Pattern Recognition 95 (2019) 114–127

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/patcog

Deep forest hashing for image retrieval

Meng Zhou, Xianhua Zeng*, Aozhu Chen

Chongqing Key Laboratory of Image Cognition, College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China

ARTICLE INFO

Article history:
Received 5 December 2018
Revised 21 May 2019
Accepted 5 June 2019
Available online 5 June 2019

Keywords:
Hashing learning
Image retrieval
Deep forest hashing
Shorter binary codes

ABSTRACT

Hashing methods have been intensively studied and widely used in image retrieval. Hashing methods aim to learn a group of hash functions to map original data into compact binary codes and simultaneously preserve some notion of similarity in the Hamming space. The generated binary codes are effective for image retrieval and highly efficient for large-scale data storage. The decision tree is a fast and interpretable model, but the current decision tree based hashing methods have insufficient learning ability due to the use of shallow decision trees. Most current deep hashing methods are based on deep neural networks. However, considering the deficiencies of deep neural network-based hashing, such as the presence of too many hyperparameters, poor interpretability, and requirement for expensive and powerful computational facilities during the training process, a non-deep neural network-based hashing model need to be designed to achieve efficient image retrieval with few hyperparameters, easy theoretical analysis and an efficient training process. The multi-grained cascade forest (gcForest) is a novel deep model that generates a deep forest ensemble classifier to process data layer-by-layer with multi-grained scanning and a cascade forest. To date, gcForest has not been used to generate compact binary codes; therefore, we propose a deep forest-based method for hashing learning that aims to learn shorter binary codes to achieve effective and efficient image retrieval. The experimental results show that the proposed method has better performance with shorter binary codes than other corresponding hashing methods.

1. Introduction

The rapid development of information technology has substantially increased the data accumulated in various fields, and the age of big data has arrived. Determining how to quickly and accurately search the data that users need in massive data has become an urgent problem to be solved. Efficient retrieval on large-scale data has become a popular research topic in academia and industry. Hashing learning has attracted considerable attention in recent years due to its excellent performance in processing high-dimensional data. Our paper focuses on the deep hashing model using deep forests as hash functions with considerably fewer hyperparameters, competitive performance and convincing theoretic analysis.

Hashing learning aims to transform every data item into a low-dimensional representation, i.e., equivalently, a short code consisting of a sequence of bits referred to as hash code [1,2]. Hashing methods can be generally divided into two main categories: data-independent hashing and data-dependent hashing [3–5]. The main difference is that hash functions used by data-independent methods are either manually designed or randomly generated, while those in data-dependent methods are automatically learned from the data. The most representative data-independent hashing methods for image retrieval tasks are locality-sensitive hashing (LSH) [6] and its variants superbit LSH [7], nonmetric LSH [8], kernelized LSH [9] and LSH with faster computation of the hash functions [10]. LSH uses a set of randomly generated hyperplanes that are sampled from a Gaussian distribution, and then projects the original high-dimensional data onto the hyperplane, and finally thresholds the projection results as the output of the hash functions. The emergence of LSH greatly improves the efficiency of image retrieval and provides a new perspective for solving large-scale image retrieval. However, the hash functions in data-independent methods represented by LSH are randomly generated or manually specified, irrespective of the distribution of the original data, so the accuracy of the algorithm increases slowly when the number of bits increases. Therefore, it is difficult to obtain a stable retrieval result in practical applications. Different from data-independent methods, the hash function for each code in data-dependent methods is learned from the data, and has practical implications.

Data-dependent methods seem to be the trend in current research and applications. Spectral hashing (SH) [11], a classic data-dependent hash method, uses a complete dataset from the

* Corresponding author.
E-mail address: zengxh@cqupt.edu.cn (X. Zeng).

https://doi.org/10.1016/j.patcog.2019.06.005
0031-3203/

training set to construct a complete graph with the similarity (Gaussian similarity) between all data samples as the weights of the edges. Each hash function can be regarded as a cut of the graph. Each corresponding cut satisfies the condition that the sum of weights corresponding to the cut edge is the smallest, and the whole complete graph is as evenly divided into two parts as possible. This problem can be transformed into the classic normalized cut problem in graph theory, and the hash codes are solved by signing the eigenvectors corresponding to the minimum eigenvalues of the Laplacian matrix. Data-dependent method can be further categorized as supervised and unsupervised hashing according to the label information. Unsupervised hashing methods attempt to preserve the similarity in the original feature space and supervised hashing methods aim to preserve the semantic similarity. Examples of unsupervised methods include iterative quantization (ITQ) [3], isotropic hashing (IsoHash) [4], discrete graph hashing (DGH) [12], and scalable graph hashing (SGH) [13]. Examples of supervised methods include supervised hashing with kernels (KSH) [14], two step hashing (TSH) [15], fast supervised hashing (FastH) [16,17], supervised discrete hashing (SDH) [18] and its fast version, fast supervised discrete hashing (FSDH) [19], supervised discrete discriminant hashing (SDDH) [20], ranking-based supervised hashing (RSH) [21] and discrete semantic ranking hashing (DSeRH) [22]. Quantization-based hashing (QBH) [23] is a general framework applied to both unsupervised and supervised hashing.

In recent years, deep learning has achieved outstanding results in various fields, especially in speech recognition and computer vision. Semantic hashing [24] is the first work using deep learning; afterwards, many scholars considered the combination of a hashing method and deep learning [25,26], such as semi-supervised deep learning hashing (DLH) [27], network in network hashing (NINH) [28], convolutional neural network hashing (CNNH) [29], similarity-adaptive deep hashing (SADH) [30], deep semantic ranking-based hashing (DSRH) [31], deep hashing based on classification and quantization errors (DHCQ) [32], deep supervised discrete hashing (DSDH)[33] and deep pairwise-supervised hashing (DPSH) [34]. DPSH utilizes the deep neural network to perform simultaneous feature learning and hash code learning for applications with pairwise labels. Deep learning requires that to address complicated learning tasks, it is likely that learning models must go deep [35]. Currently, the most popular deep model is the deep neural network. Although the deep neural network is powerful, there are still some shortcomings. Deep neural networks need large-scale data for training. The training process usually requires powerful computing devices. There are too many hyperparameters to learn in deep neural networks. Finally, the performance depends heavily on parameter tuning. Considering the new deep model, gcForest [35] and the shallow level decision tree used in existing tree-based and forest-based hashing methods, we propose a deep hashing model using deep forests as hash functions. To the best of our knowledge, this model represents a new deep hashing method that is distinguished from deep hashing models based on deep neural networks. The following contributions should be highlighted:

- The proposed method considers three types of similarity metrics to preserve the semantic similarity and manifold similarity among the data points in the Hamming space.
- Different sized sliding windows are used to extract multi-grained features from raw data. And the feature extraction phase is dependent on the hash function learning stage, which helps in learning better hash functions.
- Compared with deep neural network-based hashing methods, the proposed method has fewer hyperparameters, faster training speed and easier theoretical analysis.
- The proposed method learns shorter binary code representations to achieve effective and efficient image retrieval.

The rest of this paper is organized as follows. Section 2 includes some related works. Section 3 proposes our deep forest hashing method. Section 4 gives an analysis about deep forest hashing. Section 5 reports the experimental results. Section 6 concludes this paper.

2. Related work

2.1. Tree-based and forest-based hashing

FastH [16,17], the first attempt to use decision trees as hash functions, adopts a two-step learning strategy, binary code inference and learning boosted trees as hash functions to quickly learn hash codes based on supervised labels. ForestHash [36] embeds tiny convolutional neural networks (CNNs) into shallow random forests in which random trees act as hash functions by assigning the value of “1” to the visited tree leaf and “0” otherwise. Scalable forest hashing [37] utilizes multiple tree structures to support scalable coding. Each leaf node in the tree structure is encoded with a binary bit stream. Scalable forest hashing improves the hashing efficacy with balanced hash buckets by leveraging the tree structure for hierarchical data partitioning. The decision trees or forests in these tree-based models are simple but shallow-level tree models. The learning ability of these models may not be sufficient.

2.2. Deep forest

To solve complex image retrieval tasks, the model should be deep. Currently, the most widely used deep model is the deep neural network. The multi-grained cascade forest, a novel deep model, is a tree ensemble model containing two components, i.e., multi-grained scanning and a cascade forest. Compared with the deep neural network, gcForest has fewer hyperparameters and does not require fine-tuning. In contrast, hyperparameter tuning is difficult for deep neural networks. The number of cascade layers is determined according to different tasks, enabling the complexity of the model to adapt to different scale data. Furthermore, gcForest exploits two types of forests, i.e., random forest [38] and completely random tree forest [39], which helps enhance diversity. To the best of our knowledge, there is currently no work that use this deep model to generate hash codes. Motivated by this, we attempt to exploit it for hashing learning in image retrieval tasks.

2.3. Deep hashing

Most existing hashing methods are based on hand-crafted features that are independent of the hash function learning procedure, which may not be supportive for generating compact binary codes. Deep hashing methods employ deep neural networks to realize compatible feature learning with hash function learning. Our method addresses this problem by applying the fine-grained scanning component to extract multi-grained features, which is discussed in Section 3.1. As previously mentioned, although the deep neural network is powerful, it still has some deficiencies.

