effective solution for more practical applications. To exploit data similarities, existing unsupervised hashing methods [29, 30, 43, 35, 41] have extensively employed graph-based paradigms. Nevertheless, existing methods usually suffer from the ‘static graph’ problem. More concretely, they often adopt explicitly pre-computed graphs, introducing biased prior knowledge of data relevance. Besides, graphs cannot be adaptively updated to better model the data structure. The interaction between hash function learning and graph construction can be hardly established. The ‘static graph’ problem greatly hinders the effectiveness of graph-based unsupervised hashing mechanisms.

In this work, we tackle the above long-standing chal-

*Equal Contribution
†Corresponding Author
lenge by proposing a novel adaptive graph, which is directly driven by the learned binary codes. The graph is then seamlessly embedded into a generative network that has recently been verified effective for learning reconstructive binary codes [8, 10, 40, 50]. In general, our network can be regarded as a variant of Wasserstein Auto-Encoder [42] with two kinds of bottlenecks (i.e., latent variables). Hence, we call the proposed method Twin-Bottleneck Hashing (TBH). Fig. 1 illustrates the differences between TBH and the related models. As shown in Fig. 1 (c), the binary bottleneck (BinBN) contributes to constructing the code-driven similarity graph, while the continuous bottleneck (ConBN) mainly guarantees the reconstruction quality. Furthermore, Graph Convolutional Networks (GCNs) [19] are leveraged as a ‘tunnel’ for the graph and the ConBN to fully exploit data relationships. As a result, similarity-preserving latent representations are fed into the decoder for high-quality reconstruction. Finally, as a reward, the updated network setting is back-propagated through the ConBN to the encoder, which can better fulfill our ultimate goal of binary coding.

More concretely, TBH tackles the ‘static graph’ problem by directly leveraging the latent binary codes to adaptively capture the intrinsic data structure. To this end, an adaptive similarity graph is computed directly based on the Hamming distances between binary codes, and is used to guide the ConBN through neural graph convolution [14, 19]. This design provides an optimal mechanism for efficient retrieval tasks by directly incorporating the Hamming distance into training. On the other hand, as a side benefit of the twin-bottleneck module, TBH can also overcome another important limitation in generative hashing models [5, 8], i.e., directly inputting the BinBN to the decoder leads to poor data reconstruction capability. For simplicity, we call this problem as ‘deficient BinBN’. Particularly, we address this problem by leveraging the ConBN, which is believed to have higher encoding capacity, for decoding. In this way, one can expect these continuous latent variables to preserve more entropy than binary ones. Consequently, the reconstruction procedure in the generative model becomes more effective.

In addition, during the optimization procedure, existing hashing methods often employ alternating iteration for auxiliary binary variables [34] or even discard the binary constraints using some relaxation techniques [9]. In contrast, our model employs the distributional derivative estimator [8] to compute the gradients across binary latent variables, ensuring that the binary constraints are not violated. Therefore, the whole TBH model can be conveniently optimized by the standard Stochastic Gradient Descent (SGD) algorithm.

The main contributions of this work are summarized as:

- We propose a novel unsupervised hashing framework by incorporating twin bottlenecks into a unified generative network. The binary and continuous bottlenecks work collaboratively to generate discriminative binary codes without much loss of reconstruction capability.
- A code-driven adjacency graph is proposed with efficient computation in the Hamming space. The graph is updated adaptively to better fit the inherent data structure. Moreover, GCNs are leveraged to further exploit the data relationships.
- The auto-encoding framework is novely leveraged to determine the reward of the encoding quality on top of the code-driven graph, shaping the idea of learning similarities by decoding.
- Extensive experiments show that the proposed TBH model massively boosts the state-of-the-art retrieval performance on four large-scale image datasets.

2. Related Work

Learning to hash, including the supervised and unsupervised scenario [5, 6, 9, 12, 24, 27, 29, 30], has been studied for years. This work is mostly related to the graph-based approaches [43, 29, 30, 35] and deep generative models based ones [5, 8, 10, 34, 40, 50].

Unsupervised hashing with graphs. As a well-known graph-based approach, Spectral Hashing (SpH) [43] determines pair-wise code distances according to the graph Laplacians of the data similarity affinity in the original feature space. Anchor Graph Hashing (AGH) [30] successfully defines a small set of anchors to approximate this graph. These approaches assume that the original or mid-level data feature distance reflects the actual data relevance. As discussed in the problem of ‘static graph’, this is not always realistic. Additionally, the pre-computed graph is isolated from the training process, making it hard to obtain optimal codes. Although this issue has already been considered in [35] by an alternating code updating scheme, its similarity graph is still only dependent on real-valued features during training. We decide to build the batch-wise graph directly upon the Hamming distance so that the learned graph is automatically optimized by the neural network.

