Neuroinform
DOI 10.1007/s12021-016-9300-2

ORIGINAL ARTICLE

Neuron-Miner: An Advanced Tool for Morphological Search and Retrieval in Neuroscientific Image Databases

Sailesh Conjeti[1] · Sepideh Mesbah[1] · Mohammadreza Negahdar[4] · Philipp L. Rautenberg[2] · Shaoting Zhang[3] · Nassir Navab[1] · Amin Katouzian[4]

Abstract The steadily growing amounts of digital neuroscientific data demands for a reliable, systematic, and computationally effective retrieval algorithm. In this paper, we present Neuron-Miner, which is a tool for fast and accurate reference-based retrieval within neuron image databases. The proposed algorithm is established upon hashing (search and retrieval) technique by employing multiple unsupervised random trees, collectively called as Hashing Forests (HF). The HF are trained to parse the neuromorphological space hierarchically and preserve the inherent neuron neighborhoods while encoding with compact binary codewords. We further introduce the inverse-coding formulation within HF to effectively mitigate pairwise neuron similarity comparisons, thus allowing scalability to massive databases with little additional time overhead. The proposed hashing tool has superior approximation of the true neuromorphological neighborhood with better retrieval and ranking performance in comparison to existing generalized hashing methods. This is exhaustively validated by quantifying the results over 31266 neuron reconstructions from Neuromorpho.org dataset curated from 147 different archives. We envisage that finding and ranking similar neurons through reference-based querying *via* Neuron Miner would assist neuroscientists in objectively understanding the relationship between neuronal structure and function for applications in comparative anatomy or diagnosis.

Keywords Neuroscientific databases · Data mining · Hashing · Neuromorphological space · Random Forests

Introduction

Neuroscientists often analyze the 3D morphology of neurons to understand neuronal network connectivity and how neural information is processed for evaluating brain functionality (Costa et al. 2010). The size and diversity of neuroscientific databases have been rapidly increased over the past decade, resulting in deluge of publicly available datasets (especially 3D digitally reconstructed neurons), which consist of heterogeneous multi-center data acquired from different species, brain regions, and experimental settings (Ascoli et al. 2007; Rautenberg et al. 2014). Figure 1 demonstrates the evolution of the number of neurons in one such popular public database (Neuromorpho.org). This has motivated researchers, particularly computer scientists, to develop new search systems for image retrieval and processing over large-scale datasets for the purpose of neuron categorization and ultimately comprehension of its functionality.

In this paper, we propose a data-driven scheme search and retrieval for large neuron data-bases called hashing

Electronic supplementary material The online version of this article (doi:10.1007/s12021-016-9300-2) contains supplementary material, which is available to authorized users.

S. Conjeti and S. Mesbah contributed equally towards the work.

✉ Sailesh Conjeti
sailesh.conjeti@tum.de

[1] Chair for Computer Aided Medical Procedures, Technische Universität München, München, Germany

[2] Max Plank Digital Library, München, Germany

[3] Department of Computer Science, University of North Carolina at Charlotte, Charlotte, USA

[4] IBM Almaden Research Center, CA, USA

Published online: 07 May 2016

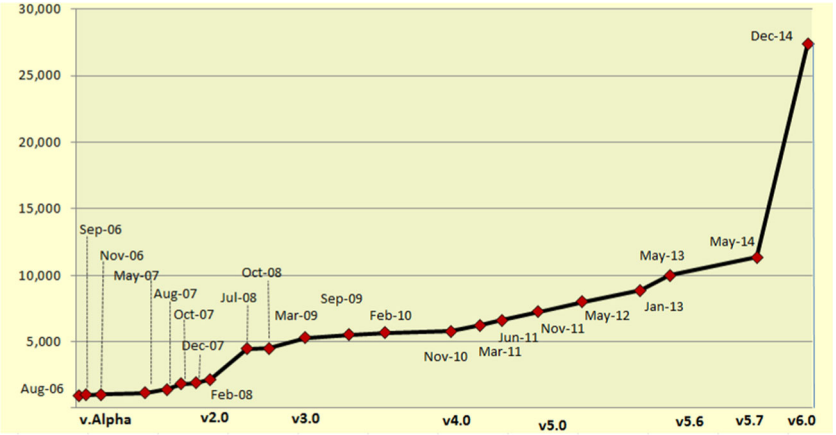

Fig. 1 Evolution of the number of neurons in the NeuroMorpho database in different released versions till December 2014

forests (HF). Further, to the best of our knowledge, this is the first work that focuses entirety on hashing in large neuroscientific databases. This paper also introduces the Neuron-Miner tool, which is a software package designed to facilitate easy visualization and analysis of neurons. This is a lightweight framework which works on any Android-supported device (mobile phones, tablets etc.) and interfaces with the central database using a client-server architecture. The neuron search and retrieval framework is integrated into Neuron-Miner software tool. This paper is an extension of our earlier work (Mesbah et al. 2015), where we introduced the HF for the first time and its application for search and retrieval in neuroscientific image database. In contrast to Mesbah et al. (2015), the current work has the following additional improvements: (1) improvised formulation of hashing forests with inclusion of oblique splitting functions and the concept of cluster validity, (2) exhaustive validations on retrieval performance, ranking, and time analysis over a larger and more heterogeneous dataset of 31266 neurons (additional 13,060 neurons in comparison to Mesbah et al. (2015)), and (3) additional emphasis on the software implementation aspects and design paradigms behind the Neuron-Miner tool, where HF is integrated. The hash table generation and the formulation for code comparison (forward and inverse coding) have been suitably adapted from Mesbah et al. (2015).

State-of-the Art

Query-based retrieval of relevant neurons within databases is important for comparative morphological analysis which are used to study age related changes (Rautenberg et al. 2009) and the relationship between structure and function (Costa et al. 2010). In this section, we provide an overview on generalized hashing methods proposed in the machine learning community and further delve into some recent works on neuron retrieval techniques in the neuroscience community.

Generalized Hashing Methods

Several efficient encoding and searching approaches have been proposed for retrieval in machine learning community. These include data independent methods (*e.g.* Locality Sensitive Hashing (LSH) (Gionis et al. 1999; Slaney and Casey 2008)), data-driven methods like Spectral Hashing (SH) (Weiss et al. 2012), and Self Taught Hashing (STH) (Zhang et al. 2010).

Locality Sensitive Hashing (LSH) The idea behind the LSH is that if two points are similar and close together, then upon randomized linear projections they will remain close to each other. The LSH generates binary encoding by partitioning feature space with randomly generated hyperplanes (Gionis et al. 1999; Slaney and Casey 2008). The LSH is a data independent method since it randomly generates hashing functions regardless of the data distribution.

Spectral Hashing (SH) Unlike LSH, instead of working on the original feature space, the SH generates hash codes using low dimensional representation obtained using Laplacian Eigenmaps (LEM) (Weiss et al. 2012). The SH is a data dependent hashing approach, which generates the hash codes by thresholding a subset of Laplacian eigenvectors at zero.

Self Taught Hashing (STH) Zhang et al. (2010) introduced self-learnt hashing functions (data dependent) by median-thresholding the low dimensional embedding from LEM and training a support vector machine (SVM) classifier as the hash function for encoding new input data. The STH focuses on the local similarity structure and for each query it finds the k-nearest neighbors using binary Hamming distance.

Data independent methods like LSH require large code words to efficiently parse the feature space as they rely

on their asymptotic nature for convergence (Yu and Yuan 2014). For similar code lengths, SH and STH can provide better retrieval as the hashing function is based on the data distribution. However, these data driven methods are mainly challenged by their restricted scalability in code size and lack of independence of the hashing functions. It has also been observed that increasing code length in such data driven methods may not necessarily improve performance monotonically, depending on the data characteristics. Redressing these issues are crucial for fast growing heterogeneous neuroscientific databases, where both scalability and retrieval performance are important.

Non-hashing based methods In addition to hashing based methods, distance preserving dimensionality reduction methods are also popular for retrieval. For compare and contrast the accuracy and time efficiency of hashing methods, we additionally include popular linear subspace learning methods like Principal Component Analysis (PCA) (Hotelling 1933) and Neighborhood Preserving Embedding (NPE) (He et al. 2005) for non-hashing based retrieval. PCA estimates the subspace projections that maximally preserve the data variance and aims at preserving the global Euclidean structure of the data. Differing from PCA, NPE preserves the local geometric structure of the data during subspace projection which is ideal for similarity preserving retrieval. Additionally, NPE is also reported to be less sensitive to outliers in comparison to PCA. He et al. (2005)

Retrieval in Neuroscience

Retrieval methods have also been investigated recently in neuroscience for large scale neuron retrieval. Costa et al. (2014) proposed a neuron search algorithm, where pairwise 3D structural alignment was employed to find similar neurons. In another approach, Polavaram et al. (2014) focused on the evaluation of morphological similarities and dissimilarities between groups of neurons deploying unsupervised clustering technique using expert-labelled meta data (like species, brain region, cell type, and archive). Recently, Wan et al. (2015) proposed a tool called BlastNeuron for comparing and clustering 3D neuron reconstructions. They retrieve similar neurons for a query neuron by retrieving candidate neurons based on their global morphology features, followed by local spatial alignment between the topology and geometry of the retrieved candidates to rank them. Due to pairwise comparisons of the neurons, the speed and scalability of these approaches are challenging especially in the current case of retrieval in large databases. This challenge can be efficiently overcome through hashing, as it effectively reduces pairwise comparisons to computation of efficient binary distances on compact binary codewords.