3. Deep forest hashing

We believe that to address a complicated image retrieval task using hash codes, learned deep models are likely important and inevitable. However, deep neural networks for hash codes require a considerable amount of data to train. The deep neural networks used for hashing learning require expensive and powerful computational facilities during the training process and have too many

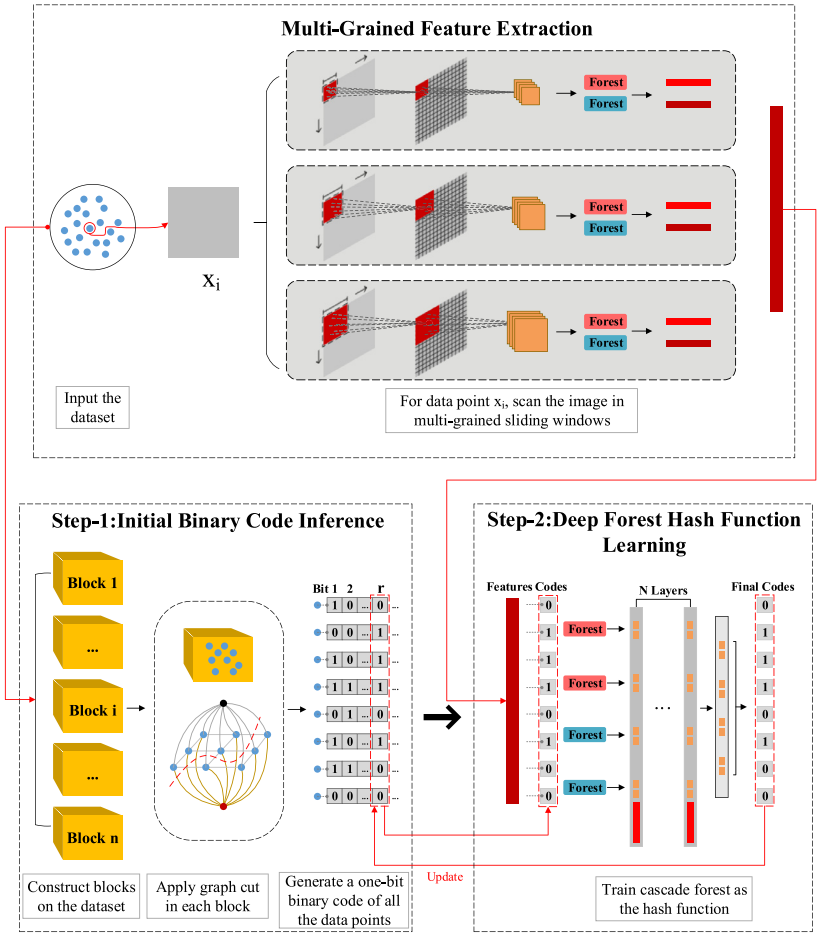


Fig. 1. Illustration of the deep forest hashing structure, including multi-grained feature extraction, initial binary code inference and deep forest hash function learning.

hyperparameters. In addition, the internal structure is similar to a black box, which is not interpretable. Deep forest hashing, a novel deep model used for hashing learning, constructs an ensemble structure to go deep adaptively and has fewer hyperparameters. Using this model, theoretical analysis is easier due to the tree-based structure. Moreover, this model can achieve effective and efficient image retrieval with learning shorter binary code representations.

Inspired by TSH [15] and FastH [16,17], we employ a two-stage learning strategy to construct our deep hashing model named the deep forest hashing model (DFH) using deep forests as hash functions to obtain much fewer hyperparameters, competitive performances and promised theoretic analysis. The overall architecture of DFH is summarized in Fig. 1. The training procedures of the DFH model are described as follows.

- First, extract the multi-grained features. Following the gcForest classifier [35], we scan the data in multi-grained sliding windows to extract features. Then, we input the multi-grained features to cascade forests to train hash functions.
- Second, step-1: initial binary code inference. We construct blocks according to the semantic similarity and compute the manifold similarity as the edge weight to construct a weighted graph. Then, in each block, we apply graph cut to generate a one-bit binary code of all data points.
- Third, step-2: deep forest hash function learning. We train cascade forests as hash functions using the binary codes obtained by graph cut as classification labels. The current binary bit is updated by applying the learned hash function.
- Finally, step-1 and step-2 are alternately performed until the maximum bit number is reached.

Suppose we have n data points (images) $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^D$ and their corresponding semantic labels $y = \{y_1, y_2, \ldots, y_n\}$. We can construct an affinity matrix Y, the element y_{ij} in Y denotes the semantic relation between data point x_i and data point x_j. Data points from the same category of ground truth are defined semantically similar relation, i.e., $y_{ij} = 1$, and a dissimilar relation, i.e., $y_{ij} = -1$, and if the pairwise relation is undefined, $y_{ij} = 0$.

In this paper, three types of similarity metrics between data points, i.e., semantic similarity, manifold similarity and Hamming similarity, are adopted. The semantic similarity is directly described by labels, which is the affinity matrix Y.

Regarding the manifold similarity, we utilize a supervised manifold learning method to preserve the neighborhood Euclidean distance between the features of the neighboring data. We attempt to preserve intraclass distance between the data points and maximize the interclass distance; distance $D(x_i, x_j)$ between the data points can be computed:

$$D(x_i, x_j) = \begin{cases} \sqrt{1 - e^{\frac{-d^2(x_i, x_j)}{\beta}}}, & y_i = y_j \\ \sqrt{e^{\frac{d^2(x_i, x_j)}{\beta}}} - \alpha, & y_i \neq y_j, \end{cases} \tag{1}$$

where $d(x_i, x_j)$ denotes the Euclidean distance between data point x_i and data point x_j, and α and β are constants. Then, manifold similarity $S_M(x_i, x_j)$ can be computed:

$$S_M(x_i, x_j) = e^{\frac{-D^2(x_i, x_j)}{\tau}}, \tag{2}$$

where τ is a constant. As the values of the semantic similarity of data points are $\{-1, 0, 1\}$, we scale $S_M(x_i, x_j)$ to $(-1, 1)$:

$$S_{M,ij} = -1 + 2\frac{S_M(x_i, x_j) - \min(S_M(x_i, x_j))}{\max(S_M(x_i, x_j)) - \min(S_M(x_i, x_j))}, \tag{3}$$

where $\max(S_M(x_i, x_j))$ and $\min(S_M(x_i, x_j))$ denote the maximum and minimum value of $S_M(x_i, x_j)$, respectively. We also use the manifold similarity to construct a weighted graph, which will be discussed in Section 3.2.

Usually, the Hamming similarity is calculated by the inner product of two binary codes:

$$S_H(x_i, x_j) = \sum_{r=1}^{k} h_r(x_i) h_r(x_j). \tag{4}$$

We aim to learn a set of hash functions $H = \{h_1, h_2, \ldots, h_k\}$ by mapping the data points in the original space to Hamming space $B \subset \{-1, 1\}^k$ and preserve both the semantic similarity and manifold similarity between the samples. We formulate hashing learning based on the three similarity, and the formulation of the loss function can be written as:

$$\min_{h_r(\bullet)} \sum_{i=1}^{n}\sum_{j=1}^{n} |y_{ij}| \left[\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{k}\sum_{r=1}^{k} h_r(x_i) h_r(x_j)\right]^2, \tag{5}$$

where $|y_{ij}|$ is devised to prevent undefined pairwise relations from influencing the hashing learning results. Notably, FastH does not include the $S_{M,ij}$. In contrast to FastH, we consider both semantic similarity and manifold similarity, which can not only help distinguish data points between different categories but also help distinguish data points that belong to the same category but have different attributes. For example, if we consider only semantic similarity, trucks and airplanes may be distinguished because they belong to different categories, but blue trucks and red ones are unlikely to be distinguished because they have different colors although they belong to the same category. We hope that data points with the same semantic label are still close when being mapped onto the Hamming space and that data points with different semantic labels are as far as possible. In addition, to achieve the purpose of having data points with the same semantic label and similar attributes closer in the Hamming space and data points with the same semantic label but different attributes not as close, we fully consider the semantic similarity and manifold similarity.

Optimizing directly for learning hash functions is difficult. Following the idea in FastH [16,17], we decompose the hashing learning into two parts: initial binary code inference and deep forest hash function learning introducing auxiliary variables:

$$z_{r,i} = h_r(x_i). \tag{6}$$

Clearly, $z_{r,i}$ represents the r-th bit of the binary code of the i-th data point. Using auxiliary variables, the problem can be decomposed into two subproblems:

$$\min_{Z \in \{-1,1\}^{k \times n}} \sum_{i=1}^{n}\sum_{j=1}^{n} |y_{ij}| \left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{k}\sum_{r=1}^{k} z_{r,i} z_{r,j}\right)^2, \tag{7a}$$

$$\min_{z_{r,i}(\bullet)} \sum_{r=1}^{m}\sum_{i=1}^{n} \delta(z_{r,i} = h_r(x_i)), \tag{7b}$$

Eq. (7a) and (7b) are the binary code inference problem and binary classification problem, which is solved in Sections 3.2 and 3.3, respectively.

3.1. Multi-grained feature extraction

Almost all existing non-deep neural network hashing methods directly use either raw data or hand-crafted features to learn hash functions. However, the feature extraction phase is independent of the hash function learning stage in hand-crafted feature-based methods, suggesting that the hand-crafted features might not be helpful in learning compact and efficient hash codes. Although deep neural network-based hashing methods can extract the deep features of raw data, we have shown the deficiencies of deep neural networks.

