

Improving Similarity Search with High-dimensional Locality-sensitive Hashing

Jaiyam Sharma and Saket Navlakha

Abstract—We propose a new class of data-independent locality-sensitive hashing (LSH) algorithms based on the fruit fly olfactory circuit. The fundamental difference of this approach is that, instead of assigning hashes as dense points in a low dimensional space, hashes are assigned in a high dimensional space, which enhances their separability. We show theoretically and empirically that this new family of hash functions is locality-sensitive and preserves rank similarity for inputs in any ℓ_p space. We then analyze different variations on this strategy and show empirically that they outperform existing LSH methods for nearest-neighbors search on six benchmark datasets. Finally, we propose a multi-probe version of our algorithm that achieves higher performance for the same query time, or conversely, that maintains performance of prior approaches while taking significantly less indexing time and memory. Overall, our approach leverages the advantages of separability provided by high-dimensional spaces, while still remaining computationally efficient.

Index Terms—locality-sensitive hashing, similarity search, neuroscience, fruit fly olfaction, algorithms, data-independent hashing

1 INTRODUCTION

SIMILARITY search is an essential problem in machine learning and information retrieval [1], [2]. The main challenge in this problem is to efficiently search a database to find the most similar items to a query item, under some measure of similarity. While efficient methods exist when searching within small databases (linear search) or when items are low dimensional (tree-based search [3], [4]), exact similarity search has remained challenging for large databases with high-dimensional items [5].

This has motivated *approximate* similarity search algorithms, under the premise that for many applications, finding reasonably similar items is “good enough” as long as they can be found quickly. One popular approximate similarity search algorithm is based on *locality-sensitive hashing* (LSH [6]). The main idea of many LSH approaches is to compute a low-dimensional hash for each input item, such that similar items in input space lie nearby in hash space for efficient retrieval.

In this work, we take inspiration from the fruit fly olfactory circuit [7] and consider a conceptually different hashing strategy, where instead of hashing items to lie densely in a low-dimensional space, items are hashed to lie in a high-dimensional space. We make the following contributions:

- 1) We present and analyze the locality-sensitive properties of a binary high-dimensional hashing scheme, called DenseFly. We also show that DenseFly outperforms previous algorithms on six benchmark datasets for nearest-neighbors retrieval;

- 2) We prove that FlyHash (from Dasgupta et al. [7]) preserves rank similarity under any ℓ_p norm, and empirically outperforms prior LSH schemes designed to preserve rank similarity;
- 3) We develop and analyze multi-probe versions of DenseFly and FlyHash that improve search performance while using similar space and time complexity as prior approaches. These multi-probe approaches offer a solution to an open problem stated by Dasgupta et al. [7] of learning efficient binning strategies for high-dimensional hashes, which is critical for making high-dimensional hashing usable for practical applications.

2 RELATED WORK

Dasgupta et al. [7] recently argued that the fruit fly olfactory circuit uses a variant of locality-sensitive hashing to associate similar behaviors with similar odors it experiences. When presented with an odor (a “query”), fruit flies must recall similar odors previously experienced in order to determine the most appropriate behavioral response. Such a similarity search helps to overcome noise and to generalize behaviors across similar stimuli.

The fly olfactory circuit uses a three-layer architecture to assign an input odor a hash, which corresponds to a set of neurons that fire whenever the input odor is experienced. The first layer consists of 50 odorant receptor neuron (ORN) types, each of which fires at a different rate for a given odor. An odor is thus initially encoded as a point in 50-dimensional space, \mathbb{R}_+^{50} . The distribution of firing rates for these 50 ORNs approximates an exponential distribution, whose mean depends on the odor’s concentration [8], [9]. The second layer consists of 50 projection neurons (PNs) that receive feed-forward input from the ORNs and recurrent inhibition from lateral inhibitory neurons. As a result, the distribution of firing rates for the 50 PNs is mean-centered,

- J. Sharma is with the Integrative Biology Laboratory, Salk Institute for Biological Studies, La Jolla, CA 92037.
E-mail: jaiyamsharma@gmail.com
- S. Navlakha is with the Integrative Biology Laboratory, Salk Institute for Biological Studies, La Jolla, CA 92037.
E-mail: navlakha@salk.edu

such that the mean firing rate of PNs is roughly the same for all odors and odor concentrations [10], [11], [12], [13]. The third layer *expands* the dimensionality: the 50 PNs connect to 2000 Kenyon cells (KCs) via a sparse, binary random projection matrix [14]. Each KC samples from roughly 6 random PNs, and sums up their firing rates. Each KC then provides feed-forward excitation to an inhibitory neuron call APL, which provides proportional feed-back inhibition to each KC. As a result of this winner-take-all computation, only the top $\sim 5\%$ of the highest firing KCs remain firing for the odor; the remaining 95% are silenced. This 5% corresponds to the hash of the odor. Thus, an odor is assigned a sparse point in a higher dimensionality space than it was originally encoded (from 50-dimensional to 2000-dimensional).

The fly evolved a unique combination of computational ingredients that have been explored piece-meal in other LSH hashing schemes. For example, SimHash [15], [16] and Super-bit LSH [17] use dense Gaussian random projections, and FastLSH [18] uses sparse Gaussian projections, both in low dimensions to compute hashes, whereas the fly uses sparse, binary random projections in high dimensions. Comitant LSH [19] uses high dimensionality, but admit to exponentially increasing computational costs as the dimensionality increases, whereas the complexity of our approach scales linearly. WTAHash [20] uses a local winner-take-all mechanism (see below), whereas the fly uses a global winner-take-all mechanism. Sparse projections have also been used within other hashing schemes [21], [22], [23], [24], [25], but with some differences. For example, Li et al. [24] and Achlioptas [26] use signed binary random projections instead of 0-1 random projections used here; Li et al. [24] also do not propose to expand the hash dimensionality.

3 METHODS

We consider two types of binary hashing schemes consisting of hash functions, h_1 and h_2 . The LSH function h_1 provides a distance-preserving embedding of items in d -dimensional input space to mk -dimensional binary hash space, where the values of m and k are algorithm specific and selected to make the space or time complexity of all algorithms comparable (Section 3.5). The function h_2 places each input item into a discrete bin for lookup. Formally:

Definition 1. A hash function $h_1 : \mathbb{R}^d \rightarrow \{0, 1\}^{mk}$ is called locality-sensitive if for any two input items $p, q \in \mathbb{R}^d$, $\Pr[h_1(p) = h_1(q)] = \text{sim}(p, q)$, where $\text{sim}(p, q) \in [0, 1]$ is a similarity measure between p and q .

Definition 2. A hash function $h_2 : \mathbb{R}^d \rightarrow [0, \dots, b]$ places each input item into a discrete bin.