For better scalability of retrieval to a growing heterogeneous database, we design hashing functions established upon unsupervised random forests called Hashing Forests (HF). The HF are trained to parse the neuromorphological space in a hierarchical fashion and we demonstrate that HF can generate more sensitive code words than LSH, SH or STH by effectively utilizing its tree-structure and ensemble nature. Trees in HF are trained independently and they are more easily augmented for evolving databases than SH and STH, which require complete retraining. In comparison to random forest hashing method proposed by Yu and Yuan (2014), we introduce an inverse coding scheme which effectively mitigates database-wide pairwise comparisons, which is better suited for fast large-scale hashing. Figure 2 schematically illustrates the different methodological steps involved in using HF for neuron retrieval. In the following section, we provide underlying mathematical formulation

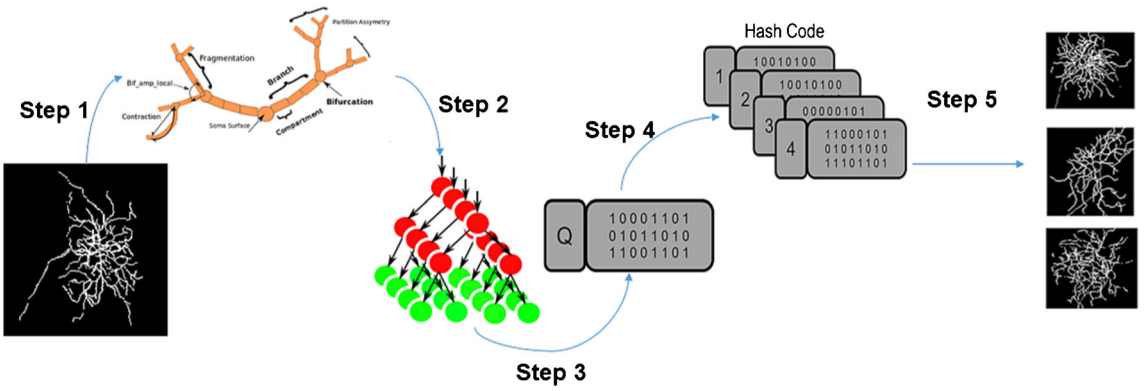

Fig. 2 *Schematic of the proposed method:* For a query neuron, we extract neuromorphological features (Step 1), which are then fed into the learnt Hashing Forests (Step 2). This results in a similarity-preserving binary query hash code (Step 3). Comparing this hash code to the Hash Table (codes of neurons in the search database) through the proposed inverse coding scheme (Step 4), we find and rank neurons that are morphologically similar to the query neuron (Step 5)

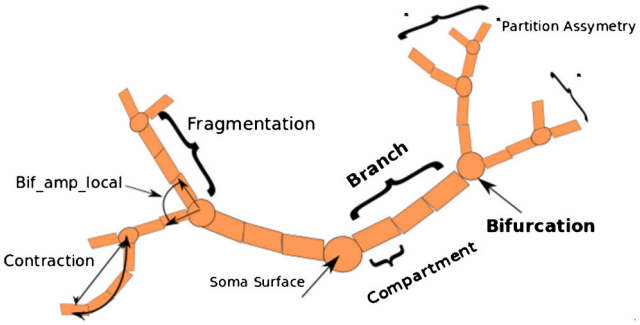

Fig. 3 Representation of a neuron with some of its morphological measurements

and discuss its software integration with the Neuron Miner tool.

Mathematical Formulation

Neuromorphological Space

The concept of the neuromorphological space has been introduced by Costa et al. (2010), where each neuron is represented by a set of morphological and topological measures. This feature space is used for multidimensional analysis of neuronal shape and showed that the cells of the same brain regions, types, or species tend to cluster together. This motivated us to leverage this space for evaluating inter-neuron similarity and propose a data-driven retrieval system for large neuron databases. We used the publicly-available Lmeasure software toolbox for extracting quantitative morphological measurements from digital 3D reconstructions of neurons (Scorcioni et al. 2008). These include features quantifying neurons at the level of the whole neuron, the branch level, and the bifurcation level. A sample neuron with some of its quantitative morphological measurements is illustrated in Fig. 3.

In total, we selected 37 features tabulated in Table 1. A detailed description of each feature metric has been presented in Overview of L-Measure (2015).

Hashing Forests

Hashing functions are used to encode records into hash codes, such that simple binary distance, defined on the hash codes, preserves similarity amongst the encoded records. Ideally, the hash function should generate compact and easy to compute representations, which can then be used for accurate search and fast retrieval (Weiss et al. 2012). In the context of neuroscientific databases, the desired morphological similarity preserving aspect of the hashing function implies that *morphologically similar neurons are encoded with similar binary codes*. This implies that for a particular *query* neuron (*say* n_q), the bucket of K morphologically similar neurons retrieved from the database $\mathscr{D}$ through hashing should be ideally as same as the K-nearest neighbors calculated using standardized Euclidean distance over the whole neuromorphological space. The rational behind this assumption is that the Euclidean distance defined on the subspaces between neurons correlates well with the desired morphological similarity. However, the Euclidean distance is not a desirable similarity measure because it requires significant memory expenditure and increases computational time for exhaustive pairwise comparison during hashing throughout in large neuron database.

In this work, we model hashing functions through *unsupervised* randomized forests ($\mathscr{H}$), which generates compact binary code blocks (*say,* $C_{\mathscr{H}}$) encoding the neuromorphological feature space. We hypothesize that computationally cheaper binary distance measures (*such as hamming distance etc.*), defined between the generated code blocks correlates well with the inter-neuron morphological similarity, and thus aiding in effective hashing. In the following sections, we discuss in detail the different stages involved

Table 1 Quantitative morphological measurements extracted from 3D digital reconstructions of neurons

Search specificity	Features
All features	arbor length, arbor height, arbor width, arbor depth, total volume, total number of tips, number of bifurcations, total surface, number of branches, diameter, soma surface, number of stems, contraction, fragmentation, Pk- classic + Branch level and Bifurcation level features.
Branch level	average and max helicity, average and max fractal dimension, average and max branch path length, max branch order,average terminal degree, path distance, euclidean distance.
Bifurcation level	average partition asymmetry, average and max local amplitude angle, average and max remote amplitude angle, average and max local tilt angle,average and max remote tilt angle, average and max local torque angle, average and max remote torque angle

in using hashing forests for encoding and retrieval within neuroscientific image databases.

Training phase

The hashing forest ($\mathscr{H}$) is an ensemble of binary decision trees which partitions the feature space hierarchically based on learnt binary oblique split functions. We introduce randomness through feature subspace bagging and bootstrapping to generate maximally decorrelated trees. The goal of a decision forest is to combine the predictions of several decorrelated trees built with different components in order to achieve high robustness in regards to noisy features.

Let:
X Neuron Morphological Features
Θ_s { `numTrees`, `treeDepth`, `numSplit`, `numFeat`, `numVar` } Configuration Parameters.
Result: Trained Hashing Forest $\mathscr{H}$
Initialization: Current Tree Index $k = 1$
repeat
  Get bagged dataset set X^k from X with `numFeat` features;
  Train Tree h^k
  $X_{n=1} \leftarrow X^k$ (Root Node);
  repeat
    $\phi_n \leftarrow$ `GenSplit` $(X_n, \texttt{numSplit}, \texttt{numVar})$
    $X_{2n} \leftarrow \{\mathbf{x} | \mathbf{x} \in X_n \;\&\; \phi_n(\mathbf{x}) \leq 0\}$
    $X_{2n+1} \leftarrow \{\mathbf{x} | \mathbf{x} \in X_n \;\&\; \phi_n(\mathbf{x}) > 0\}$
    $n \leftarrow n+1$
  until n reaches $2^{\texttt{treeDepth}} - 1$;
  $h^k \leftarrow$ Split nodes ϕ_n.
  $k \leftarrow k+1$;
until k reaches `numTrees`;
$\mathscr{H} \leftarrow \{h^1, \cdots, h, \cdots, h^{\texttt{numTrees}}\}$;

Algorithm 1: Training Hashing Forest (`TrainHF`)

The pseudo code for training of the hashing tree has been represented in Algorithm 1. Here, each hashing tree h^k is grown in an *unsupervised* fashion by recursively partitioning the feature set X_n, which reaches a particular node n into two subsets X_{2n} and X_{2n+1}. At each split node n, split functions ϕ_n are generated as shown in Eq. 1, which are randomly selected hyperplanes that split the feature space into two subsets. The hyperplane ϕ_n is parametrized by the parameter set θ_n, which comprise of the node-level feature-wise mean vector μ_n, the feature-wise standard deviation vector σ_n, and a vector of coefficients of individual features α_n along with an intercept scalar α_n^0. The values of μ_n and σ_n are estimated locally from training data X_n that reaches the node n.

The oblique split $\phi_n(\mathbf{x}, \theta_n)$ is defined as follows:

$$\phi(\mathbf{x}, \theta_n) = \left(\frac{\mathbf{x} - \mu_n}{\sigma_n}\right).\alpha_n + \alpha_n^0 \tag{1}$$

In this work, we use randomized node optimization, generating a family of candidate splits ($\mathscr{F}_n$), where each split (say $\theta_c \in \mathscr{F}_n$) is multivariate and assigned randomly generated coefficient values (say α_c computed from a parameter hypersphere of radius 1 centred at the origin (*i.e.* $\sqrt{\sum |\alpha_c|.^2} = 1$). The intercept α_c^0 is generated as a random value between the minimum and maximum of $\left(\frac{\mathbf{x}-\mu_n}{\sigma_n}\right).\alpha_c$. The coefficients are standardized by normalizing them to make their l_2 norm = 1 (*i.e.* $\sqrt{|\alpha_c|.^2 + |\alpha_c^0|^2} = 1$). Here, $\mathscr{F}_n$ is generated by randomly selecting `numVar` features from X_n at each split node and the candidate split function, which maximizes the node scoring function (given by Eq. (5)) is assigned to the split node in which:

$$\theta_n = \text{argmax}_{\theta_c \in \mathscr{F}_n} \mathscr{E}(X_n, \theta_c) \tag{2}$$

Using the above oblique split, the data set X_n is split into left and right subsets X_{2n} and X_{2n+1} by corresponding split functions as follows:

$$X_{2n} = \{\mathbf{x} | \mathbf{x} \in X_n \wedge \phi_n(\mathbf{x}, \theta_c) \leq 0\} \tag{3}$$

$$X_{2n+1} = \{\mathbf{x} | \mathbf{x} \in X_n \wedge \phi_n(\mathbf{x}, \theta_c) > 0\} \tag{4}$$

The choice of the optimal node split function is an interplay of two major factors namely, tree balance ($\mathscr{E}_B$) and cluster validity ($\mathscr{E}_C$), which are unified in defining $\mathscr{E}$ as shown below:

$$\mathscr{E}(X_n, \theta_c) = \underbrace{\mathscr{E}_C(X_n, \theta_c)}_{\text{Cluster Validity}} \times \underbrace{\mathscr{E}_B(X_n, \theta_c)}_{\text{Tree Balance}} \tag{5}$$

The pseudo code for generation of optimal splits using cluster validity has been represented in Algorithm 2.