Unsupervised generative hashing. Stochastic Generative Hashing (SGH) [8] is closely related to our model in the way of employing the auto-encoding framework and the discrete stochastic neurons. SGH [8] simply utilizes the binary latent variables as the encoder-decoder bottleneck. This design does not fully consider the code similarity and may lead to high reconstruction error, which harms the training effectiveness (‘deficient BinBN’). Though auto-encoding schemes apply deterministic decoding error, we are also aware that some existing models [10, 40, 50] are proposed with implicit reconstruction likelihood such as the discriminator in Generative Adversarial Network (GAN) [13]. Note that TBH also involves adversarial training, but only for regularization purpose.
3. Proposed Model

TBH produces binary features of the given data for efficient ANN search. Given a data collection \( X = \{x_i\}_{i=1}^N \in \mathbb{R}^{N \times D} \), the goal is to learn an encoding function \( h : \mathbb{R}^D \to (0, 1)^M \). Here \( N \) refers to the set size; \( D \) indicates the original data dimensionality and \( M \) is the target code length. Traditionally, the code of a data point, e.g., an image or a feature vector, is obtained by applying an element-wise sign function (i.e., \( \text{sign}(\cdot) \)) to the encoding function:

\[
b = \frac{\text{sign}(h(x) - 0.5) + 1}{2} \in \{0, 1\}^M,
\]

where \( b \) is the binary code. Some auto-encoding hashing methods [8, 38] introduce stochasticity on the encoding layer (see Eq. (2)) to estimate the gradient across \( b \). We also adopt this paradigm to make TBH fully trainable with SGD, while during test, Eq. (1) is used to encode out-of-sample data points.

3.1. Network Overview

The network structure of TBH is illustrated in Fig. 2. It typically involves a twin-bottleneck auto-encoder for our unsupervised feature learning task and two WAE [42] discriminators for latent variable regularization. The network setting, including the numbers of layers and hidden states, is also provided in Fig. 2.

An arbitrary datum \( x \) is firstly fed to the encoders to produce two sets of latent variables, i.e., the binary code \( b \) and the continuous feature \( z \). Note that the back-propagatable discrete stochastic neurons are introduced to obtain the binary code \( b \). This procedure will be explained in Sec. 3.2.1. Subsequently, a similarity graph within a training batch is built according to the Hamming distances between binary codes. As shown in Fig. 2, we use an adjacency matrix \( A \) to represent this batch-wise similarity graph. The continuous latent variable \( z \) is tweaked by Graph Convolutional Network (GCN) [19] with the adjacency \( A \), resulting in the final latent variable \( z' \) (see Sec. 3.2.2) for reconstruction. Following [42], two discriminators \( d_1(\cdot) \) and \( d_2(\cdot) \) are utilized to regularize the latent variables, producing informative 0-1 balanced codes.

3.1.1 Why Does This Work?

Our key idea is to utilize the reconstruction loss of the auxiliary decoder side as sort of reward/critic to score the encode quality through the GCN layer and encoder. Hence, TBH directly solves both ‘static graph’ and ‘deficient BinBN’ problems. First of all, the utilization of continuous latent variables mitigates the information loss on the binary bottleneck in [5, 8], as more detailed data information can be kept. This design promotes the reconstruction quality and training effectiveness. Secondly, a direct back-propagation pathway from the decoder to the binary encoder \( f_1(\cdot) \) is established through the GCN [19]. The GCN layer selectively mixes and tunes the latent data representation based on code distances so that data with similar binary representations have stronger influences on each other. Therefore, the binary encoder is effectively trained upon successfully detecting relevant data for reconstruction.

3.2. Auto-Encoding Twin-Bottleneck

3.2.1 Encoder: Learning Factorized Representations

Different from conventional auto-encoders, TBH involves a twin-bottleneck architecture. Apart from the \( M \)-bit binary code \( b \in \{0, 1\}^M \), the continuous latent variable \( z \in \mathbb{R}^L \) is introduced to capture detailed data information. Here \( L \)
refers to the dimensionality of $z$. As shown in Fig. 2, two encoding layers, respectively for $b$ and $z$, are topped on the identical fully-connected layer which receives original data representations $x$. We denote these two encoding functions, \textit{i.e.}, $[b, z] = f(x; \theta)$, as follows:
\begin{equation}
  b = \alpha \left( f_1(x; \theta_1), e \right) \in \{0, 1\}^M,
  \quad z = f_2(x; \theta_2) \in \mathbb{R}^L,
\end{equation}
where $\theta_1$ and $\theta_2$ indicate the network parameters. Note that $\theta_1$ overlaps with $\theta_2$ w.r.t. the weights of the shared fully-connected layer. The first layer of $f_1(\cdot)$ and $f_2(\cdot)$ comes with a ReLU [32] non-linearity. The activation function for the second layer of $f_1(\cdot)$ is the sigmoid function to restrict its output values within an interval of $(0, 1)$, while $f_2(\cdot)$ uses ReLU [32] non-linearity again on the second layer.