Here, we use multi-grained scanning, a component in gcForest, to extract multi-grained features from raw data. We use the following two types of forests for sliding window scanning: random forest and completely random tree forest. Considering the example of the MNIST dataset of handwritten digits, each data point is a grayscale image with a size of 28×28 pixels and 10 classes of digits labeled from 0 to 9. Three scanning windows with sizes of 7×7, 10×10, and 13×13 produce 121 instances of 7×7 pixels, 100 instances of 10×10 pixels and 64 instances of 13×13 pixels. Processed by random forest and completely random tree forest, each instance is transformed into a 10-dimensional feature vector. Then, the feature vectors processed by each forest are reshaped into 11×11, 10×10, and 8×8 feature matrixes with 10 layers. Subsequently, a pooling operation is conducted to preserve the main features and reduce the data dimensions. Finally, we obtain 6 groups of features processed by two types of forest and scanned by 3 sizes of windows.

$$\begin{aligned} F(x_i) \quad &= \{f_{RF,7\times7}(x_i), f_{RF,10\times10}(x_i), f_{RF,13\times13}(x_i), \\ &\quad f_{CF,7\times7}(x_i), f_{CF,10\times10}(x_i), f_{CF,13\times13}(x_i)\}, \end{aligned} \tag{8}$$

where f_{RF} denotes the features generated by random forest, f_{CF} denotes the features generated by completely random tree forest. And F denotes multi-grained feature groups.

All features, i.e., $2 \times 6 \times 6 \times 10 + 2 \times 5 \times 5 \times 10 + 2 \times 4 \times 4 \times 10 = 1540$-dimensions, are utilized to train the cascade forest as hash functions. We discuss the hash function learning occurring in step-2 in Section 3.3. Thus, we finally obtain a group of features that are optimally compatible with the hash learning procedure. The detailed procedure of the multi-grained feature extraction is shown in Fig. 2, and the hyperparameters of the multi-grained scanning layer and examples are shown in Table 1.

3.2. Initial binary code inference

Initial binary code inference aims to solve the binary code inference problem in Eq. (7a). In Eq. (7a), we optimize a one-bit sequence at a time in terms of the preceding bits. When solving the r-th bit, the cost-loss in Eq. (7a) is written as:

$$\begin{aligned} &\sum_{i=1}^{n}\sum_{j=1}^{n} |y_{ij}| \left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\sum_{p=1}^{r} z_{p,i} z_{p,j}\right)^2 \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} |y_{ij}| \left[\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\left(\sum_{p=1}^{r-1} z_{p,i} z_{p,j} + z_{r,i} z_{r,j}\right)\right]^2 \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} -2\frac{1}{r}|y_{ij}| \left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\sum_{p=1}^{r-1} z_{p,i} z_{p,j}\right) z_{r,i} z_{r,j} \\ &\quad + const. \end{aligned} \tag{9}$$

We can rewrite the optimization of the r-th bit into a binary quadratic problem:

$$\min_{z_r \in \{-1,1\}^n} \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} z_{r,i} z_{r,j}, \tag{10a}$$

$$a_{ij} = -\frac{1}{r}|y_{ij}| \left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\sum_{p=1}^{r-1} z^*_{p,i} z^*_{p,j}\right), \tag{10b}$$

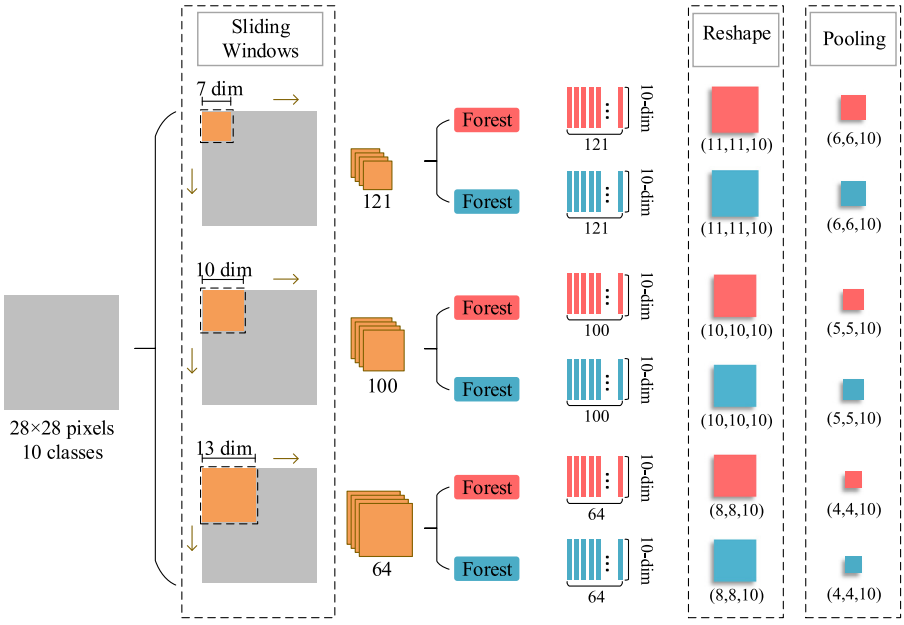


Fig. 2. Illustration of multi-grained feature extraction using sliding window scanning. In our model, we use three grains for scanning. The forest in the red box denotes the random forest and the forest in the blue box denotes the completely random tree forest. Here, we take the 28×28 pixel images in MNIST as an example. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

where z^* denotes a binary code in preceding bits.

When solving Eq. (10a), we divide the data points into a few blocks and optimize the corresponding variables in one block at a time while conditioning the remaining of variables. Let B denote a block of data points. The cost in Eq. (10a) can be rewritten as:

$$\begin{aligned}&\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} z_{r,i} z_{r,j} \\ &= \sum_{i\in B}\sum_{j\in B} a_{ij} z_{r,i} z_{r,j} + \sum_{i\in B}\sum_{j\notin B} a_{ij} z_{r,i} z_{r,j} \\ &\quad + \sum_{i\notin B}\sum_{j\in B} a_{ij} z_{r,i} z_{r,j} + \sum_{i\notin B}\sum_{j\notin B} a_{ij} z_{r,i} z_{r,j}. \end{aligned} \tag{11}$$

When optimizing one block, we set the variables that are not involved in the target block to constants. Thus, the optimization of one block can be written as:

$$\min_{z_{r,B}\in\{-1,1\}^{|B|}} \sum_{i\in B}\sum_{j\in B} a_{ij} z_{r,i} z_{r,j} + 2\sum_{i\in B}\sum_{j\notin B} a_{ij} z_{r,i} \hat{z}_{r,j}, \tag{12}$$

where $\hat{z}_r$ denotes a binary code of the r-th bit that is not involved in the target block. Using the definition of a_{ij} provided in Eq. (10b), the optimization of one block can be written as:

$$\min_{z_{r,B}\in\{-1,1\}^{|B|}} \sum_{i\in B} u_i z_{r,i} + \sum_{i\in B}\sum_{j\in B} v_{ij} z_{r,i} z_{r,j}, \tag{13a}$$

$$v_{ij} = -\frac{1}{r}|y_{ij}|\left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\sum_{p=1}^{r-1} z^*_{p,i} z^*_{p,j}\right), \tag{13b}$$

$$u_i = -2\frac{1}{r}\sum_{j\notin B} \hat{z}_{r,j} |y_{ij}|\left(\frac{1}{2}(y_{ij} + S_{M,ij}) - \frac{1}{r}\sum_{p=1}^{r-1} z^*_{p,i} z^*_{p,j}\right), \tag{13c}$$

where v_{ij} and u_i are constants.

When solving Eq. (13a), we can apply the fast and effective graph cut algorithm; $u_i z_{r,i}$ is a data term and $v_{ij} z_{r,i} z_{r,j}$ is a smoothness term. When constructing the graph, the manifold similarity is used as the edge weight to construct a weighted graph. Manifold similarity is defined in the Eqs. (1) and (2). We refer to this process as manifold graph cut.

The binary codes generated in step-1 are used as classification labels to train the cascade forest as hash functions in step-2, as shown in Fig. 3.

3.3. Deep forest hash function learning

In the second step, we solve the binary classification problem. To generate compact binary code, we employ cascade forests as hash functions to solve the classification problem:

$$\min_{h_r(\bullet)} \sum_{i=1}^{n} \delta(z_{r,i} = h_r(F(x_i))), \tag{14}$$

where $z_{r,i}$ denotes the binary codes generated in step-1, h_r denotes the cascade forest hash function and F denotes the multi-grained

Table 1
Configuration of multi-grained scanning. Take the 28×28 pixel images in MNIST as an example.

Raw data	Window size	Stride	Instances generated	Random forest	Completely random tree forest	Reshape	Pooling (average)
	7×7		121	✓		(11,11,10)	(6,6,10)
					✓		
28×28	10×10	2	100	✓		(10,10,10)	(5,5,10)
					✓		
	13×13		64	✓		(8,8,10)	(4,4,10)
					✓		

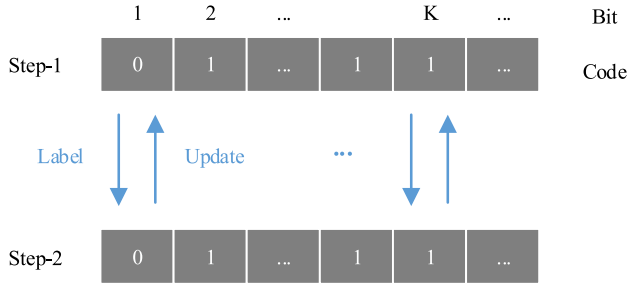


Fig. 3. Illustration of the process for alternate performance of step-1 and step-2. The initial codes gained in step-1 are used as classification labels in step-2. The result of step-2 will update the binary codes in step-1.

feature groups extracted by multi-grained sliding windows. Instead of using hand-crafted features or raw features, we use the multi-grained features as input to train cascade forest hash functions. And the binary codes $z_{r,i}$ are used as supervised labels.