Two disadvantages of using h_1 — be it low or high dimensional — for lookup are that some bins may be empty, and that true nearest-neighbors may lie in a nearby bin. This has motivated *multi-probe LSH* [27] where, instead of probing only the bin the query falls in, nearby bins are searched, as well.

Below we describe three existing methods for designing h_1 (SimHash, WTAHash, FlyHash) plus our method (DenseFly). We then describe our methods for providing low-dimensional binning for h_2 to FlyHash and DenseFly. All

algorithms described below are data-independent, meaning that the hash for an input is constructed without using any other input items. Overall, we propose a hybrid fly hashing scheme that takes advantage of high-dimensionality to provide better ranking of candidates, and low-dimensionality to quickly find candidate neighbors to rank.

3.1 SimHash

Charikar [15] proposed the following hashing scheme for generating a binary hash code for an input vector, x . First, mk (i.e., the hashing dimension) random projection vectors, r_1, r_2, \dots, r_{mk} , are generated, each of dimension d . Each element in each random projection vector is drawn uniformly from a Gaussian distribution, $\mathcal{N}(0, 1)$. Then, the i^{th} value of the binary hash is computed as:

$$h_1(x)_i = \begin{cases} 1 & \text{if } r_i \cdot x \geq 0 \\ 0 & \text{if } r_i \cdot x < 0. \end{cases} \quad (1)$$

This scheme preserves distances under the angular and Euclidean distance measures [16].

3.2 WTAHash (Winner-take-all hash)

Yagnik et al. [20] proposed the following binary hashing scheme. First, m permutations, $\theta_1, \theta_2, \dots, \theta_m$ of the input vector are computed. For each permutation i , we consider the first k components, and find the index of the component with the maximum value. C_i is then a zero vector of length k with a single 1 at the index of the component with the maximum value. The concatenation of the m vectors, $h_1(x) = [C_1, C_2, \dots, C_m]$ corresponds to the hash of x . This hash code is sparse — there is exactly one 1 in each successive block of length k — and by setting $mk > d$, hashes can be generated that are of dimension greater than the input dimension. We call k the WTA factor.

WTAHash preserves distances under the rank correlation measure [20]. It also generalizes MinHash [28], [29], and was shown to outperform several data-dependent LSH algorithms, including PCAHash [30], [31], spectral hash, and, by transitivity, restricted Boltzmann machines [32].

3.3 FlyHash and DenseFly

The two fly hashing schemes (Algorithm 1, Figure 1) first project the input vector into an mk -dimensional hash space using a sparse, binary random matrix, proven to preserve locality [7]. This random projection has a sampling rate of α , meaning that in each random projection, only $\lfloor ad \rfloor$ input indices are considered (summed). In the fly circuit, $\alpha \sim 0.1$ since each Kenyon cell samples from roughly 10% (6/50) of the projection neurons.

The first scheme, FlyHash [7], sparsifies and binarizes this representation by setting the indices of the top m elements to 1, and the remaining indices to 0. In the fly circuit, $k = 20$, since the top firing 5% of Kenyon cells are retained, and the rest are silenced by the APL inhibitory neuron. Thus, a FlyHash hash is an mk -dimensional vector with exactly m ones, as in WTAHash. However, in contrast to WTAHash, where the WTA is applied locally onto each block of length k , for FlyHash, the sparsification happens globally considering all mk indices together. We later prove

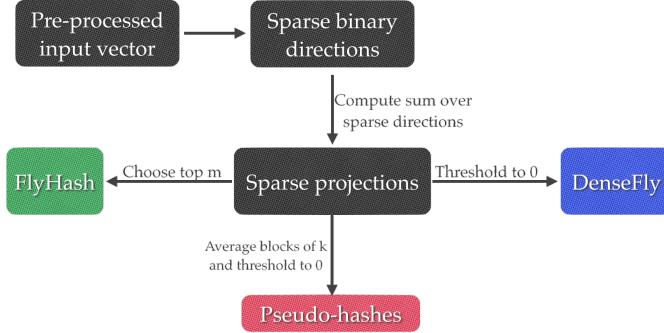


Fig. 1. Overview of the fly hashing algorithms.

(Lemma 3) that this difference allows more pairwise orders to be encoded within the same hashing dimension.

For FlyHash, the number of unique Hamming distances between two hashes is limited by the hash length m . Greater separability can be achieved if the number of 1s in the high-dimensional hash is allowed to vary. The second scheme, DenseFly, sparsifies and binarizes the representation by setting all indices with values ≥ 0 to 1, and the remaining to 0 (akin to SimHash though in high dimensions). We will show that this method provides even better separability than FlyHash in high dimensions.

3.4 Multi-probe hashing to find candidate nearest-neighbors

In practice, the most similar item to a query may have a similar, but not exactly the same, mk -dimensional hash as the query. In such a case, it is important to also identify candidate items with a similar hash as the query. Dasgupta et al. [7] did not propose a multi-probe binning strategy, without which their FlyHash algorithm is unusable in practice.

SimHash. For low-dimensional hashes, SimHash efficiently probes nearby hash bins using a technique called *multi-probe* [27]. All items with the same mk -dimensional hash are placed into the same bin; then, given an input x , the bin of $h_1(x)$ is probed, as well as all bins within Hamming distance r from this bin. This approach leads to large reductions in search space during retrieval as only bins which differ from the query point by r bits are probed. Notably, even though multi-probe avoids a linear search over all points in the dataset, a linear search over the bins themselves is unavoidable.

FlyHash and DenseFly. For high-dimensional hashes, multi-probe is even more essential because even if two input vectors are similar, it is unlikely that their high dimensional hashes will be exactly the same. For example, on the SIFT-1M dataset with a WTA factor $k = 20$ and $m = 16$, FlyHash produces about 860,000 unique hashes (about 86% the size of the dataset). In contrast, SimHash with $mk = 16$ produces about 40,000 unique hashes (about 4% the size of the dataset). Multi-probing directly in the high-dimensional space using the SimHash scheme, however, is unlikely to reduce the search space without spending significant time probing many nearby bins.

One solution to this problem is to use low-dimensional hashes to reduce the search space and quickly find candidate neighbors, and then to use high-dimensional hashes to rank these neighbors according to their similarity to the query. We introduce a simple algorithm for computing such low-dimensional hashes, called *pseudo-hashes* (Algorithm 1). To create an m -dimensional pseudo-hash of an mk -dimensional hash, we consider each successive block j of length k ; if the sum (or equivalently, the average) of the activations of this block is > 0 , we set the j^{th} bit of the pseudo-hash to 1, and 0 otherwise. Binning, then, can be performed using the same procedure as SimHash.

Given a query, we perform multi-probe on its low-dimensional pseudo-hash (h_2) to generate candidate nearest neighbors. Candidates are then ranked based on their Hamming distance to the query in high-dimensional hash space (h_1). Thus, our approach combines the advantages of low-dimensional probing and high-dimensional ranking of candidate nearest-neighbors.