More importantly, $\alpha(e, \epsilon)$ in Eq. (2) is the element-wise discrete stochastic neuron activation [8] with a set of random variables $\epsilon \sim \mathcal{U}(0, 1)^M$, which is used for back-propagation through the binary variable $b$. A discrete stochastic neuron is defined as:
\begin{equation}
  b^{(i)} = \alpha \left( f_1(x; \theta_1), e^{(i)} \right) = \begin{cases}
  1 & f_1(x; \theta_1)^{(i)} \geq \epsilon^{(i)}, \\
  0 & f_1(x; \theta_1)^{(i)} < \epsilon^{(i)},
\end{cases}
\end{equation}
where the superscript $(i)$ denotes the $i$-th element in the corresponding vector. During the training phase, this operation preserves the binary constraints and allows gradient estimation through distributional derivative [8] with Monte Carlo sampling, which will be elaborated later.

### 3.2.2 Bottlenecks: Code-Driven Hamming Graph

Different from the existing graph-based hashing approaches [29, 30, 48] where graphs are basically \textit{fixed} during training, TBH automatically detects relevant data points in a graph and mixes their representations for decoding with a back-propagatable scheme.

The outputs of the encoder, \textit{i.e.}, $b$ and $z$, are utilized to produce the final input $z'$ to the decoder. For simplicity, we use batch-wise notations with capitalized letters. In particular, $z_B = [z_1, z_2, \cdots, z_N]$ $\in \mathbb{R}^{N_B \times L}$ and $B = [b_1, b_2, \cdots, b_N]$ $\in \{0, 1\}^{N_B \times M}$ respectively refer to the continuous and binary latent variables for a batch of $N_B$ data points. The inputs to the decoder are therefore denoted as $z_B' = [z'_1, z'_2, \cdots, z'_N] \in \mathbb{R}^{N_B \times L}$. We construct the graph based on the whole training batch with each datum as a vertex, and the edges are determined by the Hamming distances between the binary codes. The normalized graph adjacency $A \in [0, 1]^{N_B \times N_B}$ is computed by:
\begin{equation}
  A = I + \frac{1}{M} \left( B_B (B_B - J)^T + (B_B - J)B_B^T \right),
\end{equation}
where $J = I^T I$ is a matrix full of ones. Eq. (4) is an equilibrium of $A_{ij} = 1 - \text{Hamming}(b_i, b_j)/M$ for each entry of $A$. Then this adjacency, together with the continuous variables $Z_B$, is processed by the GCN layer [19], which is defined as:
\begin{equation}
  Z_B' = \text{sigmoid} \left( D^{-\frac{1}{2}} A D^{-\frac{1}{2}} (Z_B W_{\theta_B}) \right).
\end{equation}
Here $W_{\theta_B} \in \mathbb{R}^{L \times L}$ is a set of trainable projection parameters and $D = \text{diag} \left( A^T I \right)$.

As the batch-wise adjacency $A$ is constructed exactly from the codes, a trainable pathway is then established from the decoder to $B_B$. Intuitively, the reconstruction penalty scales up when unrelated data are closely located in the Hamming space. Ultimately, only relevant data points with similar binary representations are linked during decoding. Although GCNs [19] are utilized as well in [38, 48], these works generally use pre-computed graphs and cannot handle the \textit{static graph} problem.

### 3.2.3 Decoder: Rewarding the Hashing Results

The decoder is an auxiliary component to the model, determining the code quality produced by the encoder. As shown in Fig. 2, the decoder $g(\cdot)$ of TBH consists of two fully-connected layers, which are topped on the GCN [19] layer. We impose an ReLU [32] non-linearity on the first layer and an identity activation on the second layer. Therefore, the decoding output $\hat{x}$ is represented as $x = g(z'; \theta_1) \in \mathbb{R}^D$, where $\theta_1$ refers to the network parameters within the scope of the decoder and $z'$ is a row vector of $Z_B$ generated by the GCN [19]. We elaborate the detailed loss of the decoding side in Sec. 3.4.

To keep the content concise, we do not propose a large convolutional network receiving and generating raw images, since our goal is to learn compact binary features. The decoder provides deterministic reconstruction penalty, \textit{e.g.}, the $\ell_2$-norm, back to the encoders during optimization. This ensures a more stable and controllable training procedure than the implicit generation penalties, \textit{e.g.}, the discriminators in GAN-based hashing [10, 40, 50].

### 3.3 Implicit Bottleneck Regularization

The latent variables in the bottleneck are regularized to avoid wasting bits and align representation distributions. Different from the deterministic regularization terms such as bit de-correlation [9, 27] and entropy-like loss [8], TBH mimics WAE [42] to adversarially regularize the latent variables with auxiliary discriminators. The detailed settings of the discriminators, \textit{i.e.}, $d_1(\cdot; \varphi_1)$ and $d_2(\cdot; \varphi_2)$ with network parameters $\varphi_1$ and $\varphi_2$, are illustrated in Fig. 2, particularly involving two fully-connected layers successively with ReLU [32] and sigmoid non-linearities.