The cascade forest employs a cascade structure in which each layer is an ensemble of decision tree forests. The cascade forest structure is illustrated in Fig. 4. To encourage diversity, different types of forests can be used as base classifiers. Considering the completely random tree forest and random forest, the completely random tree forest randomly selects a feature for the split as each node of the tree, and the tree grows until each leaf node contains only the same class of instances; the random forest randomly selects $\sqrt{d}$ numbers of features as candidates(d is the number of input features) and selects the one with the best Gini value for the split. In addition to the completely random tree forest and the random forest, we can replace the base classifiers with any other classifiers, such as XGBoost [40], GBDT [41] and AdaBoost [42], in the cascade layers.

Suppose that there are L layers in a cascade forest and T_l forests in each cascade layer. l denotes the index of the layer of the cascade forest, and $l = 1, \ldots, L$. t denotes the index of the forest in each cascade layer, and $t = 1, \ldots, T_l$. $p_{i,c}^{t,l}$ denotes the probability of class c for data point x_i produced by the t-th forest at the l-th cascade layer. Notably, the class c corresponds to hash code $\{-1, 1\}$ and $p_{i,1}^{t,l} + p_{i,-1}^{t,l} = 1$. When training the cascade forest, the class distribution forms a class vector, which is then concatenated with the feature vector to serve as input in the next level of the cascade. Therefore, the input vector X_l at each cascade layer should be:

$$X_l = \begin{cases} F(x_i), & l = 1 \\ \left(F(x_i), p_{i,c}^{t,l}, t = 1, \ldots, T_l\right), & l \geq 2, \end{cases} \tag{15}$$

where $\left(F(x_i), p_{i,c}^{t,l}, t = 1, \ldots, T_l\right)$ denotes the concatenated feature vector and class vector. Thus, the training of each cascade layer is supervised by the hash codes generated in step-1. At the final cascade layer, the average value of the class probability distribution is computed:

$$p_{i,c}^{ave,L} = T_L^{-1} \sum_{t=1}^{T_L} p_{i,c}^{t,L} \tag{16}$$

The final predicted hash code is determined by the value of $p_{i,c}^{ave,L}$:

$$z_{r,i} = \begin{cases} 1, & p_{i,1}^{ave,L} > p_{i,-1}^{ave,L} \\ -1, & p_{i,-1}^{ave,L} > p_{i,1}^{ave,L}. \end{cases} \tag{17}$$

We summarize the DFH model in Algorithm 1.

Algorithm 1 Deep forest hashing.

Input: Training data points:$\{X_1, X_2, \ldots, X_n\}$; Affinity matrix: Y; Bit length: k

Output: Hash functions: $H = h_1, h_2, \ldots, h_k$

1: Use different sized sliding windows to extract multi-grained features.
2: Construct blocks according to affinity matrix Y.
3: **for** r= 1, ..., k **do**
4: Step-1: //initial binary code inference
5: compute the manifold similarity in Eqs. (1) and (2) as the edge weight to construct a weighted graph.
6: apply graph cut on each block to obtain binary codes of the r-th bit.
7: Step-2: //deep forest hash function learning
8: train the cascade forest to obtain the hash function h_r
9: update the binary codes of the r-th bit by the output of hash function h_r
10:**end**

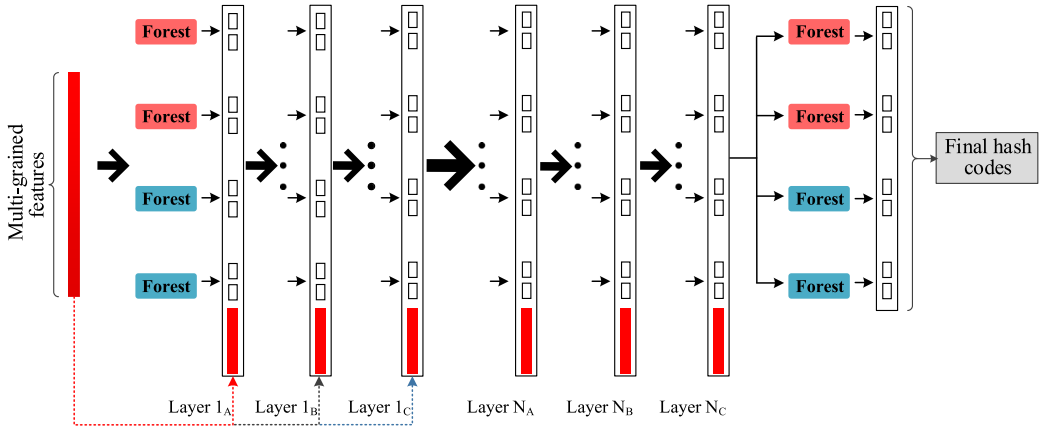


Fig. 4. Illustration of the cascade forest structure. Here, two types of forests are included in each layer. The forest in the red box denotes the random forest, and the forest in the blue box denotes the completely random tree forest. Any classifier can be used as a base classifier in the cascade layers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

4. Analysis

In this section, we provide an analysis of the time complexity of the DFH algorithm and the components of the multi-grained feature extraction, initial binary code inference and deep forest hash function learning in the DFH model.

4.1. Time complexity of DFH

We analyze the time complexity of DFH during the training phase and query phase. Suppose there are n data points in C categories. Each data point has d dimensions. We use e-grained $(g_1, g_2, \ldots, g_e)$ sliding windows to scan the data at a stride of s. There are T_s forests in the multi-grained scanning layer and N_{st} trees in each forest. There are L layers in a cascade forest, T_c forests in each cascade layer, and N_{ct} trees in each forest. In addition, the length of the hash code is k.

During the training phase, we first extract multi-grained features of the data points. A sliding window with a size of $g_j \times g_j$ could produce $(\frac{\sqrt{d}-g_j+1}{s})^2$ instances. The time complexity of one-grained scanning is $O(T_s * N_{st} * {g_j}^2 * (\frac{\sqrt{d}-g_j+1}{s})^2 * \log(\frac{\sqrt{d}-g_j+1}{s})^2)$. The time complexity of multi-grained scanning is $O(\sum_{j=1}^{e} T_s * N_{st} * {g_j}^2 * (\frac{\sqrt{d}-g_j+1}{s})^2 * \log(\frac{\sqrt{d}-g_j+1}{s})^2)$. In step-1 of the initial binary code inference, we divide the data points into C blocks. In each block, we construct a graph. The time complexity of constructing a graph is $O((\frac{n}{C})^2)$. The time complexity of constructing C blocks is $O(\frac{n^2}{C})$. In step-2 of the deep forest hash function learning, the time complexity of building a decision tree is $O(d^*n^*\log(n))$. The time complexity of building a cascade forest is $O(L^*T_c^*N_{ct}^*d^*n^*\log(n))$. The time complexity of generating one-bit hash code is $O(\frac{n^2}{C} + L * T_c * N_{ct} * d * n * \log(n))$. Therefore, the total time complexity of DFH during the training phase is $O(\sum_{j=1}^{e} T_s * N_{st} * {g_j}^2 * (\frac{\sqrt{d}-g_j+1}{s})^2 * \log(\frac{\sqrt{d}-g_j+1}{s})^2 + k * (\frac{n^2}{C} + L * T_c * N_{ct} * d * n * \log(n)))$.

During the query phase, we directly use cascade forests to map the data points to hash codes. The time complexity of a decision tree for classification is $O(n^*\log n)$. Therefore, the time complexity of the DFH during the query phase is $O(k^*L^*T_c^*N_{ct}^*n^*\log(n))$.

4.2. Multi-grained feature extraction

We use multi-grained feature extraction for three reasons. First, instead of using hand-crafted features, DFH uses multi-grained sliding windows to extract multi-grained features, which is dependent on the hash function learning procedure, and the resulting features are optimally compatible with the hashing procedure. Second, although convolutional neural networks can extract deep features, which is dependent on hash function learning, we focus on tree-based and forest-based models in this paper because of the deficiencies of deep neural networks previously discussed. The multi-grained feature extraction is actually implemented by a one level cascade forest with sliding features as input. Third, multi-grained feature extraction, an important component in DFH, helps significantly improve performance. To verify the importance of multi-grained feature extraction, we designed a variant of DFH called DFH-ca, which contains only cascade forests without multi-grained scanning layers. DFH-ca is discussed in detail in the experimental section.

4.3. Initial binary code inference

In step-1, i.e., initial binary code inference, we divide all data points into several blocks to manage large-scale data. In each block, we use the graph cut algorithm to initialize one-bit code for all data points in the block. The graph cut algorithm is a typical and efficient solution to binary or multilabel assignment problems in computer vision based on the max-flow or min-cut and requires no continuous relaxation. The graph cut algorithm is mainly used to minimize energy in the following form:

$$\varepsilon(I) = \sum_i \varepsilon_u(I_i) + \sum_{(i,j)} \varepsilon_p(I_i, I_j), \tag{18}$$

where I denotes the label of the data point in the block, e.g., -1 or 1 in our model. ε_u is a data term that depends on a data point, and ε_p is a smoothness term that depends on two data points. As shown in Eq. (13a), the initial binary code inference is the energy form.