Algorithm 1 FlyHash and DenseFly

Input: vector $x \in \mathbb{R}^d$, hash length m , WTA factor k , sampling rate α for the random projection.

```

# Generate mk sparse, binary random projections by
# summing from [ad] random indices each.
S = {Si | Si = rand([ad], d)}, where |S| = mk

# Compute high-dimensional hash, h1.
for j = 1 to mk do
    a(x)j = ∑i ∈ Sj xi # Compute activations
end for

if FlyHash then
    h1(x) = WTA(a(x)) ∈ {0, 1}mk # Winner-take-all
else if DenseFly then
    h1(x) = sgn(a(x)) ∈ {0, 1}mk # Threshold at 0
end if

# Compute low-dimensional pseudo-hash (bin), h2.
for j = 1 to m do
    p(x)j = sgn(∑u=k(j-1)+1u=kj a(x)u/k)
end for
h2(x) = g(p(x)) ∈ [0, ..., b] # Place in bin
  
```

Note: The function $\text{rand}(a, b)$ returns a set of a random integers in $[0, b]$. The function $g(\cdot)$ is a conventional hash function used to place a pseudo-hash into a discrete bin.

WTAGhash. To our knowledge, there is no method for doing multi-probe with WTAGhash. Pseudo-hashing cannot be applied for WTAGhash because there is a 1 in every block of length k , hence all psuedo-hashes will be a 1-vector of length m .

3.5 Strategy for comparing algorithms

We adopt a strategy for fairly comparing two algorithms by equating either their computational cost or their hash dimensionality, as described below.

Selecting hyperparameters. We consider hash lengths $m \in [16, 128]$. We compare all algorithms using $k = 4$, which was reported to be optimal by Yagnik et al. [20] for WTAHash, and $k = 20$, which is used by the fly circuit (i.e., only the top 5% of Kenyon cells fire for an odor).

Comparing SimHash versus FlyHash. SimHash random projections are more expensive to compute than FlyHash random projections; this additional expense allows us to compute more random projections (i.e., higher dimensionality) while not increasing the computational cost of generating a hash. Specifically, for an input vector x of dimension d , SimHash computes the dot product of x with a dense Gaussian random matrix. Computing the value of each hash dimension requires $2d$ operations: d multiplications plus d additions. FlyHash (effectively) computes the dot product of x with a sparse binary random matrix, with sampling rate α . Each dimension requires $\lfloor \alpha d \rfloor$ addition operations only (no multiplications are needed). Using $\alpha = 0.1$, as per the fly circuit, to equate the computational cost of both algorithms, the Fly is afforded $k = 20$ additional hashing dimensions. Thus, for SimHash, $mk = m$ (i.e., $k = 1$) and for FlyHash, $mk = 20m$. The number of ones in the hash for each algorithm may be different. In experiments with $k = 4$, we keep $\alpha = 0.1$, meaning that both fly-based algorithms have $1/5^{\text{th}}$ the computational complexity as SimHash.

Comparing WTAHash versus FlyHash. Since WTAHash does not use random projections, it is difficult to equate the computational cost of generating hashes. Instead, to compare WTAHash and FlyHash, we set the hash dimensionality and the number of 1s in each hash to be equal. Specifically, for WTAHash, we compute m permutations of the input, and consider the first k components of each permutation. This produces a hash of dimension mk with exactly m ones. For FlyHash, we compute mk random projections, and set the indices of the top m dimensions to 1.

Comparing FlyHash versus DenseFly. DenseFly computes sparse binary random projections akin to FlyHash, but unlike FlyHash, it does not apply a WTA mechanism but rather uses the sign of the activations to assign a value to the bit, like SimHash. To fairly compare FlyHash and DenseFly, we set the hashing dimension (mk) to be the same to equate the computational complexity of generating hashes, though the number of ones may differ.

Comparing multi-probe hashing. SimHash uses low-dimensional hashes to both build the hash index and to rank candidates (based on Hamming distances to the query hash) during retrieval. DenseFly uses pseudo-hashes of the same low dimensionality as SimHash to create the index; however, unlike SimHash, DenseFly uses the high-dimensional hashes to rank candidates. Thus, once the bins and indices are computed, the pseudo-hashes do not need to be stored. A pseudo-hash for a query is only used to determine which bin to look in to find candidate neighbors.

3.6 Evaluation datasets and metrics

Datasets. We evaluate each algorithm on six datasets (Table 1). There are three datasets with a random subset of 10,000 inputs each (GLoVE, LabelMe, MNIST), and two datasets with 1 million inputs each (SIFT-1M, GIST-1M). We

also included a dataset of 10,000 random inputs, where each input is a 128-dimensional vector drawn from a uniform random distribution, $\mathcal{U}(0, 1)$. This dataset was included because it has no structure and presents a worst-case empirical analysis. For all datasets, the only pre-processing step used is to center each input vector about the mean.

TABLE 1
Datasets used in the evaluation.

Dataset	Size	Dimension	Reference
Random	10,000	128	—
GLoVe	10,000	300	Pennington et al. [33]
LabelMe	10,000	512	Russell et al. [34]
MNIST	10,000	784	Lecun et al. [35]
SIFT-1M	1,000,000	128	Jegou et al. [36]
GIST-1M	1,000,000	960	Jegou et al. [36]

Accuracy in identifying nearest-neighbors. Following Yagnik et al. [20] and Weiss et al. [32], we evaluate each algorithm’s ability to identify nearest neighbors using two performance metrics: area under the precision-recall curve (AUPRC) and mean average precision (mAP). For all datasets, following Jin et al. [37], given a query point, we computed a ranked list of the top 2% of true nearest neighbors (excluding the query) based on Euclidean distance between vectors in input space. Each hashing algorithm similarly generates a ranked list of predicted nearest neighbors based on Hamming distance between hashes (h_1). We then compute the mAP and AUPRC on the two ranked lists. Means and standard deviations are calculated over 500 runs.

Time and space complexity. While mAP and AUPRC evaluate the quality of hashes, in practice, such gains may not be practically usable if constraints such as query time, indexing time, and memory usage are not met. We use two approaches to evaluate the time and space complexity of each algorithm’s multi-probe version (h_2).

The goal of the first evaluation is to test how the mAP of SimHash and DenseFly fare under the same query time. For each algorithm, we hash the query to a bin. Bins nearby the query bin are probed with an increasing search radius. For each radii, the mAP is calculated for the ranked candidates. As the search radius increases, more candidates are pooled and ranked, leading to larger query times and larger mAP scores.