In order to balance zeros and ones in a binary code, we assume that $b$ is priored by a binomial distribution $\mathcal{B}(M, 0.5)$, which could maximize the code entropy. Meanwhile, regularization is also applied to the continuous variables $z'$ after the GCN for decoding. We expect $z'$ to obey a uniform distribution $\mathcal{U}(0, 1)^L$ to fully explore the latent
space. To that end, we employ the following two discriminators $d_1$ and $d_2$ for $b$ and $z'$, respectively:

$$
\begin{align*}
    d_1(b; \varphi_1) &\in (0, 1), \\
    d_1(y^{(b)}; \varphi_1) &\in (0, 1), \\
    d_2(z'; \varphi_2) &\in (0, 1), \\
    d_2(y^{(c)}; \varphi_2) &\in (0, 1),
\end{align*}
$$

where $y^{(b)} \sim B(M, 0.5)$ and $y^{(c)} \sim U(0, 1)^L$ are random signals sampled from the targeted distributions for implicit regularizing $b$ and $z'$ respectively.

The WAE-like regularizers focus on minimizing the distributional discrepancy between the produced feature space and the target one. This design fits TBH better than deterministic regularizers [8, 9], since such kinds of regularizers (e.g., bit de-correlation) impose direct penalties on each sample, which may heavily skew the similarity graph built upon the codes and consequently degrades the training quality. Experiments also support our insight (see Table 3).

### 3.4. Learning Codes and Similarity by Decoding

As TBH involves adversarial training, two learning objectives, i.e., $L_{AE}$ for the auto-encoding step and $L_D$ for the discriminating step, are respectively introduced.

#### 3.4.1 Auto-Encoding Objective

The Auto-Encoding objective $L_{AE}$ is written as follows:

$$
L_{AE} = \frac{1}{N_B} \sum_{i=1}^{N_B} \mathbb{E}_{b_i} \left[ \frac{1}{2M} \| x_i - \hat{x}_i \|^2 - \lambda \log d_1(b_i; \varphi_1) - \lambda \log d_2(z'_i; \varphi_2) \right],
$$

where $\lambda$ is a hyper-parameter controlling the penalty of the discriminator according to [42]. $b$ is obtained from Eq. (3), $z'$ is computed according to Eq. (5), and the decoding result $\hat{x} = g(z'; \theta_1)$ is obtained from the decoder. $L_{AE}$ is used to optimize the network parameters within the scope of $\theta = \{\theta_1, \theta_2, \theta_3, \theta_4\}$. Eq. (7) comes with an expectation term $\mathbb{E}[\cdot]$ over the latent binary code, since $b$ is generated by a sampling process.

Inspired by [8], we estimate the gradient through the binary bottleneck with distributional derivatives by utilizing a set of random signals $\epsilon \sim U(0, 1)^M$. The gradient of $L_{AE}$ w.r.t. $\theta$ is estimated by:

$$
\nabla_{\theta} L_{AE} \approx \frac{1}{N_B} \sum_{i=1}^{N_B} \mathbb{E}_\epsilon \left[ \nabla_{\theta} \left( \frac{1}{2M} \| x_i - \hat{x}_i \|^2 \right) - \lambda \log d_1(b_i; \varphi_1) - \lambda \log d_2(z'_i; \varphi_2) \right].
$$

We refer the reader to [8] and our Supplementary Material for the details of Eq. (8). Notably, a similar approach of sampling-based gradient estimation for discrete variables was employed in [2], which has been proved as a special case of the REINFORCE algorithm [44].

#### 3.4.2 Discriminating Objective

The Discriminating objective $L_D$ is defined by:

$$
L_D = -\frac{\lambda}{N_B} \sum_{i=1}^{N_B} \left( \log d_1(y^{(b)}_i; \varphi_1) + \log d_2(y^{(c)}_i; \varphi_2) + \log (1 - d_1(b_i; \varphi_1)) + \log (1 - d_2(z'_i; \varphi_2)) \right).
$$

Here $\lambda$ refers to the same hyper-parameter as in Eq. (7). Similarly, $L_D$ optimizes the network parameters within the scope of $\varphi = \{\varphi_1, \varphi_2\}$. As the discriminating step does not propagate error back to the auto-encoder, there is no need to estimate the gradient through the indifferentiable binary bottleneck. Thus the expectation term $\mathbb{E}[\cdot]$ in Eq. (7) is deprecated in Eq. (9).

The training procedure of TBH is summarized in Algorithm 1, where $\Gamma(\cdot)$ refers to the adaptive gradient scaler, for which we adopt the Adam optimizer [18]. Monte Carlo sampling is performed on the binary bottleneck, once a data batch is fed to the encoder. Therefore, the learning objective can be computed using the network outputs.