In addition, we introduce a supervised manifold technique to compute the manifold similarity of the data as the edge weights to construct a graph. Thus, we can not only preserve the similarity of data points with the same label in the Hamming space but also preserve the similarity of data points with shorter distances and enlarge the difference of those with larger distances.

4.4. Deep forest hash function learning

In step-2, i.e., deep forest hash function learning, we apply cascade forests as hash functions. We will analyze this novel deep model from three perspectives: hyperparameter setting, diversity enhancement and the reasons why to go deep.

The flexible hyperparameter setting. The hyperparameters in the cascade forests are very simple compared with those in deep neural networks. The model most similar to DFH is FastH. Here, we provide a brief comparison of the parameters of the boosted tree (hash function in FastH), the convolutional neural network (a member of the deep neural network family) and the cascade forest (hash function in DFH) in Table 2.

The diversity in deep forest hash function. The cascade forest is designed to be a homogeneous ensemble model that combines the same type of base learners. A good ensemble model has two conditions, i.e., good and different; thus, the base learners must be good in precision, and diversity must exist between them.

Random forests and completely random tree forests are used in each cascade layer to encourage diversity in cascade forest. The base learners are also ensemble classifiers (ensemble of decision trees), therefore a cascade forest is an ensemble of ensembles. A random forest randomly select $\sqrt{d}$ features, and then select the best Gini value for the split, while a completely random tree forest randomly selects a feature for the split. The two types of classifiers enhance diversity in the ensemble model through different feature sampling and split selection methods.

Benefits of deep models. To the best of our knowledge, deep learning models are powerful due to the following three reasons: layer-by-layer processing, feature transformation and sufficient model complexity.

First, regarding layer-by-layer processing, a cascade forest is a model with cascade structure in which each layer of the cascade receives feature information processed by its preceding layer and outputs its processing results to the next layer.

Second, regarding feature transformation, each forest in a cascade layer generates an estimate of the class distribution by counting the percentage of different classes of examples at the leaf node where the concerned example falls into and then averages across all trees in the same forest. The class distribution forms a class vector, which is then concatenated with the original vector as input to the next level of cascade. The concatenated vector is a re-representation of the original input. Each layer can be considered

Table 2
Comparison of parameters in boosted tree (the hash function in FastH), convolutional neural network (the member of deep neural network family) and cascade forest (hash function in DFH).

Boosted tree	Convolutional neural network	Cascade forest
No. of trees	**Type of activation functions**	**Type of forests:**
Tree depth	Sigmoid, ReLU, tanh, linear, etc.	Completely random tree forest, random forest, etc.
Boosting iterations	**Architecture configurations:**	**Forest in cascade:**
	No. of hidden layers	No. of forests
	No. of nodes in hidden layer	No. of trees in each forest
	No. of feature maps	Tree depth
	Kernel size	
	Optimization configurations:	
	Learning rate	
	Dropout	
	Momentum	
	L1/L2 weight regularization penalty	
	Weight initialization	
	Batch size	

trained with supervised label information. Moreover, the representation learning ability of cascade forests can be further enhanced by multi-grained scanning when the inputs have high dimensionality.

Third, regarding sufficient model complexity, a cascade forest is an ensemble of ensembles of decision tree forests. Thus, several cascade layers exist in a cascade forest, and several forests exist in each cascade layer, and several trees exist in each forest. The number of forests in each cascade layer and the number of trees in each forest are main hyperparameters. The ensemble of trees and forests guarantees that a cascade forest has sufficient model complexity.

Furthermore, instead of simply exploring deeper, the cascade forest can adaptively determine its model complexity. Thus, during the training phase, the training set is divided into a growing set and an estimating set. The growing set is used to grow the cascade, and the estimating set is used to estimate the performance. If growing a new layer does not improve performance, the cascade layers stop growing.

5. Experiments

5.1. Dataset and configuration

We conduct a series of experiments to evaluate DFH in image retrieval tasks with three benchmarks: MNIST[1], CIFAR-10[2] and NUS-WIDE[3]. The MNIST dataset consists of 28×28 grayscale handwritten digit images of 0 to 9 with 7000 examples per class and a total of 70,000 images. The CIFAR-10 dataset consists of 60,000 32×32 color images in 10 categories and 6000 images per category. The NUS-WIDE, a dataset of nearly 270,000 images collected from the real-world web, is a multilabel dataset. Each image is annotated with one or multiple labels from 81 categories. We use only the images associated with the 21 most frequent categories in which, each label is associated with at least 5000 images.

The python implementation of our model DFH is available at https://github.com/Jack-Chow-my/DFH.

We compare our method with several hashing methods. These methods can be categorized into four classes:

- Data-independent hashing: LSH [6].
- Unsupervised data-dependent hashing: SH [11] and ITQ [3].
- Supervised data-dependent hashing: FastH [16,17] and SDH [18].
- Deep neural network-based hashing: CNNH [29].

Regarding the settings of the datasets, following [28,29,34], in MNIST and CIFAR-10, we randomly select 1000 images (100 images per category) as the query set. Regarding the unsupervised methods, we use the remaining images as training samples. Regarding the supervised methods, we randomly select 5000 images (500 images per class) from the remaining images as the training set. In NUS-WIDE, we randomly select 2100 images from 21 categories (100 images per category) as the query set. Regarding the unsupervised methods, the rest images of the selected 21 categories are used as the training set. Regarding the supervised methods, we sample 10,500 images (500 images per class) from the 21 selected categories to form the training set.

Regarding the non-deep hashing methods, we represent each image in MNIST by a 784-dimensional grayscale vector; we represent each image in CIFAR-10 by a 512-dimensional GIST vector; we represent each image in NUS-WIDE by a 1134-dimensional low-level feature vector, including a 64-D color histogram, 144-D color correlogram, 73-D edge direction histogram, 128-D wavelet texture, 225-D blockwise color moments and 500-D SIFT features. Regarding the deep hashing models, we use the raw images directly as input.

As most existing hashing methods, we use the mean average precision (MAP) to compare the performance of DFH with that of other methods.

5.2. Comparison results of image retrieval

In this section, we compare DFH with four other classes of hashing methods, including LSH, SH, ITQ, FastH and SDH, which are non-deep hashing methods, and CNNH, which is a deep neural network-based hashing method.

The MAP results are shown in Tables 4, 5 and 6. DFH is observed to outperform CNNH, FastH, SDH, ITQ, SH and LSH on MNIST, CIFAR-10 and NUS-WIDE, especially FastH using a shallow level decision tree and CNNH using a deep convolutional network, proving the effectiveness and efficiency of deep forest applications in hashing. In our model, in MNIST, we use 2 forests with 20 trees in each forest in a multi-grained scanning layer and 4 forests with 100 trees in each forest in a cascade forest layer. In CIFAR-10, we use 2 forests with 20 trees in each forest in a multi-grained scanning layer and 4 forests with 200 trees in each forest in a cascade forest layer. In NUS-WIDE, we use 2 forests with 20 trees in each forest in a multi-grained scanning layer and 4 forests with 400 trees in each forest in a cascade forest layer. The default hyperparameters in gcForest are 2 forests with 500 trees in each forest in the multi-grained scanning layer and 8 forests with 500 trees in

[1] http://yann.lecun.com/exdb/mnist.
[2] http://www.cs.toronto.edu/~kriz/cifar.html.
[3] http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm.

Table 3
Configuration of DFH on MNIST, CIFAR-10 and NUS-WIDE.

	Datasets	MINIST	CIFAR-10	NUS-WIDE
Multi-grained scanning	Window sizes	{7 × 7} {10 × 10} {13 × 13}	{8 × 8} {11 × 11} {16 × 16}	{39 × 39} {56 × 56} {79 × 79}
	Stride	2	2	2
	No. of random forests	1	1	1
	No. of completely-random tree forests	1	1	1
	No. of trees in each forest	20	20	20
	Max depth	10	10	10
Cascade forest	No. of random forests	2	2	2
	No. of completely-random tree forests	2	2	2
	No. of trees in each forest	100	200	400

Table 4
MAP results of image retrieval on MNIST.

	Method	MNIST(MAP)			
		12-bits	24-bits	32-bits	48-bits
Deep forest based hashing	**DFH(ours)**	**0.979**	**0.981**	**0.984**	**0.985**
Deep neural network-based hashing	CNNH [29]	0.957	0.963	0.956	0.960
Supervised data-dependent hashing	FastH [16,17]	0.937	0.946	0.947	0.950
	SDH [18]	0.937	0.944	0.948	0.957
Unsupervised data-dependent hashing	ITQ [3]	0.388	0.436	0.422	0.429
	SH [11]	0.265	0.267	0.259	0.250
Data-independent hashing	LSH [6]	0.187	0.209	0.235	0.243

Table 5
MAP results of image retrieval on CIFAR-10.