The goal of the second evaluation is to roughly equate the performance (mAP and query time) of both algorithms and compare the time to build the index and the memory consumed by the index. To do this, we note that to store the hashes, DenseFly requires k times more memory to store the high-dimensional hashes. Thus, we allow SimHash to pool candidates from k independent hash tables while using only 1 hash table for DenseFly. While this ensures that both algorithms use roughly the same memory to store hashes, SimHash also requires: (a) k times the computational complexity of DenseFly to generate k hash tables, (b) roughly k times more time to index the input vectors to bins for each hash table, and (c) more memory for storing bins and indices. Following Lv et al. [27], we evaluate mAP at a fixed

number of nearest neighbors (100). As before, each query is hashed to a bin. If the bin has ≥ 100 candidates, we stop and rank these candidates. Else, we keep increasing the search radius by 1 until we have least 100 candidates to rank. We then rank all candidates and compute the mAP versus the true 100 nearest-neighbors. Each algorithm uses the minimal radius required to identify 100 candidates (different search radii may be used by different algorithms).

4 RESULTS

First, we present theoretical analysis of the DenseFly and FlyHash high-dimensional hashing algorithms, proving that DenseFly generates hashes that are locality-sensitive according to Euclidean and cosine distances, and that FlyHash preserves rank similarity for any ℓ_p norm; we also prove that pseudo-hashes are effective for reducing the search space of candidate nearest-neighbors without increasing computational complexity. Second, we evaluate how well each algorithm identifies nearest-neighbors using the hash function, h_1 , based on its query time, computational complexity, memory consumption, and indexing time. Third, we evaluate the multi-probe versions of SimHash, FlyHash, and DenseFly (h_2).

4.1 Theoretical analysis of high-dimensional hashing algorithms

Lemma 1. *DenseFly generates hashes that are locality-sensitive.*

Proof. The idea of the proof is to show that DenseFly approximates a high-dimensional SimHash, but at k times lower computational cost. Thus, by transitivity, DenseFly preserves cosine and Euclidean distances, as shown for SimHash [16].

The set S (Algorithm 1), containing the indices that each Kenyon cell (KC) samples from, can be represented as a sparse binary matrix, M . In Algorithm 1, we fixed each column of M to contain exactly $\lfloor \alpha d \rfloor$ ones. However, maintaining exactly $\lfloor \alpha d \rfloor$ ones is not necessary for the hashing scheme, and in fact, in the fly's olfactory circuit, the number of projection neurons sampled by each KC is approximately a binomial distribution with a mean of 6 [13], [14]. Suppose the projection directions in the fly's hashing schemes (FlyHash and DenseFly) are sampled from a binomial distribution; i.e., let $M \in \{0, 1\}^{dmk}$ be a sparse binary matrix whose elements are sampled from dmk independent Bernoulli trials each with success probability α , so that the total number of successful trials follows $B(dmk, \alpha)$. Pseudo-hashes are calculated by averaging m blocks of k sparse projections. Thus, the expected activation of Kenyon cell j to input x is:

$$\mathbb{E}[a_{\text{DenseFly}}(x)_j] = \mathbb{E}\left[\sum_{u=k(j-1)+1}^{u=kj} \sum_i M_{ui} x_i / k\right]. \quad (2)$$

Using the linearity of expectation,

$$\mathbb{E}[a_{\text{DenseFly}}(x)_j] = k \mathbb{E}\left[\sum_i M_{ui} x_i\right] / k,$$

where u is any arbitrary index in $[1, mk]$. Thus, $\mathbb{E}[a_{\text{DenseFly}}(x)_j] = \alpha \sum_i x_i$, as $m \rightarrow \infty$. The expected value

of a DenseFly activation is given in Equation (2) with special condition that $k = 1$.

Similarly, the projection directions in SimHash are sampled from a Gaussian distribution; i.e., let $M^D \in \mathbb{R}^{d \times m}$ be a dense matrix whose elements are sampled from $\mathcal{N}(\mu, \sigma)$. Using linearity of expectation, the expected value of the j^{th} SimHash projection to input x is:

$$\mathbb{E}[a_{\text{SimHash}}(x)_j] = \mathbb{E}\left[\sum_i M_{ji}^D x_i\right] = \mu \sum_i x_i.$$

Thus, $\mathbb{E}[a_{\text{DenseFly}}(x)_j] = \mathbb{E}[a_{\text{SimHash}}(x)_j] \forall j \in [1, m]$ if $\mu = \alpha$.

In other words, sparse activations of DenseFly approximate the dense activations of SimHash as the hash dimension increases. Thus, a DenseFly hash approximates SimHash of dimension mk . In practice, this approximation works well even for small values of m since hashes depend only on the sign of the activations. \square

We supported this result by empirical analysis showing that the AUPRC for DenseFly is very similar to that of SimHash when using equal dimensions (Figure S1). DenseFly, however, takes k -times less computation. In other words, we proved that the computational complexity of SimHash could be reduced k -fold while still achieving the same performance.

We next analyze how FlyHash preserves a popular similarity measure for nearest-neighbors, called rank similarity [20], and how FlyHash better separates items in high-dimensional space compared to WTAHash (which was designed for rank similarity). Dasgupta et al. [7] did not analyze FlyHash for rank similarity neither theoretically nor empirically.

Lemma 2. *FlyHash preserves rank similarity of inputs under any ℓ_p norm.*

Proof. The idea is to show that small perturbations to an input vector does not affect its hash.

Consider an input vector x of dimensionality d whose hash of dimension mk is to be computed. The activation of the j^{th} component (Kenyon cell) in the hash is given by $a_j = \sum_{i \in S_j} x_i$, where S_j is the set of dimensions of x that the j^{th} Kenyon cell samples from. Consider a perturbed version of the input, $x' = x + \delta x$, where $\|\delta x\|_p = \epsilon$. The activity of the j^{th} Kenyon cell to the perturbed vector x' is given by:

$$a'_j = \sum_{i \in S_j} x'_i = a_j + \sum_{i \in S_j} \delta x_i.$$

By the method of Lagrange multipliers, $|a'_j - a_j| \leq d\alpha\epsilon / \sqrt[p]{d\alpha} \forall j$ (Supplement). Moreover, for any index $u \neq j$,

$$||a'_j - a'_u| - |a_j - a_u|| \leq |(a'_j - a'_u) - (a_j - a_u)| \leq 2d\alpha\epsilon / \sqrt[p]{d\alpha}.$$

In particular, let j be the index of $h_1(x)$ corresponding to the smallest activation in the 'winner' set of the hash (i.e., the smallest activation such that its bit in the hash is set to 1). Conversely, let u be the index of $h_1(x)$ corresponding to the largest activation in the 'loser' set of the hash. Let $\beta = a_j - a_u > 0$. Then,

$$\beta - 2d\alpha\epsilon / \sqrt[p]{d\alpha} \leq |a'_j - a'_u| \leq \beta + 2d\alpha\epsilon / \sqrt[p]{d\alpha}.$$

For $\epsilon < \beta\sqrt[3]{d\alpha}/2d\alpha$, it follows that $(a'_j - a'_u) \in [\beta - 2d\alpha\epsilon/\sqrt[3]{d\alpha}, \beta + 2d\alpha\epsilon/\sqrt[3]{d\alpha}]$. Thus, $a'_j > a'_u$. Since, j and u correspond to the lowest difference between the elements of the winner and loser sets, it follows that all other pairwise rank orders defined by FlyHash are also maintained. Thus, FlyHash preserves rank similarity between two vectors whose distance in input space is small. As ϵ increases, the partial order corresponding to the lowest difference in activations is violated first leading to progressively higher Hamming distances between the corresponding hashes. \square

Lemma 3. *FlyHash encodes m -times more pairwise orders than WTAHash for the same hash dimension.*

Proof. The idea is that WTAHash imposes a local constraint on the winner-take-all (exactly one 1 in each block of length k), whereas FlyHash uses a global winner-take-all, which allows FlyHash to encode more pairwise orders.