Let:
X_n Training data reaching node n
{ `numSplit`, `numVar` } Split Parameters
Result: Split Function ϕ_n
Initialization: Current Candidate Index $c = 1$
$\mu_n \leftarrow$ Featurewise Mean (X_n);
$\sigma_n \leftarrow$ Featurewise Standard Deviation (X_n);
repeat
  $\alpha_c \leftarrow \text{rand}(\sqrt{\sum |\alpha_c|.^2} = 1)$;
  $\text{maxVal} \leftarrow \max\,(\alpha_c .* (\frac{\mathbf{x}-\mu_n}{\sigma_n}) | \mathbf{x} \in X_n)$;
  $\text{minVal} \leftarrow \min\,(\alpha_c .* (\frac{\mathbf{x}-\mu_n}{\sigma_n}) | \mathbf{x} \in X_n)$;
  $\alpha_c^0 \leftarrow \text{minVal} + (\text{rand}([0,1]) * (\text{maxVal} - \text{minVal}))$;
  Parameter Set: $\theta_c \leftarrow \{\mu_n, \sigma_n, \alpha_c, \alpha_c^0\}$;
  Cluster Validity: $\mathscr{E}_C^{\text{KL}}(\theta_c, X_n)$ using Eq. 6;
  Tree Balance: $\mathscr{E}_B(X_n, \theta_c)$ using Eq. 7;
  $\mathscr{E}(X_n, \theta_c) \leftarrow \mathscr{E}_C(X_n, \theta_c) \times \mathscr{E}_B(X_n, \theta_c)$ using Eq. 5;
  $c \leftarrow c+1$;
until c reaches `numSplit`;
$c^* \leftarrow \text{argmax}_c\, \mathscr{E}(X_n, \theta_c)$;
$\theta_n \leftarrow \theta_{c^*}$
$\phi(\mathbf{x}, \theta_n) = \left(\frac{\mathbf{x}-\mu_n}{\sigma_n}\right).\alpha_{c^*} + \alpha_{c^*}^0$ } Optimal Split;

Algorithm 2: Learn Oblique Split (`GenSplit`)

Cluster Validity Recursive splitting of the neuromorphological space can be modelled as a clustering operation, where morphologically similar neurons are grouped together as we traverse down the tree. It is therefore appropriate to evaluate the splitting functions on how well they partition the feature space such that data elements within particular child node are more morphologically similar than across the other children nodes. This translates into two measurement criterion for cluster validity: (1) *Compactness* within the a child node and (2) *Separation* across other children nodes. The compactness is often associated with dispersion within a dataset allocated to a particular child node and the separation is measured as a distance measure between the datasets across other children nodes.

Towards this end, we evaluate the cluster validity $\mathscr{E}_C(X_n, \theta_c)$ of split, generated by a candidate split function ϕ_c, using the Krzanowski and Lai Index (Albalate and Suendermann 2009; Kovács et al. 2005; Desgraupes 2013). In Krzanowski and Lai Index, the compactness and separation of the candidate split function ϕ_c is evaluated together by minimizing the empirical distortion induced due to splitting the dataset into the children nodes as follows:

$$\mathscr{E}_C^{\mathrm{KL}}(\theta_c, X_n) = 2^{2/\texttt{numFeat}} \left(\frac{1}{|X_n|} \sum_{\mathbf{x}_i \in X_n} \min_{j \in (X_{2n}, X_{2n+1})} (d_M(\mathbf{x}_i, \mathbf{c}_j)) \right) \quad (6)$$

where, $d_M(\mathbf{x}_i, \mathbf{c}_j))$ is the Mahalanobis distance. It is defined as $d_M(\mathbf{x}_i, \mathbf{c}_j) = \sqrt{(\mathbf{x}_i - \mathbf{c}_j)\Gamma^{-1}(\mathbf{x}_i - \mathbf{c}_j)^T}$, where Γ is the covariance matrix estimated from X_n. $\mathbf{c}_j$ refers to the node specific centroid evaluated from the dataset allocated to a particular child node. This measures were evaluated for each candidate split function at the split nodes and contributed as the $\mathscr{E}_C$ term in the node scoring function (Eq. 5).

Tree Balance Imbalance in an unsupervised tree is induced if the split divides the datasets into the children nodes in a skewed fashion, resulting in one child node encoding a larger data subset than the other node *i.e.* $|X_{2n}| > |X_{2n+1}|$ or $|X_{2n+1}| > |X_{2n}|$. We measure the degree of tree balance using a sigmoid function as follows:

$$\mathscr{E}_B(X_n, \theta_c) = \frac{2}{1 + e^{\gamma . \tau(X_n, \theta_c)}} \quad (7)$$

where

$$\tau(X_n, \theta_c) = \max \left\{ \left(\frac{|X_{2n}|}{|X_{2n+1}|} - 1 \right), \left(\frac{|X_{2n+1}|}{|X_{2n}|} - 1 \right) \right\} \quad (8)$$

In Eq. 7, γ is a hyper parameter that controls the importance of imposing tree balance while evaluating oblique splits. Increasing γ implies higher importance placed on tree-balance in Eq. 5 and this is illustrated in Fig. 4, where increasing γ penalizes more as tree imbalance increases.

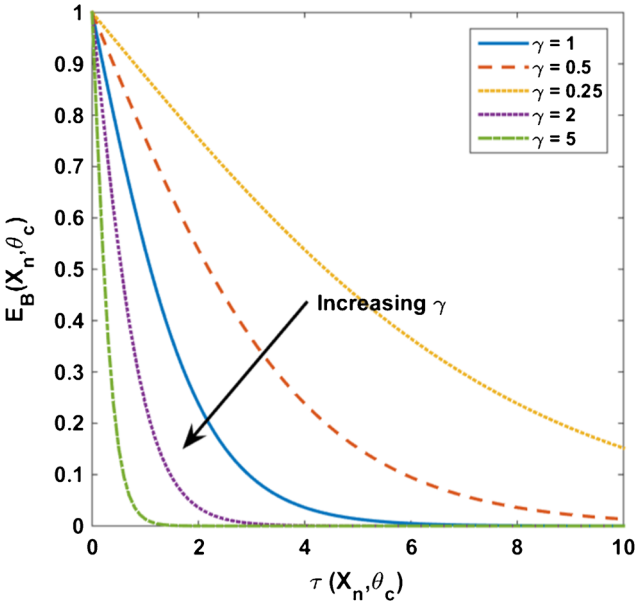

Fig. 4 Variation of $\mathscr{E}_B(X_n, \theta_c)$ with increasing tree imbalance $\tau(X_n, \theta_c)$ under different values of γ

The tree is grown through recursive splitting of the training dataset until the maximum defined tree-depth (treeDepth) is reached. We create an ensemble of such independently grown trees to create the hashing forest $\mathscr{H}$. Such a forest of `numTrees` binary trees with maximum depth of `treeDepth`, requires `numTrees` × `treeDepth` bits to encode each neuron. The time complexities to grow a tree and ensemble a forest are tabulated in Table 2 (S1-S2) (Louppe 2014).

Extension to non-Euclidean distances In the proposed formulation, the oblique split defined in Eq. 1 falls under the family of hyperplane hashing based locality sensitive hashing methods. The theoretical guarantees of such methods applies only to certain metrics such as $l_p \in (0, 2]$ (Wang et al. 2016). For extension of the proposed HF method to more complex metric spaces like weighted distance, power distance and other l_p distances, the splitting function has to be suitably defined to split data in that particular metric space. Typically, this can be done by considering a random sample reaching the split node as a pivot element and evaluating the metric distance of all other samples about this pivotal element. The split function can then be defined as a simple threshold over the obtained metric distances. The subsequent encoding and retrieval schemes proposed for the current HF formulation can be seamlessly extended for the non-Euclidean variants of HF.

Hash Table Generation:

Given a trained tree (h^k) of the hashing forest $\mathscr{H}$, each neuron n_i (*characterized by the neuromorphological feature*

Table 2 Time complexity analysis

S1	Building tree	$\mathcal{O}(\sqrt{N} * M * d) + \mathcal{O}(M * 2^{d-1})$	
S2	Building forests	$\mathcal{O}(T * (\sqrt{N} * M * d + M * 2^{d-1}))$	Training
S3	Generating hash table	$\mathcal{O}(T * d) + \mathcal{O}(M * S)$	
S4	Generating Single Query Code	$\mathcal{O}(T * d)$	
S5	Calculating Inter-neuron similarity with Forward Code	$\mathcal{O}(M * S)$	
S6	Calculating Inter-neuron similarity with Inverse Code	$\mathcal{O}(T * d)$	Retrieval
S7	Quick sort	$\mathcal{O}(M \log M)$	

Symbols: Code word Size $S = T(2^{d+1} - 2)$); Number of Trees T; Number of Features N; Tree Depth d; Retrieval Database Size M.

vector $\mathbf{x}_i$) in the database is passed through it till it reaches the leaf node. For a tree h^k of depth `treeDepth`, the split and the leaf nodes are assigned breadth-first order indices (say n_k), which are associated with binary bit b_n^k in the code word $C_k(\mathbf{x}_i)$. For a particular neuron, if node n_k is part of its path, then b_n^k is set to 1, otherwise, to 0. This leads to a $(2^{\texttt{treeDepth}+1} - 2)$ bit sparse code word $C_k(\mathbf{x}_i)$. It must however be noted that only `treeDepth` bits are required to generate the codeword as there are only $2^{\texttt{treeDepth}}$ possible traversal paths, each leading to a unique leaf node. We repeat the same process for every other tree in the forest to generate the sparse code block $C_{\mathscr{H}}(\mathbf{x}_i)$ of size $S = (\texttt{numTrees} \times (2^{\texttt{treeDepth}+1} - 2))$ for each neuron.

For faster retrieval, we pre-compute the code blocks for all M neurons in retrieval/training database $\mathscr{D}$ and generate a hash table of size $M \times S$. This is stored using (M*(`numTrees`*`treeDepth`)) bits along with traversal paths saved in a $(2^{\texttt{treeDepth}} * (2^{\texttt{treeDepth}+1} - 2))$ binary look-up table. However, as the database gets bigger, so will the time required for calculating the pairwise hamming distance between the codewords of all the data points in the dataset. The time complexity of generating the hash table for all the neurons in the database is shown in Table 2 (S3). To address this problem, we further propose to generate the inverse codewords to improve the retrieval speed performance.