	Method	CIFAR-10(MAP)			
		12-bits	24-bits	32-bits	48-bits
Deep forest based hashing	**DFH(ours)**	**0.457**	**0.513**	**0.524**	**0.559**
Deep neural network-based hashing	CNNH [29]	0.439	0.476	0.472	0.489
Supervised data-dependent hashing	FastH [16,17]	0.305	0.349	0.369	0.384
	SDH [18]	0.285	0.392	0.341	0.356
Unsupervised data-dependent hashing	ITQ [3]	0.162	0.169	0.172	0.175
	SH [11]	0.127	0.128	0.126	0.129
Data-independent hashing	LSH [6]	0.121	0.126	0.120	0.120

Table 6
MAP results of image retrieval on NUS-WIDE.

	Method	NUS-WIDE(MAP)			
		12-bits	24-bits	32-bits	48-bits
Deep forest based hashing	**DFH(ours)**	**0.622**	**0.659**	**0.674**	**0.695**
Deep neural network-based hashing	CNNH [29]	0.611	0.618	0.625	0.608
Supervised data-dependent hashing	FastH [16,17]	0.621	0.650	0.665	0.687
	SDH [18]	0.568	0.600	0.608	0.637
Unsupervised data-dependent hashing	ITQ [3]	0.452	0.468	0.472	0.477
	SH [11]	0.454	0.406	0.405	0.400
Data-independent hashing	LSH [6]	0.403	0.421	0.426	0.441

each forest in the cascade forest layer. The hyperparameter settings are shown in Table 3.

Here, DFH, CNNH and FastH are similar models. CNNH adopts a convolutional neural network to perform hashing, and FastH employs boosted trees as hash functions. Both DFH and CNNH are deep models. Both DFH and FastH are tree-based models. On the one hand, a deep model can address complex learning tasks; on the other hand, deep models are apt to have many hyperparameters. Tree-based models usually have fewer hyperparameters and require less training time. The similarities and differences among DFH, CNNH and FastH are shown in Table 7.

In addition, DFH can achieve a relatively good performance with a very short hash code as follows: the MAP of DFH with 12-bits can reach 0.979 in MNIST, 0.457 in CIFAR-10 and 0.622 in NUS-WIDE, which is very important for storage cost and retrieval speed. Moreover, the number of forests and trees are much lower than the default settings in gcForest.

5.3. Influence of multi-grained feature extraction

The multi-grained scanning in DFH helps extract multi-grained features from the raw data. To verify the importance of the multi-grained feature extraction, we designed a variant of DFH called DFH-ca, which contains only cascade forests without multi-grained scanning layers. For simplicity, we use 2 forests with 20 trees in each forest in a multi-grained scanning layer in DFH and DFH-ca in all three datasets. In MNIST, we use 4 forests with 100 trees in each forest in a cascade forest layer in DFH and DFH-ca. In CIFAR-10, we use 4 forests with 200 trees in each forest in a cascade forest layer in DFH and DFH-ca. In NUS-WIDE, we use 4 forests with

Table 7
The similarities and differences of DFH, CNNH and FastH.

	Fast training speed	Fewer hyperparameters	Deep model	Interpretability
DFH(ours)	✓	✓	✓	✓
CNNH			✓	
FastH	✓	✓		✓

Table 8
Configuration of DFH and DFH-ca on MNIST, CIFAR-10 and NUS-WDIE.

	Datasets	MNIST		CIFAR-10		NUS-WIDE	
	Methods	DFH	DFH-ca	DFH	DFH-ca	DFH	DFH-ca
Multi-grained scanning	Window sizes	{7 × 7} {10 × 10} {13 × 13}		{8 × 8} {11 × 11} {16 × 16}		{39 × 39} {56 × 56} {79 × 79}	
	Stride	2	None	2	None	2	None
	No. of random forests	1		1		1	
	No. of completely random tree forests	1		1		1	
	No. of trees in each forest	20		20		20	
	Max depth	10		10		10	
Cascade forest	No. of random forests	2	2	2	2	2	2
	No. of completely random tree forests	2	2	2	2	2	2
	No. of trees in each forest	100	100	200	200	400	400

Table 9
MAP results of DFH, DFH-ca and CNNH on MNIST, CIFAR-10 and NUS-WDIE.

	12-bits	24-bits	32-bits	48-bits
		MNIST		
DFH-ca	0.884	0.961	0.964	0.972
CNNH	0.957	0.963	0.956	0.960
DFH	0.979	0.981	0.984	0.985
		CIFAR-10		
DFH-ca	0.412	0.465	0.491	0.520
CNNH	0.439	0.476	0.472	0.489
DFH	0.457	0.513	0.524	0.559
		NUS-WIDE		
DFH-ca	0.567	0.579	0.586	0.597
CNNH	0.623	0.657	0.672	0.608
DFH	0.622	0.659	0.674	0.695

400 trees in each forest in a cascade forest layer in DFH and DFH-ca. The hyperparameter settings are shown in Table 8, and the MAP results of MNIST, CIFAR-10 and NUS-WIDE are shown in Table 9.

Clearly, the multi-grained feature extraction stage significantly helps improve performance. In all three datasets, the MAP results of DFH are higher than those of DFH-ca. In addition to the comparison between DFH and DFH-ca, we include CNNH to verify the contributions of the multi-grained feature extraction to DFH. In MNIST and CIFAR-10, the MAP results of both DFH and CNNH with 12-bits and 24-bits hash codes are higher than those of DFH-ca, and the performance of DFH is higher than that of CNNH, suggesting that with the help of multi-grained feature extraction, DFH is superior to CNNH. In NUS-WIDE, the influence of the multi-grained feature extraction is more prominent. The MAP results of CNNH are higher than those of DFH-ca, but the results of DFH are higher than those of CNNH.

5.4. More forests and trees

In contrast to deep neural network-based hashing, DFH has fewer hyperparameters. The number of forests in each cascade layer and the number of trees in each forest are the main hyperparameters in DFH. To verify the influence of the number of forests and trees, three DFH models with higher hyperparameters called DFH_tree+ (DFH with more trees), DFH_forest+ (DFH with more forests), and DFH++ (DFH with more trees and forests) are designed. For simplicity, we use 2 forests with 20 trees in each forest in the multi-grained scanning layer in all four models. In DFH, we use 4 forests with 20 trees in each forest in a cascade forest layer. In DFH_tree+, we use 4 forests with 100 trees in each forest in a cascade forest layer. In DFH_forest+, we use 8 forests with 20 trees in each forest in a cascade forest layer. In DFH++, we use 8 forests with 100 trees in each forest in a cascade forest layer. The hyperparameter settings are shown in Table 10, and the MAP results of MNIST, CIFAR-10 and NUS-WIDE are shown in Table 11.

Evidently, more forests and more trees improve the learning ability of DFH, and increasing only the number of trees appears to be more effective than increasing only the number of forests. Furthermore, the hyperparameter settings are very flexible, which is not as difficult in deep neural network-based hashing methods.

5.5. Different base classifiers

Because of the new API offered by gcForest [35], we can change the base classifiers in each cascade layer. We consider XGBoost [40], GBDT [41], AdaBoost [42], random forest [38] and completely random tree forests [39] as the base classifiers in DFH. These classifiers are widely employed in industry and various data science competitions. Any classifier can be added to the cascade forest layers to construct a deep ensemble model. Users can choose different base classifiers according to their tasks and computing devices. For simplicity, we use only 20 trees in each forest for each base classifier. The configuration of DFH with different base classifiers is shown in Table 12.

Considering that the completely random tree forest randomly selects a feature for the split, which can help encourage the generalization of the model, we combine the completely random tree forest with other classifiers to build the cascade layer. The MAP results of DFH with different base classifiers in MNIST, CIFAR-10 and NUS-WIDE are summarized in Table 13 and Figs. 5, 6 and 7

In this experiment, we further verify the flexible settings of DFH; however, we do not attempt to determine which classifier works best, which is beyond the scope of this paper. Considering the scale of the image dataset to be retrieved, the computational resources available and the training time requirements, the different base classifiers and number of base classifiers are determined. The base classifiers we use are all tree-based models, but users can

Table 10
Configuration of DFH, DFH_tree+, DFH_forest+ and DFH++ on MNIST, CIFAR-10 and NUS-WIDE.

		DFH	DFH_tree+	DFH_forest+	DFH++
Multi-grained scanning	Window sizes	MNIST: {7 × 7, 10 × 10, 13 × 13} CIFAR-10: {8 × 8, 11 × 11, 16 × 16} NUS-WIDE: {39 × 39, 56 × 56, 79 × 79}			
	Stride	2	2	2	2
	No. random forests	1	1	1	1
	No. completely-random tree forests	1	1	1	1
	No. trees in each forests	20	20	20	20
	Max depth	10	10	10	10
Cascade forest	No. random forests	2	2	4	4
	No. completely-random tree forests	2	2	4	4
	No. trees in each forest	20	100	20	100

Table 11
The MAP results of DFH, DFH_tree+, DFH_forest+ and DFH++ on MNIST, CIFAR-10 and NUS-WIDE.

	12-bits	24-bits	32-bits	48-bits
	MNIST			
DFH	0.913	0.978	0.982	0.982
DFH_tree+	0.979	0.981	0.984	0.985
DFH_forest+	0.975	0.981	0.982	0.984
DFH++	0.980	0.983	0.984	0.985
	CIFAR-10			
DFH	0.450	0.495	0.505	0.511
DFH_tree+	0.452	0.499	0.513	0.547
DFH_forest+	0.451	0.497	0.510	0.536
DFH++	0.456	0.504	0.521	0.550
	NUS-WIDE			
DFH	0.483	0.580	0.580	0.591
DFH_tree+	0.542	0.586	0.592	0.621
DFH_forest+	0.534	0.583	0.589	0.614
DFH++	0.547	0.590	0.595	0.632

Table 13
MAP results of DFH with different base classifiers on MNIST, CIFAR-10 and NUS-WIDE.