We consider the pairwise order function $PO(X, Y)$ defined by Yagnik et al. [20], where (X, Y) are the WTA hashes of inputs (x, y) . In simple terms, $PO(X, Y)$ is the number of inequalities on which the two hashes X and Y agree.

To compute a hash, WTAHash concatenates pairwise orderings for m independent permutations of length k . Let i be the index of the 1 in a given permutation. Then, $x_i \geq x_j \forall j \in [1, k] \setminus \{i\}$. Thus, a WTAHash denotes $m(k-1)$ pairwise orderings. The WTA mechanism of FlyHash encodes pairwise orderings for the top m elements of the activations, a . Let W be the set of the top m elements of a as defined in Algorithm 1. Then, for any $j \in W$, $a_j \geq a_i \forall i \in [1, mk] \setminus W$. Thus, each $j \in W$ denotes $m(k-1)$ inequalities, and FlyHash encodes $m^2(k-1)$ pairwise orderings. Thus, the pairwise order function for FlyHash encodes m times more orders. \square

Empirically, we found that FlyHash and DenseFly achieved much higher Kendall- τ rank correlation than WTAHash, which was specifically designed to preserve rank similarity [20] (Results, Table 3). This validates our theoretical results.

Lemma 4. *Pseudo-hashes approximate SimHash with increasing WTA factor k .*

Proof. The idea is that expected activations of pseudo-hashes calculated from sparse projections is the same as the activations of SimHash calculated from dense projections.

The analysis of Equation (2) can be extended to show that pseudo-hashes approximate SimHash of the same dimensionality. Specifically,

$$\mathbb{E}[a_{pseudo}(x)_j] = \alpha \sum_i x_i, \text{ as } k \rightarrow \infty.$$

Similarly, the projection directions in SimHash are sampled from a Gaussian distribution; i.e., let $M^D \in \mathbb{R}^{d \times m}$ be a dense matrix whose elements are sampled from $\mathcal{N}(\mu, \sigma)$. Using linearity of expectation, the expected value of the j^{th} SimHash projection is:

$$\mathbb{E}[a_{SimHash}(x)_j] = \mathbb{E}\left[\sum_i M_{ji}^D x_i\right] = \mu \sum_i x_i.$$

Thus, $\mathbb{E}[a_{pseudo}(x)_j] = \mathbb{E}[a_{SimHash}(x)_j] \forall j \in [1, m]$ if $\mu = \alpha$. Similarly, the variances of $a_{SimHash}(x)$ and $a_{pseudo}(x)$

are equal if $\sigma^2 = \alpha(1 - \alpha)$. Thus, SimHash itself can be interpreted as the pseudo-hash of a FlyHash with very large dimensions. \square

Although in theory, this approximation holds for only large values of k , in practice the approximation can operate under a high degree of error since equality of hashes requires only that the sign of the activations of pseudo-hash be the same as that of SimHash.

Empirically, we found that the performance of only using pseudo-hashes (not using the high-dimensional hashes) for ranking nearest-neighbors performs similarly with SimHash for values of k as low as $k = 4$ (Figure S2 and S3), confirming our theoretical results. Notably, the computation of pseudo-hashes is performed by re-using the activations for DenseFly, as explained in Algorithm 1 and Figure 1. Thus, pseudo-hashes incur little computational cost and provide an effective tool for reducing the search space due to their low dimensionality.

4.2 Empirical evaluation of low- versus high-dimensional hashing

We compared the quality of the hashes (h_1) for identifying the nearest-neighbors of a query using the four 10k-item datasets (Figure 2A). For nearly all hash lengths, DenseFly outperforms all other methods in area under the precision-recall curve (AUPRC). For example, on the GLoVE dataset with hash length $m = 64$ and WTA factor $k = 20$, the AUPRC of DenseFly is about three-fold higher than SimHash and WTAHash, and almost two-fold higher than FlyHash (DenseFly=0.395, FlyHash=0.212, SimHash=0.106, WTAHash=0.112). On the Random dataset, which has no inherent structure, DenseFly provides a higher degree of separability in hash space compared to FlyHash and WTAHash, especially for large k (e.g., nearly 0.440 AUPRC for DenseFly versus 0.140 for FlyHash, 0.037 for WTAHash, and 0.066 for SimHash with $k = 20, m = 64$). Figure 2B shows empirical performance for all methods using $k = 4$, which shows similar results.

DenseFly also outperforms the other algorithms in identifying nearest neighbors on two larger datasets with 1M items each (Figure 3). For example, on SIFT-1M with $m = 64$ and $k = 20$, DenseFly achieves 2.6x/2.2x/1.3x higher AUPRC compared to SimHash, WTAHash, and FlyHash, respectively. These results demonstrate the promise of high-dimensional hashing on practical datasets.

4.3 Evaluating multi-probe hashing

Here, we evaluated the multi-probing schemes of SimHash and DenseFly (pseudo-hashes). Using $k = 20$, DenseFly achieves higher mAP for the same query time (Figure 4A). For example, on the GLoVE dataset, with a query time of 0.01 seconds, the mAP of DenseFly is 91.40% higher than that of SimHash, with similar gains across other datasets. Thus, the high-dimensional DenseFly is better able to rank the candidates than low-dimensional SimHash. Figure 4B shows that similar results hold for $k = 4$; i.e., DenseFly achieves higher mAP for the same query time as SimHash.