Inverse Coding

Each bit b_n^k in $C_{\mathscr{H}}$ encodes a unique neuromorphological sub-space, which is constrained by the split functions of tree h^k leading to node n_k. In order to avoid pair-wise comparisons between the neurons during retrieval in large databases, we formulate an inverse coding scheme. We transpose the hash table to generate the inverted hash table $\mathscr{I}$, which is a sparse $(S \times M)$ dimensional matrix. This implies that for feature vector $\mathbf{x}_i$, if bit b_n^k in $C_{h^k}(\mathbf{x}_i)$ is 1, then $\mathscr{I}(n_k, i) = 1$, and it belongs to the feature subspace encoded by b_n^k. Given a new *query* neuron, instead of calculating the pairwise-similarity between all neurons in $\mathscr{D}$, we extract the corresponding hash code from the hash table, which is a representation of similarity vector between the new point and all the other data points. Through the generation of the inverse hash table $\mathscr{I}$, we have effectively encoded the neuromorphological subspaces along with associated neurons.

Testing Phase

The path in which a neuron traverses through the trained trees is used to define inter-neuron similarity. For a given *query* neuron n_q (with feature vector $\mathbf{x}_q$), the corresponding code block $C_{\mathscr{H}}(\mathbf{x}_q)$ is generated in a similar fashion to the Hash Table Generation phase. In the direct retrieval formulation, pairwise comparisons (*through hamming distance*) between $C_{\mathscr{H}}(\mathbf{x}_q)$ and code blocks of neurons (say $C_{\mathscr{H}}(\mathbf{x}_i)$ for neuron n_i) in the retrieval database $\mathscr{D}$ are made to evaluate inter-neuron similarity $\mathscr{S}(n_q, n_i)$ *i.e.*

$$\mathscr{S}(n_q, n_i) = \frac{1}{S} \sum_{\forall bits} \left(C_{\mathscr{H}}(\mathbf{x}_q) == C_{\mathscr{H}}(\mathbf{x}_i) \right) \tag{9}$$

If n_q generates the same code block as a neuron in $\mathscr{D}$ (*i.e. both belong to the same neuromorphological subspace*), we assign perfect similarity to them ($\mathscr{S} = 1$). However, the pairwise comparison for large scale databases is computationally expensive as seen from its time complexity in Table 2 (S5). To mitigate this, in the inverse coding, we formulate the similarity function as $S_{\mathscr{I}} = \texttt{numTrees} * (\texttt{treeDepth} - 1)$ dimensional similarity accumulator cell $\mathscr{A}_{n_q}$. Given the code block $C_{\mathscr{H}}(\mathbf{x}_q)$ for the query neuron n_q, $\mathscr{A}_{n_q}$ is calculated as:

$$\mathscr{A}_{n_q}(i) = \frac{1}{S_{\mathscr{I}}} \sum_{\forall n_k} \mathscr{I}(n_k, i) \text{ if bit } b_n^k \text{ in } C_{\mathscr{H}}(\mathbf{x}_q) = 1 \tag{10}$$

The inter-neuron similarity $\mathscr{S}(n_q, n_i)$ is related to $\mathscr{A}_{n_q}$ as $\mathscr{S}(n_q, n_i) = \mathscr{A}_{n_q}(i)$. Such an inverse formulation is computationally more efficient for large databases (as seen

from the order complexity in Table 2 (S6)), than the forward scheme (Table 2 (S5)). The inverse coding evaluates inter-neuron similarity by accumulating the membership of the database neurons from the columns of the inverse hash table corresponding to the tree nodes reach by that of the query neuron in HF. This effectively mitigates the need for pair-wise comparisons, leading to a time complexity that is independent of the database size M.

In the task of retrieving an ordered set of K most morphologically similar neurons from $\mathscr{D}$, we sort the neurons of the database in ascending order of inter-neuron similarity using quick-sort (time complexity of $\mathscr{O}(M\ \log M)$). The top $(2K)$ nearest neighbor neurons are further re-ranked according to their normalized Euclidean distance from the *query* neuron for better morphological consistency (with an additional time complexity of $\mathscr{O}(KN + K\ \log K)$). For validation purposes, this re-ranking using normalized Euclidean distance is performed on all comparative methods and baselines considered in Section "Experiments and Results".

Experiments and Results

Database

We used 31266 3D reconstructions of neurons extracted from 147 different archives, which curated from multiple laboratories and are publicly available on http://neuromorpho.org (Ascoli et al. 2007). All archives listed as '*In the Repository*' in the list of archives in the Neuromorpho.org repository have been included in this study (Literature Search Main Results 2015). We employed the Lmeasure toolbox to extract 3D neuromorphological features, which characterize different aspects of neuron structure and topology (Scorcioni et al. 2008; Costa et al. 2010).

Evaluation Metrics

Successful morphology-preserving hashing in neuroscientific databases depends on the efficacy of the code word to compactly parse and represent the neuromorphological space as well as efficiently compute inter-neuron similarity using the generated hash codes. As part of validations, we use the following evaluation metrics:

Neighborhood Approximation We introduced the *Neighborhood Approximation* (NA) graph in Mesbah et al. (2015) to model how close the estimated neighborhood, computed from code words, (*from the hashing method*), approximates the true neighborhood around a neuron in the neuromorphological space. For a particular hashing method, the NA for the j^{th} neighbor is defined as the average of the normalized Euclidean distances between the neurons and retrieved j^{th} neighbor for all neurons in $\mathscr{D}$. Let, for neuron n_i (with feature vector $\mathbf{x}_i^0$), the j^{th} neighbor have a feature vector $\mathbf{x}_i^j$, then

$$\mathrm{NA}(j) = \frac{1}{M_{\text{test}}}\left(\sum_{i=1}^{M_{\text{test}}} \varepsilon(\mathbf{x}_i^0, \mathbf{x}_i^j)\right) \tag{11}$$

where

$$\varepsilon(\mathbf{x}_i^0, \mathbf{x}_i^j) = \left(\sqrt{\frac{1}{N}\sum_{a=1}^{N}\left(\frac{x_{ia}^0 - x_{ia}^j}{s_a}\right)^2}\right) \tag{12}$$

$\varepsilon(\mathbf{x}_i^0, \mathbf{x}_i^j)$ is the standardized Euclidean distance between $\mathbf{x}_i^0$ and its j^{th} neighbor $\mathbf{x}_i^j$ with a^{th} feature standard deviation s_a estimated over the whole database (which is chosen for invariance to scales of different features). NA-graph is averaged over M_{test} test neurons from the testing dataset.

Retrieval Performance We evaluate the retrieval performance by computing two metrics: Kendall's rank correlation coefficient κ and G_{mean}. The G_{mean} is often used in information retrieval algorithms to better understand the trade-off between precision and recall. Let $\mathscr{N}_\varepsilon(n_i)$ represents the set of neurons 'relevant' to the query neuron, which is the top K nearest neighbors defined upon the normalized Euclidean distance in the neuromorphological space and $\mathscr{N}_\mathscr{H}(n_i)$ represents the retrieved neurons through hashing as a set of k morphologically similar neurons. The G_{mean} is calculated as an average over the test database and is calculated as:

$$G_{mean} = \sqrt{Precision \times Recall} \tag{13}$$

$$= \frac{1}{M_{\text{test}}}\sum_{i=1}^{M}\sqrt{\frac{\left|\mathscr{N}_\varepsilon(n_i)\bigcap\mathscr{N}_\mathscr{H}(n_i)\right|^2}{|\mathscr{N}_\mathscr{H}(n_i)| \times |\mathscr{N}_\varepsilon(n_i)|}} \tag{14}$$

Kendall's rank correlation coefficient κ is used to measure the association between two ranked lists (Kendall 1948). We use this metric to evaluate the efficacy of ranking of relevant neurons. Given a pair of ranking lists

$$\left(r_1^R, r_1^T\right), \left(r_2^R, r_2^T\right), \cdots, \left(r_n^R, r_n^T\right)$$

(here, retrieved list (R) *vs.* true neighborhood list (T)). A pair of observations $\left(r_i^R, r_i^T\right)$ and $\left(r_j^R, r_j^T\right)$ are said to be concordant if the ranks on both lists agree *i.e.* $r_i^R > r_i^T$ and$r_j^R > r_j^T$ or $r_i^R < r_i^T$ and$r_j^R < r_j^T$. The pairs are deemed discordant if $r_i^R > r_i^T$ and$r_j^R < r_j^T$ or $r_i^R < r_i^T$ and$r_j^R > r_j^T$. If $r_i^R = r_i^T$ and$r_j^R = r_j^T$, they are

neither concordant nor discordant. The κ for the retrieval performance on test dataset is evaluated as follows:

$$\kappa = \frac{1}{M_{\text{test}}} \sum_{i=1}^{M_{\text{test}}} \frac{n_c^i - n_d^i}{n_c^i + n_d^i} \tag{15}$$

where n_c^i and n_d^i are the number of concordant and discordant pairs extracted from the respective R and T lists for each neuron n_i . In case of total agreement and disagreement between the two paired lists, the coefficient value is $\kappa = 1$ and under $\kappa = -1$, respectively.

Retrieval Time Hashing aims at minimizing the time for retrieval by reducing expensive pairwise distance computations to cheaper binary operations defined over the hash codes (like `xor` for Hamming distance computation). As discussed in Section "Mathematical Formulation", we consider two different strategies for hash code comparison: (1) Forward Coding and (2): Inverse Coding. For a particular hashing method, the training time includes time required to train the hashing functions and generate the hash table for database. The testing time includes the time required for generating the hash codes for the query items, time for comparison (forward / inverse coding), sorting and ranking the approximate nearest neighbors. We incur an additional time overhead during testing, if the fetched neighbors are re-ranked according to their normalized Euclidean distance from the query item.