	12-bits	24-bits	32-bits	48-bits
	MNIST			
DFH(RF)	0.913	0.978	0.982	0.982
DFH(GBDT)	0.900	0.979	0.981	0.982
DFH(XGBoost)	0.911	0.979	0.982	0.983
DFH(AdaBoost)	0.900	0.978	0.982	0.981
	CIFAR-10			
DFH(RF)	0.450	0.495	0.505	0.511
DFH(GBDT)	0.449	0.489	0.491	0.518
DFH(XGBoost)	0.468	0.516	0.502	0.523
DFH(AdaBoost)	0.444	0.479	0.485	0.529
	NUS-WIDE			
DFH(RF)	0.483	0.580	0.580	0.591
DFH(GBDT)	0.478	0.548	0.536	0.543
DFH(XGBoost)	0.489	0.491	0.521	0.588
DFH(AdaBoost)	0.493	0.495	0.498	0.579

attempt to use other base classifiers, such as SVM, SGD classifier and logistic regression.

5.6. More bits

We increase the length of the hash codes to 128 bits for image retrieval tasks on MNIST, CIFAR-10 and NUS-WIDE. We adopt the same configuration as the DFH in Section 5.2. The hyperparameter settings are shown in Table 3. The MAP results are shown in Table 14.

The results show that DFH can be trained with a longer bit length, and the complexity is only linearly increased with the bit length.

5.7. More experiments using other types of datasets

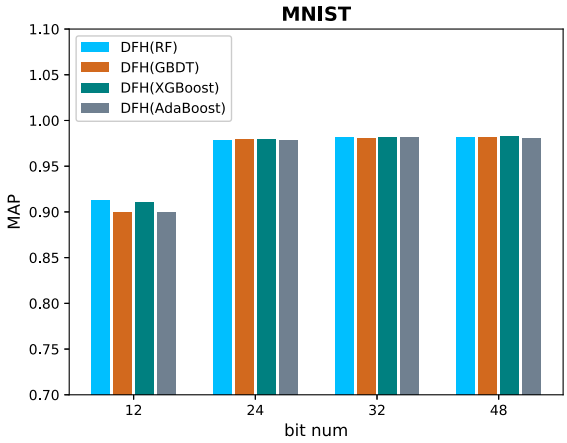
We include 2 other different datasets to conduct experiments. We validate our DFH model using UCI-datasets, LETTER[4] and YEAST[5]. LETTER consists of 16,000 training data points and 4000 test data points with 16-dimensional features. YEAST consists of 1038 training data points and 446 test data points with 8-dimensional features. Considering the small number of features of the two datasets, the multi-grained feature extraction is abandoned. The hyperparameter settings are shown in Table 15, and the MAP results are shown in Table 16. The results show that DFH can

[4] http://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recognition.data.

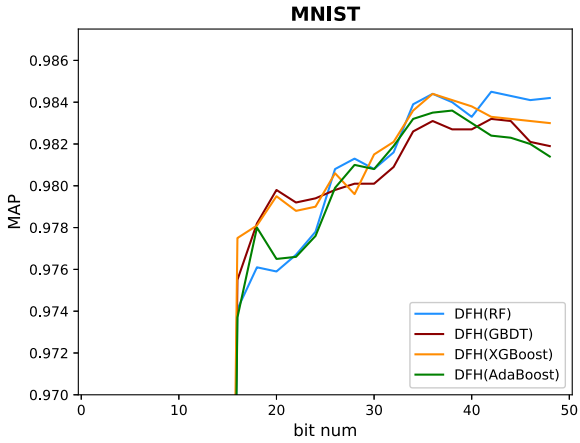
[5] http://archive.ics.uci.edu/ml/machine-learning-databases/yeast/yeast.data.

Table 12
Configuration of DFH with different base classifiers on MNIST, CIFAR-10 and NUS-WDIE.

Multi-grained scanning				
Window sizes	MNIST: {7 × 7, 10 × 10, 13 × 13} CIFAR-10: {8 × 8, 11 × 11, 16 × 16} NUS-WIDE: {39 × 39, 56 × 56, 79 × 79}			
Stride	2	Pooling method		average
No. of random forests	1	No. of completely random tree forests		1
No. of trees in each forest	20	Max depth		10
Cascade forest (with the completely random tree forest as one of the base classifiers)				
Base classifier	No. of forests	Max depth	No. of trees in each forest	Learning rate
XGBoost	2	5	20	0.1
GBDT	2	5	20	0.1
AdaBoost	2	–	20	–
Random forest	2	null	20	–
Completely random tree forest	**2**	**null**	**20**	**–**

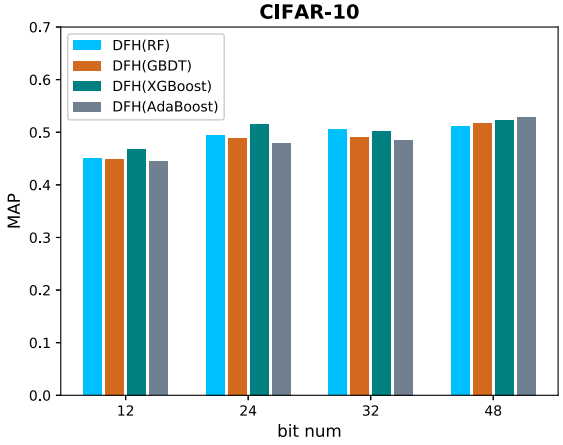


(a) MAP results of different base classifiers with hash codes from 12 to 48-bits

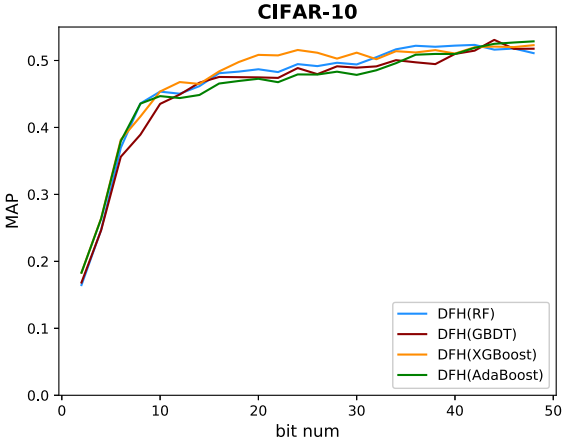


(b) MAP curves of different base classifiers

Fig. 5. Results of DFH with different base classifiers on MNIST dataset. The MAP results of DFH with different base classifiers are very close on the first few bits; we show the results of the last few bits in (b).

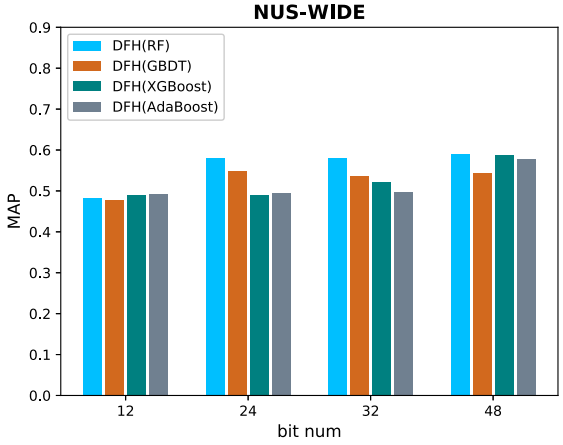


(a) MAP results of different base classifiers with hash codes from 12 to 48-bits

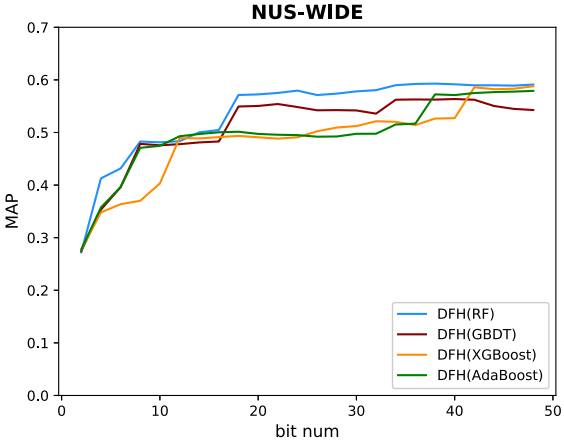


(b) MAP curves of different base classifiers

Fig. 6. Results of DFH with different base classifiers on CIFAR-10 dataset.



(a) MAP results of different base classifiers with hash codes from 12 to 48-bits



(b) MAP curves of different base classifiers

Fig. 7. Results of DFH with different base classifiers on NUS-WIDE dataset.

Table 14
MAP results of DFH with more bits on MNIST, CIFAR-10 and NUS-WDIE.

	Bits	MNIST	CIFAR-10	NUS-WIDE
DFH	56-bits	0.985	0.561	0.697
	64-bits	0.984	0.565	0.699
	72-bits	0.984	0.563	0.697
	96-bits	0.985	0.571	0.706
	128-bits	0.985	0.573	0.712

Table 15
Configuration of DFH on LETTER and YEAST.

Cascade forest	No. random forests	2
	No. completely-random tree forests	2
	No. trees in each forest	100

Table 16
MAP results of DFH on LETTER and YEAST.