Next, we evaluated the multi-probe schemes of SimHash, FlyHash (as originally conceived by Dasgupta

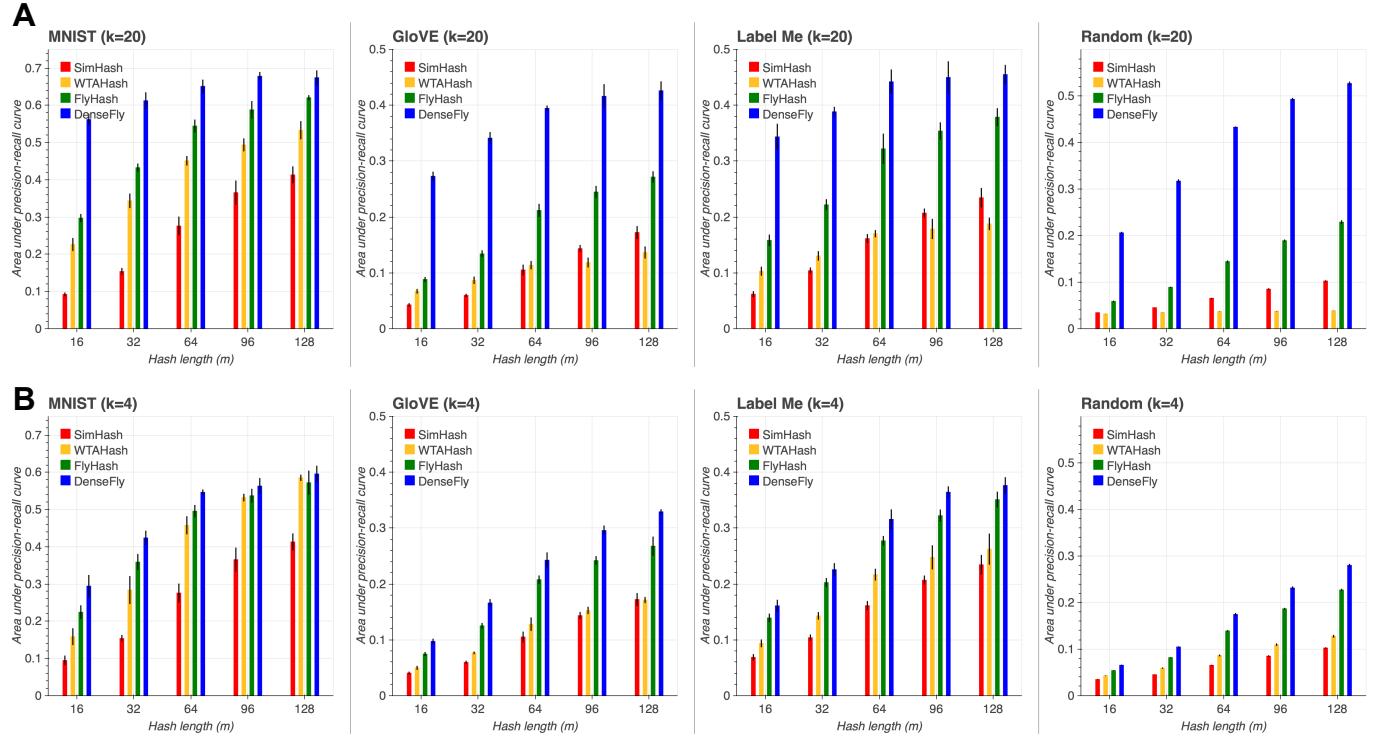


Fig. 2. Precision-recall for the MNIST, GLoVE, LabelMe, and Random datasets. A) $k = 20$. B) $k = 4$. In each panel, the x -axis is the hash length, and the y -axis is the area under the precision-recall curve (higher is better). For all datasets and hash lengths, DenseFly performs the best.

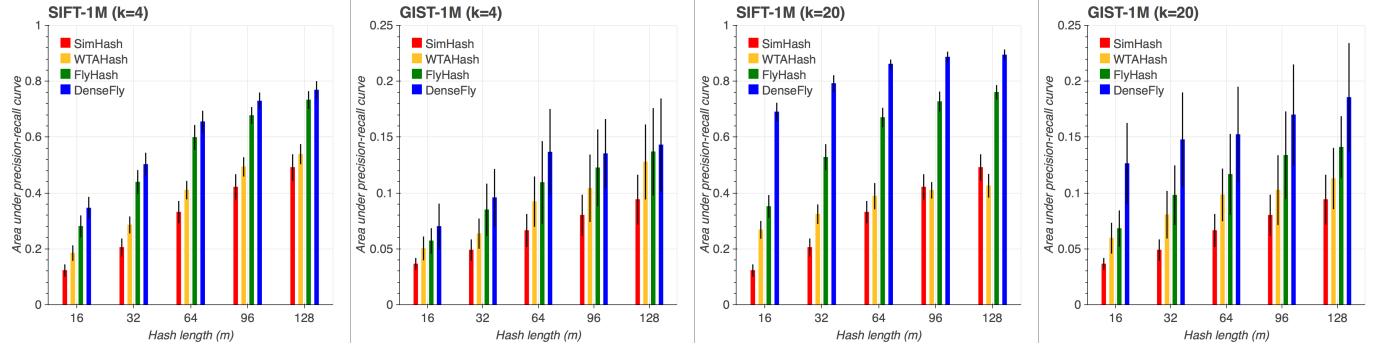


Fig. 3. Precision-recall for the SIFT-1M and GIST-1M datasets. In each panel, the x -axis is the hash length, and the y -axis is the area under the precision-recall curve (higher is better). The first two panels show results for SIFT-1M and GIST-1M using $k = 4$; the latter two show results for $k = 20$. DenseFly is comparable to or outperforms all other algorithms.

et al. [7] without multi-probe), our FlyHash multi-probe version (called FlyHash-MP), and DenseFly based on mAP as well as query time, indexing time, and memory usage. To boost the performance of SimHash, we pooled and ranked candidates over k independent hash tables as opposed to 1 table for DenseFly (Section 3.6). Table 2 shows that for nearly the same mAP as SimHash, DenseFly significantly reduces query times, indexing times, and memory consumption. For example, on the Glove-10K dataset, DenseFly achieves marginally lower mAP compared to SimHash (0.966 vs. 1.000) but requires only a fraction of the querying time (0.397 vs. 1.000), indexing time (0.239 vs. 1.000) and memory (0.381 vs. 1.000). Our multi-probe FlyHash algorithm improves over the original FlyHash, but it still produces lower mAP compared to DenseFly. Thus, DenseFly more efficiently

identifies a small set of high quality candidate nearest-neighbors for a query compared to the other algorithms.

4.4 Empirical analysis of rank correlation for each method

Finally, we empirically compared DenseFly, FlyHash, and WTAHash based on how well they preserved rank similarity [20]. For each query, we calculated the ℓ_2 distances of the top 2% of true nearest neighbors. We also calculated the Hamming distances between the query and the true nearest-neighbors in hash space. We then calculated the Kendall- τ rank correlation between these two lists of distances. Across all datasets and hash lengths tested, DenseFly outperformed both FlyHash and WTAHash (Table 3), confirming our theoretical results.