Comparative Methods

The main contribution of the hashing forests formulation presented in this paper over our previously proposed formulation (Mesbah et al. 2015) is the introduction of oblique split functions and improvised node-scoring with cluster validity measures. Further, we formulate tree-traversal path based coding scheme as opposed to leaf-based scheme proposed in Yu and Yuan (2014) for more efficient hierarchical parsing of the neuromorphological space. These propositions lead us to four baselines to test the hypothesis that introducing these contributions improve hashing performance. The baselines are tabulated in Table 3. Each baseline differs in terms of the choice of the encoding scheme (leaf node/tree path encoding), the inclusion/exclusion of cluster validity and the type of the splitting function.

Comparisons to these baselines would support our hypothesis that oblique splits with cluster validity leads to better parsing of the neuromorphological space, resulting in higher code efficiency. In addition, we validate the performance of our proposed algorithm (HF) by comparing it against popular large scale hashing methods discussed in Section "State-of-the Art", including Locality Sensitive Hashing (LSH), Spectral Hashing (SH), and Self taught hashing (STH). Additionally, as a baseline for comparison against hashing based approaches we include dimensionality reduction based retrieval methods, including Principal Component Analysis (PCA) and Neighborhood Preserving Embedding (NPE). In case of PCA and NPE, we used single-precision floating point representation for the embedding and retrieval was done by pairwise computation of Euclidean distance in the embedding space between the query item's embedding and that of the target database.

Table 3 Hashing forest baselines

Baselines	Encoding LN - Leaf node TP - Tree-path	Cluster Validity	Split type
Baseline 1 (BL1)	LN	×	Axis-aligned
Baseline 2 (BL2)	TP	×	Axis-aligned
Baseline 3 (BL3)	TP	×	Oblique
Baseline 4 (BL4)	LN	✓	Oblique
Proposed	TP	✓	Oblique

Hyperparameter Selection for Hashing Forests

The main hyper parameters to be optimized for hashing forests include: tree balance parameter (γ), number of trees (`numTrees`), and their depth (`treeDepth`). For hashing forests, the hash code word size is given by `numTrees`×`treeDepth`. Fixing the code-size, we first optimize γ to be used in further analysis. For this, we fix the code-size at 128 bits and optimize γ for three configurations of HF: Shallow Trees (HF-S with `treeDepth` = 2), Moderately Deep Trees (HF-M with `treeDepth` = 4), and Very Deep Trees (HF-D with `treeDepth` = 8). The `numTrees` are chosen accordingly as 64, 32 and 16 respectively. The hyperparameter γ was varied as [2.0, 1.0, 0.5, 0.25, 0.1and0.05] with decreasing importance towards tree-balance. The G_{mean} for each of these configurations is tabulated in Table 4.

From Table 4, comparing HF-S, HF-M and HF-D, we infer that for sufficient depth, the performance of HF is invariant to choice of `numTrees` and `treeDepth`. We observe consistent optima at $\gamma = 1.0$ for all three tested configurations. Therefore, for the rest of validations, we fix the tree balance parameter γ at 1.0 and `treeDepth` at 4, corresponding to moderately deep trees. This observation is extendable to other code-sizes as trees are grown independently in a decorrelated fashion.

Table 4 Hyperparameter selection for HF: G_{mean} *vs.* γ

γ	2.0	1.0	0.5	0.25	0.1	0.05
HF-S	51.09	54.25	51.34	52.87	53.80	52.96
HF-M	65.27	68.04	66.31	65.63	65.71	65.57
HF-D	67.18	67.65	67.15	63.51	64.20	64.10

Neighborhood Approximation

In an unsupervised hashing setting, the distance function defined on the original neuromorphological feature space (say, normalized Euclidean distance (NA_{EUC})) is deemed to have the best neighborhood approximation. Thus, the hashing method that diverges the least from the NA_{EUC} graph preserves the true neighborhood to the best possible extent. The NA graph is evaluated over the target database and the results for all comparative methods are reported in Fig. 5. For fair validation, we keep the size of the code-block fixed at 256 bits for this experiment. This evaluation is performed for all the baselines and comparative methods. The `treeDepth` for the baselines BL1-4 and HF was fixed at 4, thus leading to `numTrees` of 64.

Hashing retrieval performance vs. Code block size

We measure the G_{mean} and Kendall's κ statistic for all comparative methods as well as baselines by varying the code block size from 4 bytes to 64 bytes in geometric order of 2. We compare the performance for retrieval (both search and ranking) of the top 10 neighbored neurons using these methods for a heterogeneous test set of 800 randomly selected neurons (not included while training) and the results are tabulated in Tables 5 and 6. This validation is performed to evaluate the improvement in similarity preserving aspect of the hash code with increasing code-size. It also serves to validate our hypothesis that introducing oblique splits with cluster validity and using whole tree-traversal path for encoding leads to more efficient hash codes over the baselines and comparative methods. Figure 6 demonstrates the performance for 4 distinct neurons of differing morphologies with the closest neighbors retrieved using the proposed HF formulation and the ground truth (minimal normalized Euclidean distance). The HF was trained with $\gamma = 1.0$, `numTrees` = 64 and `treeDepth` = 4 and through visual evaluation, we observe close morphological similarity amongst the ground-truth neurons and its retrieved neighbors.

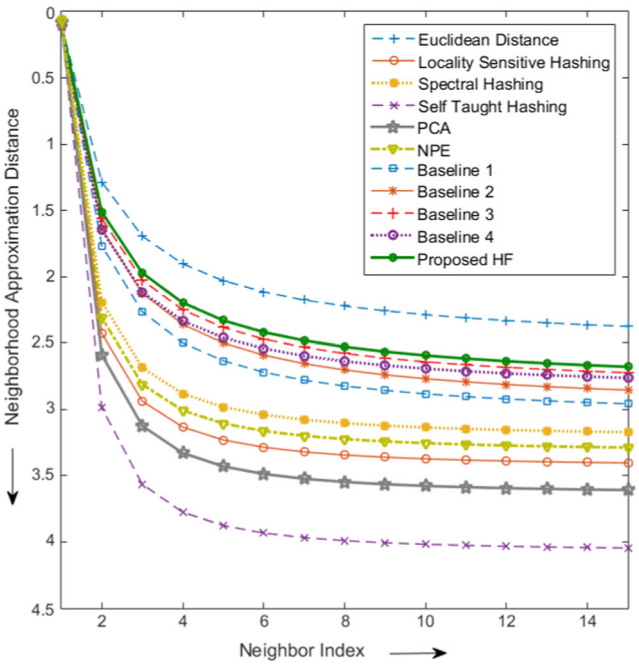

Fig. 5 Neighborhood Approximation (NA) graph for the comparative methods and configurations of hashing forests for fixed codeword size (32 bytes)

The time for retrieval is an important evaluation metric for retrieval using hashing. To compare and contrast the retrieval time against exhaustive pairwise distance computation, we report the time for training and testing for the comparative methods and baselines in Table 7. The training time includes the time to train the hash functions and extract the hashing table on the training data of 30466 neurons. The testing time includes the time for generation of the test hash codes and comparing it against the *a priori* extracted hash table using forward / inverse coding schemes for 800 test neurons. These algorithms were implemented on a general purpose 64-bit CPU with 16GB RAM memory and 2.7GHz Intel(R) Core(TM) i7-4600U processor. In case of retrieval with a mobile application, the actual retrieval time additionally depends on the data transfer speed and the hardware configuration of the mobile device.

Incremental training with database evolution

As the database evolves with addition of new data, the current form of the hashing function can be directly employed for populating the hash table with the new incoming data (method M1). Alternatively, the hashing functions can be retrained on the extended dataset with the additional new data (method M2). If the current code-size cannot

Table 5 Retrieval performance (G_{mean}) vs. Code block size

Code Size	Comparative Methods					Baselines				Proposed
(in bytes)	LSH	SH	STH	PCA‡	NPE‡	BL1	BL2	BL3	BL4	
4	24.08	26.45	24.80	42.11	48.20	29.23	30.91	29.44	30.59	34.27
8	28.86	35.20	38.20	**57.15**	52.41	29.69	42.92	44.19	45.92	49.41
16	41.15	53.00	43.40	61.60	62.65	43.11	58.40	60.61	63.16	69.51
32	46.98	67.40	47.60	64.91	67.75	59.37	72.70	76.05	81.00	83.13
64	57.51	81.60	47.20	67.72	70.80	74.53	84.27	87.31	83.48	**92.72**

* - These are non-hashing comparative methods (dimensionality reduction) using floating-point representation (1 float = 4 bytes).

Note: The best performance for a fixed code size is shown in **boldface**. and the best result amongst all the comparative methods is frameboxed.

sufficiently handle the added heterogeneity as the database evolves, the hashing functions can be augmented with further hash functions trained independently only on the additional new dataset (method M3) and appending the newly generated hash codes to the existing hash table. In case of HF, such a code augmentation translates to training additional independent hashing trees on the additional dataset and concatenating these to the tree ensemble of the existing HF. Alternatively, the hash functions can be retrained on extended dataset for a larger code-size (method M4). Comparing the alternative methods to handle database evolution, M2 and M4 are computationally expensive in comparison to M1 and M3. It must be noted that in scenarios where database evolution involves addition of new morphological features, the proposed and the comparative hashing methods can potentially be extended to multi-view formulations such as proposed in Liu et al. (2015).

To evaluate the performance of different hashing functions as the database evolves, we create a test scenario wherein the initial hash functions are trained on 19886 neurons curated from 86 archives. The database evolution is modeled by addition of 5048 new neurons from 16 additional data archives to the initial dataset. The new incoming dataset is divided into non-overlapping training and testing datasets of 4548 and 500 neurons respectively. M1 is trained for a code-size of 32 bytes, M2 is retrained for the same code size as M1, M3 augments M1 with an additional 8 bytes making the code-size 40 and M4 is retrained for a code size of 40 bytes. The retrieval performance evaluated using G_{mean} score for top-10 neighbor retrieval for each of the proposed and comparative methods is tabulated in Table 8. To analyze the time overheads incurred during each of the four methods M1-M4, we also report the training time and the testing time (using inverse coding) in Table 8.

Discussion

In the previous section, we designed experiments to validate our hashing forest performance and perform comparative analysis with reference to other large-scale generalized hashing methods and baselines. We further discuss in detail the observations and inferences we draw from them in the following section.