##### Algorithm 1: The Training Procedure of TBH

| Input: | Dataset $X = \{x_i\}_{i=1}^N$ and the maximum number of iterations $T$. |
| Output: | Network parameters $\theta_1$. |
| repeat | |
| Randomly select a mini-batch $\{x_i\}_{i=1}^{N_B}$ from $X$ |
| for $i = 1, \ldots, N_B$ do | |
| Sample $\epsilon_i \sim U(0, 1)^M$ |
| Sample $y^{(b)}_i \sim B(M, 0.5)$; $y^{(c)}_i \sim U(0, 1)^L$ |
| Sample a set of $b_i$ according to Eq. (3) using $\epsilon_i$ |
| end |
| Discriminating step: |
| $L_D \leftarrow$ Eq. (9) |
| $\varphi \leftarrow \varphi - \Gamma(\nabla_{\varphi} L_D)$ |
| Auto-encoding step: |
| $L_{AE} \leftarrow$ Eq. (7) |
| $\theta \leftarrow \theta - \Gamma(\nabla_{\theta} L_{AE})$ according to Eq. (8) |
| until convergence or reaching the maximum iteration $T$ |

After TBH is trained, we can obtain the binary codes for any out-of-sample data as follows:

$$
\begin{align*}
    b^{(q)} &= (\text{sign}(f_1(x^{(q)}; \theta_1) - 0.5) + 1)/2 \in \{0, 1\}^M, \tag{10}
\end{align*}
$$

where $x^{(q)}$ denotes a query data point. During the test phase, only $f_1(\cdot)$ is required, which considerably eases the binary coding process. Since only the forward propagation process is involved for test data, the stochasticity on the encoder $f_1(\cdot)$ used for training in Eq. (2) is not needed.

#### 4. Experiments

We evaluate the performance of the proposed TBH on four large-scale image benchmarks, i.e., CIFAR-10, NUS-WIDE, MS COCO. We additionally present results for image reconstruction on the MNIST dataset.
4.1. Implementation Details

The proposed TBH model is implemented with the popular deep learning toolbox Tensorflow [1]. The hidden layer sizes and the activation functions used in TBH are all provided in Fig. 2. The gradient estimation of Eq. (8) can be implemented with a single Tensorflow decorator in Python, following [8]. TBH only involves two hyper-parameters, i.e., $\lambda$ and $L$. We set $\lambda = 1$ and $L = 512$ by default. For all of our experiments, the $f_{c_{x}}$ features of the AlexNet [22] network are utilized for data representation. The learning rate of Adam optimizer $\Gamma (\cdot)$ [18] is set to $1 \times 10^{-4}$, with default decay rates $\beta_{1} = 0.9$ and $\beta_{2} = 0.999$. We fix the training batch size to 400. Our implementation can be found at https://github.com/ymcidence/TBH.

4.2. Datasets and Setup

**CIFAR-10** [21] consists of 60,000 images from 10 classes. We follow the common setting [10, 41] and select 10,000 images (1000 per class) as the query set. The remaining 50,000 images are regarded as the database.

**NUS-WIDE** [7] is a collection of nearly 270,000 images of 81 categories. Following the settings in [45], we adopt the subset of images from 21 most frequent categories. 100 images of each class are utilized as a query set and the remaining images form the database. For training, we employ 10,500 images uniformly selected from the 21 classes.

**MS COCO** [28] is a benchmark for multiple tasks. We adopt the pruned set as with [4] with 12,2218 images from 80 categories. We randomly select 5,000 images as queries with the remaining ones the database, from which 10,000 images are chosen for training.

Standard metrics [4, 41] are adopted to evaluate our method and other state-of-the-art methods, i.e., Mean Average Precision (MAP), Precision-Recall (P-R) curves, Precision curves within Hamming radius 2 (P@H≤2), and Precision w.r.t. 1,000 top returned samples (P@1000). We adopt MAP@1000 for CIFAR-10, and MAP@5000 for MS COCO and NUS-WIDE according to [4, 49].

4.3. Comparison with Existing Methods

We compare TBH with several state-of-the-art unsupervised hashing methods, including LSH [6], ITQ [12], SpH [43], SpherH [17], KMH [15], AGH [30], DGH [29], DeepBit [27], BGAN [40], HashGAN [16], SGH [8], BinGAN [50], GreedyHash [41], DBN [37] and DistillHash [46]. For fair comparisons, all the methods are reported with identical training and test sets. Additionally, the shallow methods are evaluated with the same deep features as the ones we are using.