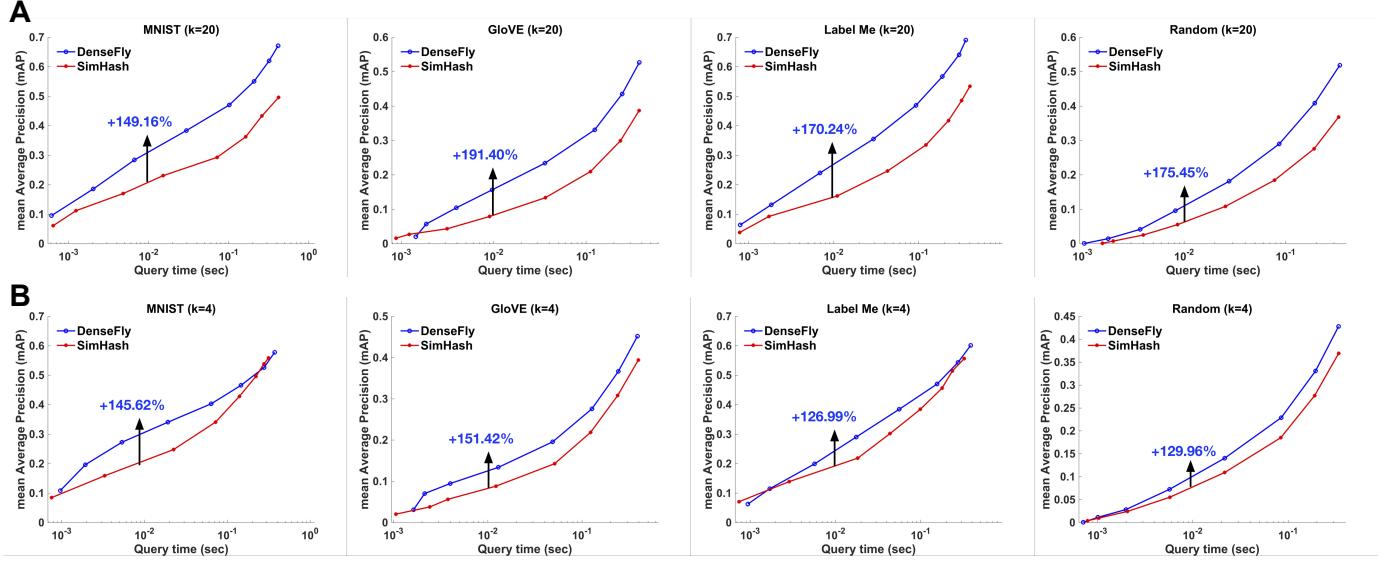


Fig. 4. Query time versus mAP for the 10k-item datasets. A) $k = 20$. B) $k = 4$. In each panel, the x -axis is query time, and the y -axis is mean average precision (higher is better) of ranked candidates using a hash length $m = 16$. Each successive dot on each curve corresponds to an increasing search radius. For nearly all datasets and query times, DenseFly with pseudo-hash binning performs better than SimHash with multi-probe binning. The arrow in each panel indicates the gain in performance for DenseFly at a query time of 0.01 seconds.

TABLE 2

Performance of multi-probe hashing for four datasets. Across all datasets, DenseFly achieves similar mAP as SimHash, but with 2x faster query times, 4x fewer hash tables, 4–5x less indexing time, and 2–4x less memory usage. FlyHash-MP evaluates our multi-probe technique applied to the original FlyHash algorithm. DenseFly and FlyHash-MP require similar indexing time and memory, but DenseFly achieves higher mAP. FlyHash without multi-probe ranks the entire database per query; it therefore does not build an index and has large query times.

Performance is shown normalized to that of SimHash. We used WTA factor, $k = 4$ and hash length, $m = 16$.

Dataset	Algorithm	# Tables	mAP @ 100	Query	Indexing	Memory
GIST-100k	SimHash	4	1.000	1.000	1.000	1.000
	DenseFly	1	0.947	0.537	0.251	0.367
	FlyHash-MP	1	0.716	0.515	0.252	0.367
	FlyHash	1	0.858	5.744	0.000	0.156
MNIST-10k	SimHash	4	1.000	1.000	1.000	1.000
	DenseFly	1	0.996	0.669	0.226	0.381
	FlyHash-MP	1	0.909	0.465	0.232	0.381
	FlyHash	1	0.985	1.697	0.000	0.174
LabelMe-10k	SimHash	4	1.000	1.000	1.000	1.000
	DenseFly	1	1.075	0.481	0.242	0.383
	FlyHash-MP	1	0.869	0.558	0.250	0.383
	FlyHash	1	0.868	2.934	0.000	0.177
GLOVE-10k	SimHash	4	1.000	1.000	1.000	1.000
	DenseFly	1	0.966	0.397	0.239	0.381
	FlyHash-MP	1	0.950	0.558	0.241	0.381
	FlyHash	1	0.905	1.639	0.000	0.174

5 CONCLUSIONS

We analyzed and evaluated a new family of neural-inspired binary locality-sensitive hash functions that perform better than existing data-independent methods (SimHash, WTA-Hash, FlyHash) across several datasets and evaluation metrics. The key insight is to use efficient projections to generate high-dimensional hashes, which we showed can be done without increasing computation or space complexity. We proved theoretically that DenseFly is locality-sensitive under the Euclidean and cosine distances, and that FlyHash preserves rank similarity for any ℓ_p norm. We also proposed a multi-probe version of our algorithm that offers an effi-

cient binning strategy for high-dimensional hashes, which is important for making this scheme usable in practical applications. Our method also performs well with only 1 hash table, which also makes this approach easier to deploy in practice. Overall, our results support findings that dimensionality expansion may be a “blessing” [38], [39], [40], especially for promoting separability for nearest-neighbors search.

There are many directions for future work. First, we focused on data-independent algorithms; biologically, the fly can “learn to hash” [41] but learning occurs online using reinforcement signals, as opposed to offline from a fixed

TABLE 3

Kendall- τ rank correlations for all 10k-item datasets. Across all datasets and hash lengths, DenseFly achieves a higher rank correlation between ℓ_2 distance in input space and ℓ_1 distance in hash space. Averages and standard deviations are shown over 100 queries. All results shown are for WTA factor, $k = 20$. Similar performance gains for DenseFly over other algorithms with $k = 4$ (not shown).