Neighborhood Approximation

The NA graph evaluates how well a code word generated by particular hashing method is able to approximate the neighborhood around a query neuron with respect to

Table 6 Retrieval performance (Kendall's rank correlation coefficient κ) vs. Code block size

Code Size	Comparative Methods					Baselines				Proposed
(in bytes)	LSH	SHx	STH	PCA‡	NPE‡	BL1	BL2	BL3	BL4	
4	0.2266	0.2450	0.2280	0.3986	0.4713	0.2879	0.2949	0.2998	0.2913	0.3250
8	0.2875	0.3569	0.3307	0.5628	0.5135	0.2970	0.4015	0.4113	0.4270	0.4554
16	0.3970	0.4729	0.3742	0.6010	0.6218	0.3869	0.5355	0.5354	0.5787	0.6505
32	0.4344	0.7329	0.4049	0.6351	0.6703	0.5492	0.6867	0.7338	0.7847	0.8149
64	0.5382	0.7867	0.4200	0.6645	0.7081	0.7035	0.8230	0.8476	0.8108	**0.9274**

∗ - These are non-hashing comparative methods (dimensionality reduction) using floating-point representation (1 float = 4 bytes).

Note: The best performance for a fixed code size is shown in **boldface** and the best result amongst all the comparative methods is frameboxed.

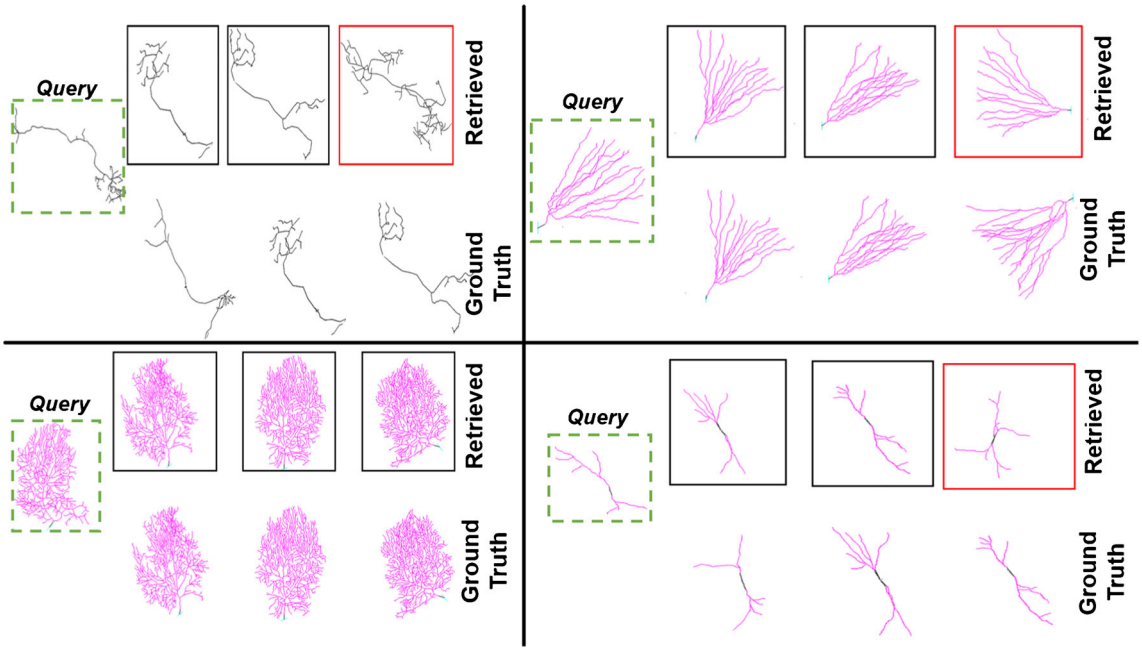

Fig. 6 Visual evaluation of neuron retrieval: For each query neuron on the left (boxed in green), the top-three neighbors retrieved with the proposed HF algorithm are shown along with ground truth neurons (using normalized Euclidean distance) are shown. The incorrect results are marked by red boxes

neighborhood defined using normalized Euclidean distance. From Fig. 5, we observe a divergent trend (with reference to the ground-truth NA_{EUC} graph) in the NA graphs of all methods as the neighbor index increases. For a fixed code size, the HF and other forest based baselines BL1-4 approximate neighborhood better than the comparative LSH, SH, and STH methods. This supports the hypothesis that effective utilization of the tree-structure along with ensemble nature of these methods improves data-driven parsing of the neuromorphological space. Comparing neighborhood approximation of dimensionality reduction driven retrieval methods, we observe that NPE exhibits better NA

Table 7 Hashing retrieval performance (Training and testing time) vs. Code block size

	Exhaustive	Code size	Comparative methods					Baselines				Proposed
		(in bytes)	LSH	SH	STH	PCA‡	NPE‡	BL1	BL2	BL3	BL4	
Training (in s)	-	4	0.046	0.34	160.9			1.83	1.77	1.78	5.85	5.65
		8	0.081	0.49	320.8			3.23	3.41	3.43	11.52	12.06
		16	0.166	1.23	684.4	0.124	289.3	6.48	6.41	6.27	25.73	22.39
		32	0.270	2.07	1385.6			15.44	13.7	13.9	45.12	48.56
		64	0.606	3.612	1953.6			29.09	26.7	27.8	93.15	97.31
Testing(in s)	91.654	4	2.642	2.652	2.718	5.29	5.18	3.282	5.648	5.654	3.282	5.646
Forward coding		8	4.213	4.228	4.357	8.46	8.77	5.036	8.705	8.705	5.035	8.708
		16	7.104	7.139	7.329	9.51	9.49	7.838	15.868	15.869	7.836	15.868
		32	13.226	13.808	13.808	14.86	14.95	14.255	31.406	31.401	14.256	31.407
		64	24.742	24.909	25.676	29.81	31.99	33.119	67.741	67.739	33.019	67.718
Testing(in s)	-	4	0.328	0.338	0.414			0.416	0.709	0.715	0.416	0.707
Inverse coding		8	0.543	0.558	0.687			0.660	1.129	1.129	0.659	1.132
		16	0.951	0.986	1.176	-	-	1.063	2.131	2.132	1.061	2.131
		32	1.581	1.653	2.163			1.728	3.771	3.766	1.729	3.772
		64	3.142	3.309	4.076			4.222	8.619	8.617	4.222	8.616

∗ - These are non-hashing comparative methods (dimensionality reduction) using floating-point representation (1 float = 4 bytes).

Table 8 Retrieval performance (G_{mean}) and time-analysis of hashing methods with database evolution

	Comparative Method	M1 Original 32 bytes[#]	M2 Retrained 32 bytes	M3 Augmented 40 bytes[$]	M4 Retrained 40 bytes
Performance	LSH	40.23	42.93	46.13	47.10
	SH	45.32	46.18	47.22	51.40
	STH	41.44	42.80	43.15	44.54
	BL2[‡]	47.87	49.20	49.77	53.43
	Proposed	68.20	70.03	71.47	73.37
Training time (in s)	LSH	0.098 (0.281)	0.265	0.119 (0.078 + 0.041)	0.292
	SH	0.030 (1.263)	1.697	0.192 (0.074 + 0.118)	2.38
	STH	3.085 (900.6)	1177.8	67.36 (51.32 + 16.04)	1662.7
	BL2[‡]	0.379 (7.94)	10.69	1.944 (0.58 + 1.364)	17.8
	Proposed	0.318 (30.107)	41.27	3.603 (2.171 + 1.432)	67.98
Testing time inverse coding (in s)	LSH	1.012	0.961	1.176	1.305
	SH	1.157	1.215	1.413	1.243
	STH	1.319	1.467	2.055	2.230
	BL2[‡]	2.681	2.582	2.972	3.156
	Proposed	2.474	2.756	3.282	3.151

$*$ - Prior art method (Mesbah et al. 2015)

[#] - $\tau_1^{M1}(\tau_2^{M1})$ - τ_1^{M1} is the time required to infer hash-codes the new incoming dataset using existing hash functions (τ_2^{M1} is the time required for training the existing hash functions, however it is not deemed as a part of the training time for M1).

[$] - Total training time for M3 is $\tau^{M3} = (\tau_1^{M3} + \tau_2^{M3})$ where τ_1^{M3} is the time to train the augmented hash codes on the incoming dataset and τ_2^{M3} is the time required to repopulate the existing dataset through the augmented hash functions.

over PCA as it effectively preserves local neighborhoods during embedding. In comparison to hashing methods, NPE demonstrates performance superior to STH and LSH and is comparable to SH.

Comparing BL2 with BL1, and proposed HF with BL4, we infer that using tree-traversal path encoding over leaf node encoding leads to better neighborhood approximation. This can be associated to the fact that complete decision path allows for a partial neighborhood contribution in calculation of inter-neuron similarity.This effect is illustrated in Fig. 7, where we consider two distinct neurons n_i and n_j which share nodes R, S_1 and S_4 during tree traversal. However, they reach different leaf nodes L_3 and L_4 respectively. The similarity metric between n_i and n_j defined with tree-traversal path-encoding is $\mathscr{S}(n_i, n_j) = 2/3$, as they shared $2/3^{\text{rds}}$ of the traversal path. In contrast, with leaf node encoding, $\mathscr{S}(n_i, n_j) = 0$, as they reach distinct leaf nodes. This partial neighborhood helps improving the neighborhood approximation of the hash codes. Finally, comparing the baselines BL2 and BL3 to proposed HF, we observe that the neighborhood approximation is improved when oblique splits (in HF over BL2) and cluster validity (in HF over BL3) are employed.

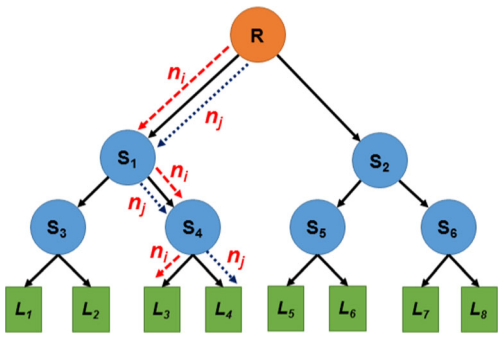

Fig. 7 Illustration of partial neighborhood effect due to tree-traversal encoding

Hashing retrieval performance vs. Code size

Comparative Methods We quantitatively evaluated the performance of the proposed method for different lengths of hash codes. It is clearly seen from Tables 5 and 6 that the proposed HF performance improves as code length increases, and achieves better results consistently in comparison to other hashing based methods in searching and ranking relevant neurons. It must be noted that we chose larger code sizes over conventional code sizes ($>$ 16 bytes), as it was observed that precision-recall performances for HF and comparative methods for smaller code sizes were not

sufficient enough for the application at hand. In comparison, the dimensionality reduction based retrieval methods (PCA, NPE) exhibit superior retrieval performance for smaller code sizes (4-8 bytes) over hashing based methods. However, the retrieval performance for PCA and NPE does not improve significantly with increasing dimensionality of embedding with the inclusion of projections corresponding to lower Eigen values.