### 4.3.1 Retrieval results

The MAP and P@1000 results of TBH and other methods are respectively provided in Tables 1 and 2, while the

<table>
<thead>
<tr>
<th>Method</th>
<th>Reference</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSH [6]</td>
<td>STOC02</td>
<td>0.106</td>
<td>0.102</td>
<td>0.105</td>
<td>0.239</td>
<td>0.266</td>
<td>0.266</td>
<td>0.355</td>
<td>0.372</td>
<td>0.341</td>
</tr>
<tr>
<td>SpH [43]</td>
<td>NIPS09</td>
<td>0.272</td>
<td>0.285</td>
<td>0.300</td>
<td>0.517</td>
<td>0.511</td>
<td>0.510</td>
<td>0.527</td>
<td>0.529</td>
<td>0.546</td>
</tr>
<tr>
<td>AGH [30]</td>
<td>ICM11</td>
<td>0.333</td>
<td>0.357</td>
<td>0.358</td>
<td>0.592</td>
<td>0.615</td>
<td>0.616</td>
<td>0.596</td>
<td>0.625</td>
<td>0.631</td>
</tr>
<tr>
<td>SpherH [17]</td>
<td>CVPR12</td>
<td>0.254</td>
<td>0.291</td>
<td>0.333</td>
<td>0.495</td>
<td>0.558</td>
<td>0.582</td>
<td>0.516</td>
<td>0.547</td>
<td>0.589</td>
</tr>
<tr>
<td>KMH [15]</td>
<td>CVPR13</td>
<td>0.279</td>
<td>0.296</td>
<td>0.334</td>
<td>0.562</td>
<td>0.597</td>
<td>0.600</td>
<td>0.543</td>
<td>0.554</td>
<td>0.592</td>
</tr>
<tr>
<td>ITQ [12]</td>
<td>PAMI13</td>
<td>0.305</td>
<td>0.325</td>
<td>0.349</td>
<td>0.627</td>
<td>0.645</td>
<td>0.664</td>
<td>0.598</td>
<td>0.624</td>
<td>0.648</td>
</tr>
<tr>
<td>DGH [29]</td>
<td>NIPS14</td>
<td>0.335</td>
<td>0.353</td>
<td>0.361</td>
<td>0.572</td>
<td>0.607</td>
<td>0.627</td>
<td>0.613</td>
<td>0.631</td>
<td>0.638</td>
</tr>
<tr>
<td>DeepBit [27]</td>
<td>CVPR16</td>
<td>0.194</td>
<td>0.249</td>
<td>0.277</td>
<td>0.392</td>
<td>0.403</td>
<td>0.429</td>
<td>0.407</td>
<td>0.419</td>
<td>0.430</td>
</tr>
<tr>
<td>SGH [8]</td>
<td>ICM17</td>
<td>0.435</td>
<td>0.437</td>
<td>0.433</td>
<td>0.593</td>
<td>0.590</td>
<td>0.607</td>
<td>0.594</td>
<td>0.610</td>
<td>0.618</td>
</tr>
<tr>
<td>BGAN [40]</td>
<td>AAAI18</td>
<td>0.525</td>
<td>0.531</td>
<td>0.562</td>
<td>0.684</td>
<td>0.714</td>
<td>0.730</td>
<td>0.645</td>
<td>0.682</td>
<td>0.707</td>
</tr>
<tr>
<td>BinGAN [50]</td>
<td>NIPS18</td>
<td>0.476</td>
<td>0.512</td>
<td>0.520</td>
<td>0.654</td>
<td>0.709</td>
<td>0.713</td>
<td>0.651</td>
<td>0.673</td>
<td>0.696</td>
</tr>
<tr>
<td>GreedyHash [41]</td>
<td>NIPS18</td>
<td>0.448</td>
<td>0.473</td>
<td>0.501</td>
<td>0.633</td>
<td>0.691</td>
<td>0.731</td>
<td>0.582</td>
<td>0.668</td>
<td>0.710</td>
</tr>
<tr>
<td>HashGAN [10]*</td>
<td>CVPR18</td>
<td>0.447</td>
<td>0.463</td>
<td>0.481</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DBN [37]</td>
<td>IJCV19</td>
<td>0.403</td>
<td>0.422</td>
<td>0.446</td>
<td>0.604</td>
<td>0.632</td>
<td>0.665</td>
<td>0.570</td>
<td>0.629</td>
<td>0.623</td>
</tr>
<tr>
<td>DistillHash [46]</td>
<td>CVPR19</td>
<td>0.284</td>
<td>0.285</td>
<td>0.288</td>
<td>0.667</td>
<td>0.675</td>
<td>0.677</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>TBH</strong></td>
<td>Proposed</td>
<td><strong>0.532</strong></td>
<td><strong>0.573</strong></td>
<td><strong>0.578</strong></td>
<td><strong>0.717</strong></td>
<td><strong>0.725</strong></td>
<td><strong>0.735</strong></td>
<td><strong>0.706</strong></td>
<td><strong>0.735</strong></td>
<td><strong>0.722</strong></td>
</tr>
</tbody>
</table>

*Note the duplicate naming of HashGAN, i.e., the unsupervised one [10] and the supervised one [4].

Table 2. P@1000 results of TBH and compared methods on CIFAR-10 and MS COCO.