Bits	12-bits	24-bits	32-bits	48-bits
LETTER	0.670	0.908	0.932	0.942
YEAST	0.577	0.617	0.640	0.643

be efficiently extended to other types of datasets for information retrieval.

6. Conclusion

In this work, we propose deep forest hashing (DFH) to learn shorter binary code representations to realize effective and efficient image retrieval. It is a two-stage hashing method by initial binary code inference and deep forest hash function learning. We consider three types of similarity metrics in our hash learning formulation to preserve the semantic similarity and manifold similarity of the data points in the Hamming space. We utilize a supervised manifold method to compute the manifold similarity of the data points. Compared with deep neural network-based hashing methods, DFH has fewer hyperparameters, faster training speed and easier theoretical analysis. Compared with other existing tree-based and forest-based hashing methods, the DFH model has deeper cascade construction that leads to higher model complexity and superior learning ability, which is verified experimentally. Compared with other non-deep neural network hashing methods, DFH extracts multi-grained features from the raw data, and the feature extraction phase is dependent on the hash function learning stage, which helps learn better hash functions. The experiments based on three benchmarks demonstrate that DFH obtains superior results over the comparison hashing methods and that DFH has very flexible hyperparameter and base classifier setting. To the best of our knowledge, DFH is the first hashing approach in image retrieval using the deep forest. We apply a two-step learning strategy and bitwise optimization for hashing, indicating that each hash function (cascade forest) generates a one-bit hash code for all data points. As the length of the hash code increases, the capacity of the model becomes too large. The design of a deep forest that can simultaneously learn image representation and cascade forest could further improve our deep forest hashing method. In addition, a one-step learning strategy may be applied to learn one cascade forest to generate all bits of hash codes for all data points. Therefore, the length of the hash codes may be adaptively determined without presetting when expanding a new cascade layer but without a significant performance gain.

Acknowledgments

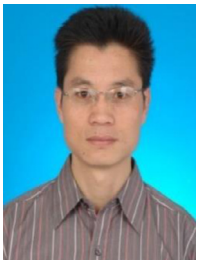
The authors would like to thank the anonymous reviewers for their help. This work was supported by the National Natural Science Foundation of China (Grant no. 61672120) and Chongqing Postgraduate Research and Innovation Project (Grant no. CYS17224).

References

[1] J. Wang, S. Zhang, J. Song, N. Sebe, H.T. Shen, A survey on learning to hash, IEEE Trans. Pattern Anal. Mach. Intell. 40 (4) (2018) 769–790.
[2] P. Li, A. Shrivastava, J.L. Moore, A.C. König, Hashing algorithms for large-scale learning, NIPS (2011) 2672–2680.
[3] Y. Gong, S. Lazebnik, Iterative quantization: a procrustean approach to learning binary codes for large-scale image retrieval, IEEE Trans. Pattern Anal. Mach. Intell. 35 (12) (2013) 2916–2929.
[4] W. Kong, W.J. Li, Isotropic hashing, NIPS (2012) 1646–1654.
[5] W. Liu, J. Wang, S. Kumar, S. Chang, Hashing with graphs, ICML (2011) 1–8.
[6] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, VLDB (1999) 518–529.
[7] J. Ji, J. Li, S. Yan, B. Zhang, Q. Tian, Super-bit locality-sensitive hashing, NIPS (2012) 108–116.
[8] Y. Mu, S. Yan, Non-metric locality-sensitive hashing, AAAI (2010) 539–544.
[9] B. Kulis, K. Grauman, Kernelized locality-sensitive hashing for scalable image search, ICCV (2009) 2130–2137.
[10] A. Shrivastava, P. Li, Densifying one permutation hashing via rotation for fast near neighbor search, ICML (2014) 557–565.
[11] Y. Weiss, A. Torralba, R. Fergus, Spectral hashing, NIPS (2009) 1753–1760.
[12] W. Liu, C. Mu, S. Kumar, S.F. Chang, Discrete graph hashing, NIPS (2014) 3419–3427.
[13] Q.Y. Jiang, W.J. Li, Scalable graph hashing with feature transformation, IJCAI (2015) 2248–2254.
[14] W. Liu, J. Wang, R. Ji, Y.G. Jiang, S.F. Chang, Supervised hashing with kernels, CVPR (2012) 2074–2081.
[15] G. Lin, C. Shen, D. Suter, A. van den Hengel, A general two-step approach to learning-based hashing, CVPR (2013) 2552–2559.
[16] G. Lin, C. Shen, Q. Shi, A. van den Hengel, D. Suter, Fast supervised hashing with decision trees for high-dimensional data, CVPR (2014) 1963–1970.
[17] G. Lin, C. Shen, A. van den Hengel, Supervised hashing using graph cuts and boosted decision trees, IEEE Trans. Pattern Anal. Mach. Intell. 37 (11) (2015) 2317–2331.
[18] F. Shen, C. Shen, W. Liu, H.T. Shen, Supervised discrete hashing, CVPR (2015) 37–45.
[19] J. Gui, T. Liu, Z. Sun, D. Tao, T. Tan, Fast supervised discrete hashing, IEEE Trans. Pattern Anal. Mach. Intell. 40 (2) (2018) 490–496.
[20] Y. Cui, J. Jiang, Z. Lai, Z. Hu, W. Wong, Supervised discrete discriminant hashing for image retrieval, Pattern Recognit. 78 (2018) 79–90.
[21] J. Wang, W. Liu, A.X. Sun, Y.G. Jiang, Learning hash codes with listwise supervision, ICCV (2013) 3032–3039.
[22] L. Liu, L. Shao, F. Shen, M. Yu, Discretely coding semantic rank orders for supervised image hashing, CVPR (2017) 5140–5149.
[23] J. Song, L. Gao, L. Liu, X. Zhu, N. Sebe, Quantization-based hashing: a general framework for scalable image and video retrieval, Pattern Recognit. 75 (2018) 175–187.
[24] R. Salakhutdinov, G.E. Hinton, Semantic hashing, Int. J. Approx. Reason. 50 (7) (2009) 969–978.
[25] V.E. Liong, J. Lu, G. Wang, P. Moulin, J. Zhou, Deep hashing for compact binary codes learning, CVPR (2015) 2475–2483.
[26] K. Lin, H.F. Yang, J.H. Hsiao, C.S. Chen, Deep learning of binary hash codes for fast image retrieval, CVPRW (2015) 27–35.
[27] J. Song, L. Gao, F. Zou, Y. Yan, N. Sebe, Deep and fast: deep learning hashing with semi-supervised graph construction, Pattern Recognit. 55 (2015) 101–108.
[28] H. Lai, Y. Pan, Y. Liu, S. Yan, Simultaneous feature learning and hash coding with deep neural networks, CVPR (2015) 3270–3278.
[29] R. Xia, Y. Pan, H. Lai, C. Liu, S. Yan, Supervised hashing for image retrieval via image representation learning, AAAI (2014) 2156–2162.
[30] F. Shen, Y. Xu, L. Liu, Y. Yang, Z. Huang, H.T. Shen, Unsupervised deep hashing with similarity-adaptive and discrete optimization, IEEE Trans. Pattern Anal. Mach. Intell. 40 (12) (2018) 3034–3044.
[31] F. Zhao, Y. Huang, L. Wang, T. Tan, Deep semantic ranking based hashing for multi-label image retrieval, CVPR (2015) 1556–1564.
[32] J. Tang, Z. Li, X. Zhu, Supervised deep hashing for scalable face image retrieval, Pattern Recognit. (2018) 25–32.
[33] Q. Li, Z. Sun, R. He, T. Tan, Deep supervised discrete hashing, NIPS (2017) 2482–2491.
[34] W.J. Li, S. Wang, W.C. Kang, Feature learning based deep supervised hashing with pairwise labels, IJCAI (2016) 1711–1717.
[35] Z.H. Zhou, J. Feng, Deep forest: towards an alternative to deep neural networks, IJCAI (2017) 3553–3559.
[36] Q. Qiu, J. Lezama, A.M. Bronstein, G. Sapiro, Foresthash: semantic hashing with shallow random forests and tiny convolutional networks, ECCV (2018) 432–448.

[37] G. Yu, J. Yuan, Scalable forest hashing for fast similarity search, ICME (2014) 1–6.
[38] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[39] F.T. Liu, K.M. Ting, Y. Yu, Z.H. Zhou, Spectrum of variable-random trees, J. Artif. Intell. Res. 32 (2008) 355–384.
[40] T. Chen, C. Guestrin, Xgboost: a scalable tree boosting system, KDD (2016) 785–794.
[41] M.H. Friedman, Greedy function approximation: a gradient boosting machine, Ann. Stat. 29 (2000) 1189–1232.
[42] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1995) 119–139.

Meng Zhou, is currently a master degree candidate in the Chongqing Key Laboratory of Image Cognition, College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing. His current research interests mainly include hashing learning, image processing and ensemble learning.



Xianhua Zeng, is currently a professor with the Chongqing Key Laboratory of Image Cognition, College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing, China. He received his Ph.D. degree in Computer software and theory from Beijing Jiaotong University in 2009. And he was a Visiting Scholar in the University of Technology, Sydney, from Aug. 2013 to Aug. 2014. His main research interests include medical image processing, machine learning and data mining.

Aozhu Chen, is currently a master degree candidate in the Chongqing Key Laboratory of Image Cognition, College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing. Her current research interests mainly include manifold learning and image color perception.