By looking at the results obtained from comparative methods, we observe that the SH performs consistently better than the LSH and the STH. Though the LSH's performance steadily increases with increasing code-size, the improvement is considerably slower that implies that the LSH needs much higher code sizes for achieving comparable performance to the proposed HF or the SH, which will significantly increase the computational cost, resulting in delayed retrieval (corroborates observations reported by Yu and Yuan (2014)). In case of the STH, as code-size increases, the eigenvectors corresponding to higher eigenvalues are utilized in defining the hashing function. This is not desirable because eigenvalues are often very noisy. It must be noted that the reported results for LSH, STH, SH, and BL2 are different from those reported in our previous work (Mesbah et al. 2015). Because, the database has increased in size and heterogeneity from 18106 neurons to 31266 neurons and subsequently the configuration parameters have been changed through cross-validation optimization. Interestingly, the trend of overall performance reported in Mesbah et al. (2015) has remained unaltered.

Baselines As established in the discussion of NA Graph, tree-traversal path based encoding with its partial neighborhood effect demonstrates considerable improvement in retrieval performance. For code size of 64 bytes, we observe from Table 5, an overall increase of 9.74 % between BL2 and BL1 (84.27 % from 74.53 %) and 9.24 % between the proposed HF and BL3 (92.72 % from 83.48 %). This trend is consistent in the ranking performance as the Kendall's κ statistic improvises by 0.1195 between BL2 and BL1 (0.8230 from 0.7035) and by 0.1166 between the proposed HF and its leaf-encoding baseline BL4. These observations further corroborate the hypothesis that partial neighborhood effect is desirable for effective retrieval of true neighbors. We also report considerable improvement of 8.45 % from 84.27 % to 92.72 % for the 64 byte code size, over our previous HF formulation (Mesbah et al. 2015) (BL2). This trend is consistently observed across all the other smaller code sizes too. These observations demonstrate the superiority of the proposed HF formulation over the baselines and validates our hypothesis that oblique splits with cluster validity improves code efficacy. The improved performance of BL3 in contrast to BL2 is due to the use of oblique splits. This can be attributed to the following aspects: (a) Oblique splits can separate distributions that lie between the coordinate axes with a single multivariate split, which might have required deep nested axis-aligned splits otherwise; (b) The learnt hashing trees are less biased to the geometrical constraints of the base learner if oblique splits are used (also observed by Menze et al. (2011)). Further, inclusion of cluster validity during training, ensured that the neighborhoods, resulting from clustering of similar neurons in the neuromorphological space (as observed by Polavaram et al. (2014)), are preserved during the generation of hashing forest splits. This has resulted in improved retrieval performance of the proposed HF in comparison to the nearest baseline BL3 (oblique splits without cluster validity).

Time Analysis We profiled the training and the testing time for retrieval of the comparative methods for varying code-lengths for 10 trials with setting identical to Table 6 and tabulated the average observed time for training and testing in Table 7. In the comparison of the training times, we observe that all the methods except STH and NPE exhibit training time of under 100 seconds. The high training time of STH and NPE is attributed to the computationally expensive eigenvalue decomposition step (order complexity of $\mathcal{O}(M^3)$). Additionally in STH, the hash functions are independently trained binary support vector machine classifiers that are computationally expensive to train for large datasets (order complexity of $\mathcal{O}(SMN)$). During retrieval, we observe that employing inverse coding for hashing methods reduces the time for comparison and ranking significantly in comparison to forward coding and is significantly lower than exhaustive pairwise distance computation. Comparing to the baselines, we observe that BL1 and BL4 exhibit lower retrieval time in comparison to BL2, BL3 and the proposed method due to the difference in the encoding schemes employed for comparison (leaf node for BL1, BL4 and tree-path encoding for BL2, BL3 and the proposed method). Compared to other hashing methods, the proposed method with inverse coding has a higher retrieval time for the same code size, but is significantly lower than pairwise comparison used in dimensionality reduction methods.

Incremental training with database evolution

With addition of new unseen data to the database, we evaluate variants of hashing methods (retraining *v.s.* augmentation) that have been proposed in Section "Experiments and Results" and report their retrieval performance (G_{mean}) in Table 8. From an overall perspective, we conclude that augmenting hashing functions with additional bits (M3) performs comparably to retraining (M4) and is superior to retrieving with the the original hash function (M1). Further, the proposed HF demonstrates significantly higher retrieval

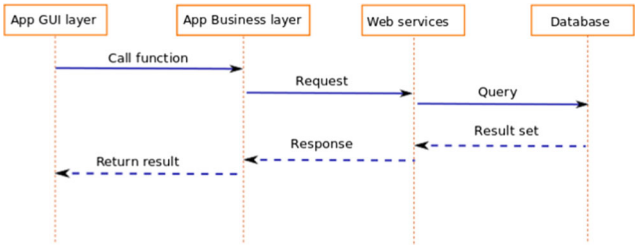

Fig. 8 Sequence diagram of the global workflow of Neuron-Miner

performance over the comparative hashing methods (LSH, STH, SH and BL2) which is highly desirable as the database continually evolves.

From Table 8 we observe that time overhead for training/augmenting the hash codes for M1 and M3 are relatively lower in comparison to retraining based methods (M2 and M4). These observations are concurrent with the expected trends as M1 involves no additional learning of the hash functions and M3 learns the augmentation hash function on a relatively smaller incoming dataset for a smaller code size. In comparison, M2 and M4 involves retraining the entire hash functions on the extended dataset (existing dataset + incoming dataset). In terms of the testing time, we observe that M1 and M3 are comparable to M2 and M4 respectively, as the time complexity for inverse coding is linear in terms of code-size and independent of the search database size.

Software Implementation

Neuron-Miner is envisaged to be used as a data-mining tool for neuroscientists to organize and visualize their data. We also aim at augmenting them with tools for effective comparative analysis spanning large heterogeneous, multi-center databases. The hashing forest algorithm is available on the server of the Max Plank Digital Library, München, Germany and is usable through the Neuron-Miner application which is available for downloading and installing on mobile devices supporting Android through https://servicehub.mpdl.mpg.de/Neuron-miner.apk. In the following section, we elaborate on the software engineering aspects of the implementation of Neuron-Miner tool, both from the Server and the Client Side.

Server side

In Fig. 8, we demonstrate the overall work flow of Neuron-Miner, which is composed of the following steps:

1. The Graphical User Interface (App GUI layer) will call the function upon user requested in the Business Layer. The App GUI layer is visible to the user through screen layout and interface of the system.
2. The Business Layer will send a request to a specific web service and determines how data can be displayed, created, stored, and changed.
3. The web service will use the input data and query a record from the database. Web services provide a method of communication between software applications running on different platforms and frameworks.
4. The database will send the selected records back to the web service.
5. The web service will use the hashing forest based neuron retrieval algorithm and send the desired result back to the client.

We used the micro web framework Python-Flask to create web services for facilitating data processing and added an Android mobile application as a client. The framework's middle ware was defined as a RESTful web service, which enables usage by several clients i.e. web applications and mobile applications. Meanwhile, for data harvesting from different sources, a Harvester routine was defined. Currently the Harvester is only connected to the Spot (http://spot.mpdl.mpg.de/), which is an on line data hub, where users can share their data. A scheduler is defined in the program,

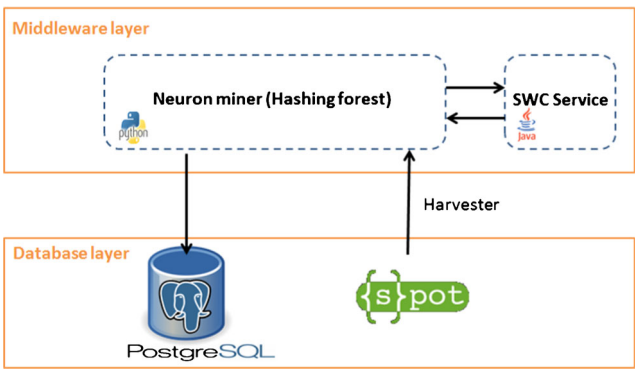

Fig. 9 Harvesting neuron-data from Spot and updating the database

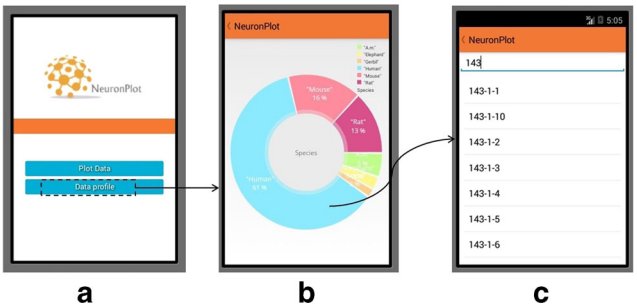

Fig. 10 Browsing neurons by animal species and searching for specific neurons. **a** navigating to the data profile section, **b** browse neurons by animal species **c** list of all the neurons, search for a specific neurons is possible through the search bar at the top

which runs the Harvester everyday to retrieve data from the Spot. Currently, a preliminary implementation of the Harvester routine is in place and its robust construction is work in progress. Once harvesting is done, the Neuron-Miner will parse the JSON file received from the Spot. Using the SWC service (https://servicehub.mpdl.mpg.de/), which uses Lmeasure, we can extract the morphological measurements of each neuron. Finally, we save the data to the PostgreSQL database and update the hash table for further analysis. Figure 9 schematically illustrates the work flow of the harvester routine. The Harvester routine updates the hash-table when a newly reconstructed neuron is added to database.