<table>
<thead>
<tr>
<th>Method</th>
<th>CIFAR-10</th>
<th>MS COCO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>KMH</td>
<td>0.242</td>
<td>0.252</td>
</tr>
<tr>
<td></td>
<td>0.557</td>
<td>0.572</td>
</tr>
<tr>
<td>SpherH</td>
<td>0.228</td>
<td>0.256</td>
</tr>
<tr>
<td></td>
<td>0.525</td>
<td>0.571</td>
</tr>
<tr>
<td>ITQ</td>
<td>0.276</td>
<td>0.292</td>
</tr>
<tr>
<td></td>
<td>0.607</td>
<td>0.637</td>
</tr>
<tr>
<td>SpH</td>
<td>0.238</td>
<td>0.239</td>
</tr>
<tr>
<td></td>
<td>0.541</td>
<td>0.548</td>
</tr>
<tr>
<td>AGH</td>
<td>0.306</td>
<td>0.321</td>
</tr>
<tr>
<td></td>
<td>0.602</td>
<td>0.635</td>
</tr>
<tr>
<td>DGH</td>
<td>0.315</td>
<td>0.323</td>
</tr>
<tr>
<td></td>
<td>0.623</td>
<td>0.642</td>
</tr>
<tr>
<td>HashGAN</td>
<td>0.418</td>
<td>0.436</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SGH</td>
<td>0.387</td>
<td>0.380</td>
</tr>
<tr>
<td></td>
<td>0.604</td>
<td>0.615</td>
</tr>
<tr>
<td>GreedyHash</td>
<td>0.322</td>
<td>0.403</td>
</tr>
<tr>
<td></td>
<td>0.603</td>
<td>0.624</td>
</tr>
<tr>
<td><strong>TBH (Ours)</strong></td>
<td><strong>0.497</strong></td>
<td><strong>0.524</strong></td>
</tr>
<tr>
<td></td>
<td><strong>0.646</strong></td>
<td><strong>0.698</strong></td>
</tr>
</tbody>
</table>
respective P-R curves and P@H≤2 results are illustrated in Fig. 3. The performance gap between TBH and existing unsupervised methods can be clearly observed. Particularly, TBH obtains remarkable MAP gain with 16-bit codes (i.e., \( M = 16 \)). Among the unsupervised baselines, GreedyHash [41] performs closely next to TBH. It bases the produced code similarity on pair-wise feature distances. As is discussed in Sec 1, this design is straightforward but sub-optimal since the original feature space is not fully revealing data relevance. On the other hand, as a generative model, HashGAN [10] significantly underperforms TBH, as the binary constraints are violated during its adversarial training. TBH differs SGH [8] by leveraging the twin-bottleneck scheme. Since SGH [8] only considers the reconstruction error and in auto-encoder, it generally does not produce convincing retrieval results.

### 4.3.2 Extremely short codes

Inspired by [41], we illustrate the retrieval performance with extremely short bit length in Fig. 4 (a). TBH works well even when the code length is set to \( M = 4 \). The significant performance gain over SGH can be observed. This is due to that, during training, the continuous bottleneck complements the information discarded by the binary one.

### 4.4. Ablation Study

In this subsection, we validate the contribution of each component of TBH, and also show some empirical analysis. Different baseline network structures are visualized in the Supplementary Material for better understanding.

#### 4.4.1 Component Analysis

We compare TBH with the following baselines. (1) **Single bottleneck.** This baseline coheres with SGH. We remove the twin-bottleneck structure and directly feed the binary codes to the decoder. (2) **Swapped bottlenecks.** We swap the functionalities of the two bottlenecks, i.e., using the continuous one for adjacency building and the binary one for decoding. (3) **Explicit regularization.** The WAE regularizers are replaced by conventional regularization terms. An entropy loss similar to SGH is used to regularize \( b \), while an \( \ell_2 \)-norm is applied to \( z' \). (4) **Without regularization.** The regularization terms for \( b \) and \( z' \) are removed in this baseline. (5) **Without stochastic neuron.** The discrete stochastic neuron \( \alpha (\cdot, \epsilon) \) is deprecated on the top of \( f_1 (\cdot) \), and bit quantization loss [9] is appended to \( L_{AE} \). (6) **Fixed graph.** \( A \) is pre-computed using feature distances. The continuous bottleneck is removed and the GCN is applied to the binary bottleneck with the fixed \( A \). (7) **Attention equilibrium.** This baseline performs weighted average on \( Z \) according to \( A \), instead of employing GCN in-between.