Client side

Mobile applications have their own limitations like, storage capacity, memory and CPU availability. In this case, it is better to assign the processing part to the web services and just display the results on a mobile platform. The user interacts with the presentation layer and sends a request through the business layer to the web service. Figures 10 and 11 are sample screen shots of the user interface of the Neuron-Miner tool. A user can perform browsing by choosing animal species and searching for a neuron. By selecting a neuron, the neuron's profile with the most important morphometric measurements will be presented. Finding similar neurons is possible through the Find Similar Neurons button in the interface as shown in Fig. 11a. The associated callback routine invokes retrieval algorithms at the middle ware layers and retrieval using HF is performed on the server side. The retrieved neurons are then sent back as response to the client application. The compiled tool is available as an Android application along with associated .apk installation file generated with the Android SDK tool.

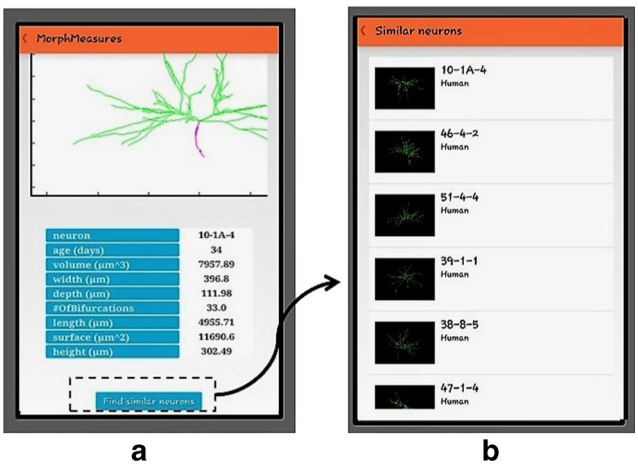

Fig. 11 Checking the profile of the neuron with some morphometric measurements and finding similar neurons. **a** the profile of a neuron, finding similar neurons to the selected neuron is possible through clicking the button , **b** the list of similar neurons

Conclusions and Future Work

In this paper, we present Neuron-Miner, a tool for data-mining from large heterogeneous neuroscientific image databases. The tool is aimed to facilitate efficient morphology - preserving search and retrieval of neurons. We propose to use hashing methods to parse and encode the neurons with binary similarity preserving code words generated by hashing functions. This is achieved by using hashing forests, which uses unsupervised random forests to extract compact representation of neuron morphological features that enables efficient query, retrieval, and analysis of neurons. The use of ensemble of trees and hierarchical tree-structure makes hashing forests more robust to noisy neuromorphological features (observed due to inconsistent 3D digital reconstruction of neuron). We establish that the proposed HF formulation has superior neighborhood approximation and retrieval performance in comparison to existing generalized hashing methods by quantifying the results over 31266 neuron reconstructions from Neuromorpho.org dataset curated from 147 different archives. To the best of our knowledge, this is the first research to present hashing in neuroscientific databases and demonstrates higher flexibility for reference-based retrieval over existing alternative methods (Search by Morphometry2015).

The proposed HF is developed to preserve morphological similarity while encoding the neuromorphological space, which has better performance *per* bit than the comparative methods. The formulation for HF is generic and it can be easily leveraged for other large-scale reference based retrieval systems. The proposed formulation utilizes inverse coding in HF, which helps avoiding pairwise comparisons across the database while retrieving, without compromising on accuracy. With the inclusion of oblique split functions in conjunction with cluster validity measures, we ensure that the native neighborhoods and clusters within the neuromorphological space are maximally preserved during hashing.

Towards the future work, the proposed Neuron-Miner tool is a beta-version and is not entirely feature complete. We will further customize and improve our mobile client along with its portability beyond Android platforms. Our future work towards improving efficiency of hashing forests includes investigating improvements to the node-scoring functions like introducing maximum-margin approaches to improve parsing of the neuromorphological space (Joly and Buisson 2011). Improvising scalability of the HF is also a direction for future pursuit (Yu and Yuan 2014). Further, we will introduce user defined search criterion by allowing users to select substructures of interest within a neuron and retrieve with a higher search specificity.

Information Sharing Statement

The Neuron-Miner tool for search and retrieval described in this paper is available as an app for Android OS 6.0 Marshmallow onwards and can be downloaded from https://servicehub.mpdl.mpg.de/Neuron-miner.apk. After successful installation of the app, it provides an interactive user-friendly interface for 3D-visualization and search and retrieval of neurons. The source code for hashing forests and the neuromorphological features dataset is available through Github upon acceptance of the paper.

Acknowledgments We thank Ajayrama Kumaraswamy, Computational Neuroscience Department Biology II, Ludwigs Maximillian Universität München, Germany for insightful discussion in the early conception of this work. We thank the assistance of Bastien Saquet of Max Plank Digital Library, München, Germany in maintaining the web-service. We would like to thank the Max Plank Digital Library, München, Germany for providing computing resources for hosting the Neuron-Miner software and making it publicly accessible.

Conflict of interests We have no conflict of interest to declare.

References

Albalate, A., & Suendermann, D. (2009). A combination approach to cluster validation based on statistical quantiles. In *2009. IJCBS'09. International Joint Conference on (pp. 549-555) Bioinformatics, Systems Biology and Intelligent Computing*: IEEE.

Ascoli, G.A., Donohue, D.E., & Halavi, M. (2007). Neuromorpho. Org: a central resource for neuronal morphologies. *The Journal of Neuroscience*, *27*(35), 9247–9251.

Costa, M., Ostrovsky, A.D., Manton, J.D., Prohaska, S., & Jefferis, G.S. (2014). NBLAST: Rapid, sensitive comparison of neuronal structure and construction of neuron family databases. bioRxiv, p.006346.

Costa, L.D.F., Zawadzki, K., Miazaki, M., Viana, M.P., & Taraskin, S. (2010). Unveiling the neuromorphological space. *Frontiers in Computational Neuroscience*, *4*, 150.

Desgraupes, B. (2013). Clustering indices. *University of Paris Ouest-Lab Modal'X*, *1*, 34.

Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. In *VLDB 99(6), p. 518-529*. Vancouver.

He, X., Cai, D., Yan, S., & Zhang, H.J. (2005). Neighborhood preserving embedding. In *2005. ICCV 2005. Tenth IEEE International Conference on (Vol. 2, pp. 1208-1213) Computer Vision*: IEEE.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, *24*(6), 417.

Joly, A., & Buisson, O. (2011). Random maximum margin hashing. In *2011 IEEE Conference on (pp. 873-880) Computer Vision and Pattern Recognition (CVPR)*: IEEE.

Kendall, M.G. (1948). Rank correlation methods. *Biometrika*, *44*(1/2), 298.

Kovács, F., Legány, C., & Babos, A. (2005). Cluster validity measurement techniques. In *Proceedings of the 6th International Symposium of Hungarian Researchers on Computational Intelligence* (pp. 18-19). Budapest.

Literature Search Main Results (2015). Available at: http://neuromorpho.org/neuroMorpho/LS_queryStatus.jsp, (Accessed: 09 February 2016).

Liu, X., Huang, L., Deng, C., Lu, J., & Lang, B. (2015). Multi-View Complementary hash tables for nearest neighbor search. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1107–1115).

Louppe, G. (2014). Understanding random forests: From theory to practice.arXiv preprint arXiv:1407.7502.

Menze, B.H., Kelm, B.M., Splitthoff, D.N., Koethe, U., & Hamprecht, F.A. (2011). On oblique random forests. In *Machine Learning and Knowledge Discovery in Databases (pp. 453-469)*: Springer Berlin Heidelberg.

Mesbah, S., Conjeti, S., Kumaraswamy, A., Rautenberg, P., Navab, N., & Katouzian, A. (2015). Hashing Forests for Morphological Search and Retrieval in Neuroscientific Image Databases. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015 (pp. 135-143)*: Springer International Publishing.

Overview of L-Measure (2015). Available at: http://cng.gmu.edu:8080/Lm/help/index.htm, (Accessed: 09 February 2016).

Polavaram, S., Gillette, T.A., Parekh, R., & Ascoli, G.A. (2014). Statistical analysis and data mining of digital reconstructions of dendritic morphologies. *Frontiers in Neuroanatomy*, *8*, 138.

Rautenberg, P.L., Grothe, B., & Felmy, F. (2009). Quantification of the three dimensional morphology of coincidence detector neurons in the medial superior olive of gerbils during late postnatal development. *Journal of Comparative Neurology*, *517*(3), 385–396.

Rautenberg, P.L., Kumaraswamy, A., Tejero-Cantero, A., Doblander, C., Norouzian, M.R., Kai, K., Jacobsen, H.A., Ai, H., Wachtler, T., & Ikeno, H. (2014). Neurondepot: keeping your colleagues in sync by combining modern cloud storage services, the local file system, and simple web applications. *Frontiers in Neuroinformatics*, *8*, 55.

Scorcioni, R., Polavaram, S., & Ascoli, G.A. (2008). L-measure: a web-accessible tool for the analysis, comparison and search of digital reconstructions of neuronal morphologies. *Nature protocols*, *3*(5), 866–876.

Search by Morphometry (2015). Available at: http://neuromorpho.org/neuroMorpho/MorphometrySearch.jsp.

Slaney, M., & Casey, M. (2008). Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, *25*(2), 128–131.

Scikit-learn: machine learning in Python – scikit-learn 0.16.1 documentation (2015) Available at: http://scikit-learn.org/stable/ (Accessed: 25 August 2015).

Wan, Y., Long, F., Qu, L., Xiao, H., Hawrylycz, M., Myers, E.W., & Peng, H. (2015). BlastNeuron for automated comparison, retrieval and clustering of 3D neuron morphologies. *Neuroinformatics*, *13*(4), 487–499.

Wang, J., Liu, W., Kumar, S., & Chang, S.F. (2016). Learning to hash for indexing big Data–A survey. *Proceedings of the IEEE*, *104*(1), 34–57.

Weiss, Y., Fergus, R., & Torralba, A. (2012). Multidimensional spectral hashing. In *Computer Vision–ECCV 2012 (pp. 340–353)*: Springer Berlin Heidelberg.

Yu, G., & Yuan, J. (2014). Scalable forest hashing for fast similarity search. In *2014 IEEE International Conference on (pp. 1-6) Multimedia and Expo (ICME)*: IEEE.

Zhang, D., Wang, J., Cai, D., & Lu, J. (2010). Self-taught hashing for fast similarity search. In *proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval (pp. 18-25)*: ACM.