Table 3 shows the performance of the baselines. We can observe that the model undergoes significant performance drop when modifying the twin-bottleneck structure. Specifically, our trainable adaptive Hamming graph plays an important role in the network. When removing this (i.e., baseline 6), the performance decreases by \( \sim 9\% \). This accords with our motivation in dealing with the 'static

<table>
<thead>
<tr>
<th>Baseline</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single bottleneck</td>
<td>0.435</td>
<td>0.437</td>
<td>0.433</td>
</tr>
<tr>
<td>Swapped bottlenecks</td>
<td>0.466</td>
<td>0.471</td>
<td>0.475</td>
</tr>
<tr>
<td>Explicit regularization</td>
<td>0.524</td>
<td>0.559</td>
<td>0.560</td>
</tr>
<tr>
<td>Without regularization</td>
<td>0.521</td>
<td>0.535</td>
<td>0.547</td>
</tr>
<tr>
<td>Without stochastic neuron</td>
<td>0.408</td>
<td>0.412</td>
<td>0.463</td>
</tr>
<tr>
<td>Fixed graph</td>
<td>0.442</td>
<td>0.464</td>
<td>0.459</td>
</tr>
<tr>
<td>Attention equilibrium</td>
<td>0.477</td>
<td>0.503</td>
<td>0.519</td>
</tr>
<tr>
<td>TBH (full model)</td>
<td><strong>0.532</strong></td>
<td><strong>0.573</strong></td>
<td><strong>0.578</strong></td>
</tr>
</tbody>
</table>

Figure 3. P-R curves and P@H≤2 results of TBH and compared methods on CIFAR-10.

Figure 4. (a) MAP@1000 results with extremely short code lengths on CIFAR-10. (b) 16-bit normalized reconstruction errors of TBH and its variants on CIFAR-10. (c) and (d) Effects of the hyper-parameters \( \lambda \) and \( L \) on CIFAR-10.
Figure 5. Adjacency matrices of 20 randomly-sampled data points of a training batch on CIFAR-10, computed based on ground-truth similarity (Left) and the Hamming distances between binary codes using Eq. (4) (Right).

Graph’ problem. In practice, we also experience training perturbations when applying different regularization and quantization penalties to the model.

4.4.2 Reconstruction Error
As mentioned in the ‘deficient BinBN’ problem, decoding from a single binary bottleneck is less effective. This is illustrated in Fig. 4 (b), where the normalized reconstruction errors of TBH, baseline_1 and baseline_4 are plotted. TBH produces lower decoding error than the single bottleneck baseline. Note that baseline_1 structurally coheres SGH.

4.4.3 Hyper-Parameter
Only two hyper-parameters are involved in TBH. The effect of the adversarial loss scaler $\lambda$ is illustrated in Fig. 4 (c). A large regularization penalty slightly influences the overall retrieval performance. The results w.r.t. different settings of $L$ on CIFAR-10 are shown in Fig. 4 (d). Typically, no dramatic performance drop is observed when squeezing the bottleneck, as data relevance is not only reflected by the continuous bottleneck. Even when setting $L$ to 64, TBH still outperforms most existing unsupervised methods, which also endorses our twin-bottleneck mechanism.

4.5. Qualitative Results
We provide some intuitive results to further justify the design. The implementation details are given in the Supplementary Material to keep the content concise.

4.5.1 The Constructed Graph by Hash Codes
We show the effectiveness of the code-driven graph learning process in Fig. 5. 20 random samples are selected from a training batch to plot the adjacency. The twin-bottleneck mechanism automatically tunes the codes, constructing $\mathbf{A}$ based on Eq. (4). Though TBH has no access to the labels, the constructed adjacency simulates the label-based one. Here brighter color indicates closer Hamming distances.

4.5.2 Visualization
Fig. 6 (a) shows the t-SNE [31] visualization results of TBH. Most of the data from different categories are clearly scattered. Interestingly, TBH successfully locates visually similar categories within short Hamming distances, e.g., Automobile/Truck and Deer/Horse. Some qualitative image retrieval results w.r.t. 32-bit codes are shown in Fig. 6 (b).

4.5.3 Toy Experiment on MNIST
Following [8], another toy experiment on image reconstruction with the MNIST [26] dataset is conducted. For this task, we directly use the flattened image pixels as the network input. The reconstruction results are reported in Fig. 6 (c). Some bad handwritings are falsely decoded to wrong numbers by SGH [8], while this phenomenon is not frequently observed in TBH. This also supports our insight in addressing the ‘deficient BinBN’ problem.

5. Conclusion
In this paper, a novel unsupervised hashing model was proposed with an auto-encoding twin-bottleneck mechanism, namely Twin-Bottleneck Hashing (TBH). The binary bottleneck explored the intrinsic data structure by adaptively constructing the code-driven similarity graph. The continuous bottleneck interactively adopted data relevance information from the binary codes for high-quality decoding and reconstruction. The proposed TBH model was fully trainable with SGD and required no empirical assumption on data similarity. Extensive experiments revealed that TBH remarkably boosted the state-of-the-art unsupervised hashing schemes in image retrieval.
[38] Yuming Shen, Li Liu, Fumin Shen, and Ling Shao. Zero-shot sketch-image hashing. In *CVPR*, 2018. 3, 4