Dataset	Hash Length	WTAHash	FlyHash	DenseFly
MNIST	16	0.204 ± 0.10	0.288 ± 0.10	0.425 ± 0.08
	32	0.276 ± 0.10	0.375 ± 0.10	0.480 ± 0.13
	64	0.333 ± 0.10	0.446 ± 0.11	0.539 ± 0.12
GLoVE	16	0.157 ± 0.10	0.189 ± 0.10	0.281 ± 0.11
	32	0.169 ± 0.10	0.224 ± 0.11	0.306 ± 0.11
	64	0.183 ± 0.11	0.243 ± 0.13	0.311 ± 0.13
LabelMe	16	0.141 ± 0.08	0.174 ± 0.08	0.282 ± 0.08
	32	0.157 ± 0.08	0.227 ± 0.09	0.342 ± 0.11
	64	0.191 ± 0.09	0.292 ± 0.10	0.368 ± 0.10
Random	16	0.037 ± 0.06	0.089 ± 0.05	0.184 ± 0.04
	32	0.043 ± 0.05	0.120 ± 0.05	0.226 ± 0.05
	64	0.051 ± 0.04	0.155 ± 0.05	0.290 ± 0.04

database [42]. Second, we fixed the sampling rate $\alpha = 0.10$, as per the fly circuit; however, more work is needed to understand how optimal sampling complexity changes with respect to input statistics and noise. Third, most prior work on multi-probe LSH have assumed that hashes are low-dimensional; while pseudo-hashes represent one approach for binning high-dimensional data via a low-dimensional intermediary, more work is needed to explore other possible strategies. Fourth, there are methods to speed-up random projection calculations, for both Gaussian matrices [18], [25] and sparse binary matrices, which can be applied in practice.

6 CODE AVAILABILITY

Source code for all algorithms is available at: <http://www.github.com/dataplayer12/Fly-LSH>

REFERENCES

- [1] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008.
- [2] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *CoRR*, vol. arXiv:1408.2927, 2014.
- [3] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.
- [4] T. Liu, A. W. Moore, A. Gray, and K. Yang, "An investigation of practical approximate nearest neighbor algorithms," in *Proc. of the 17th Intl. Conf. on Neural Information Processing Systems*, ser. NIPS '04, 2004, pp. 825–832.
- [5] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. of the Intl. Conf. on Very Large Data Bases*, ser. VLDB '99, 1999, pp. 518–529.
- [6] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. of the Annual ACM Symposium on Theory of Computing*, ser. STOC '98, 1998, pp. 604–613.
- [7] S. Dasgupta, C. F. Stevens, and S. Navlakha, "A neural algorithm for a fundamental computing problem," *Science*, vol. 358, no. 6364, pp. 793–796, 11 2017.
- [8] E. A. Hallem and J. R. Carlson, "Coding of odors by a receptor repertoire," *Cell*, vol. 125, no. 1, pp. 143–160, Apr 2006.
- [9] C. F. Stevens, "A statistical property of fly odor responses is conserved across odors," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 113, no. 24, pp. 6737–6742, Jun 2016.
- [10] C. M. Root, K. Masuyama, D. S. Green, L. E. Enell, D. R. Nassel, C. H. Lee, and J. W. Wang, "A presynaptic gain control mechanism fine-tunes olfactory behavior," *Neuron*, vol. 59, no. 2, pp. 311–321, Jul 2008.
- [11] K. Asahina, M. Louis, S. Piccinotti, and L. B. Vosshall, "A circuit supporting concentration-invariant odor perception in *Drosophila*," *J. Biol.*, vol. 8, no. 1, p. 9, 2009.
- [12] S. R. Olsen, V. Bhandawat, and R. I. Wilson, "Divisive normalization in olfactory population codes," *Neuron*, vol. 66, no. 2, pp. 287–299, Apr 2010.
- [13] C. F. Stevens, "What the fly's nose tells the fly's brain," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 112, no. 30, pp. 9460–9465, Jul 2015.
- [14] S. J. Caron, V. Ruta, L. F. Abbott, and R. Axel, "Random convergence of olfactory inputs in the *Drosophila* mushroom body," *Nature*, vol. 497, no. 7447, pp. 113–117, May 2013.
- [15] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. of the Annual ACM Symposium on Theory of Computing*, ser. STOC '02, 2002, pp. 380–388.
- [16] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. of the 20th Annual ACM Symposium on Computational Geometry*, ser. SCG '04, 2004, pp. 253–262.
- [17] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian, "Super-bit locality-sensitive hashing," in *Proc. of the Intl. Conf. on Neural Information Processing Systems*, ser. NIPS '12, 2012, pp. 108–116.
- [18] A. Dasgupta, R. Kumar, and T. Sarlos, "Fast locality-sensitive hashing," in *Proc. of the 17th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 1073–1081.
- [19] K. Eshghi and S. Rajaram, "Locality sensitive hash functions based on concomitant rank order statistics," in *Proc. of the 14th ACM Intl. Conf. on Knowledge Discovery and Data Mining*, ser. KDD '08, 2008, pp. 221–229.
- [20] J. Yagnik, D. Strele, D. A. Ross, and R. Lin, "The power of comparative reasoning," in *Proc. of the Intl. Conf. on Computer Vision*, ser. ICCV '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 2431–2438.
- [21] D. Kane and J. Nelson, "Sparsified Johnson-Lindenstrauss transforms," *Journal of the Association for Computing Machinery*, vol. 61, no. 1, 2014.
- [22] Z. Allen-Zhu, R. Gelashvili, S. Micali, and N. Shavit, "Sparse sign-consistent Johnson-Lindenstrauss matrices: compression with neuroscience-based constraints," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 111, no. 47, pp. 16 872–16 876, Nov 2014.
- [23] Q. Shi, J. Pettersson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan, "Hash kernels for structured data," *J. Mach. Learn. Res.*, vol. 10, pp. 2615–2637, Dec. 2009.

- [24] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 287–296. [Online]. Available: <http://doi.acm.org/10.1145/1150402.1150436>
- [25] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Proc. of the 28th Intl. Conf. on Neural Information Processing Systems*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1225–1233.
- [26] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 671–687, Jun. 2003. [Online]. Available: [http://dx.doi.org/10.1016/S0022-0000\(03\)00025-4](http://dx.doi.org/10.1016/S0022-0000(03)00025-4)
- [27] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proc. of the Intl. Conf. on Very Large Data Bases*, ser. VLDB '07, 2007, pp. 950–961.
- [28] A. Broder, "On the resemblance and containment of documents," in *Proc. of the Compression and Complexity of Sequences*, ser. SEQUENCES '97. IEEE Computer Society, 1997, pp. 21–.
- [29] A. Shrivastava and P. Li, "In defense of Minhash over Simhash," in *Proc. of the Intl. Conf. on Artificial Intelligence and Statistics*, ser. AISTATS '14, 2014, pp. 886–894.
- [30] B. Wang, Z. Li, M. Li, and W. y. Ma, "Large-scale duplicate detection for web image search," in *IEEE Intl. Conf. on Multimedia and Expo*, July 2006, pp. 353–356.
- [31] X.-J. Wang, L. Zhang, F. Jing, and W.-Y. Ma, "Annosearch: Image auto-annotation by search," in *IEEE Computer Society Conf. on Computer Vision and Pattern Recognition*, ser. CVPR '06, vol. 2, 2006, pp. 1483–1490.
- [32] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Proc. of the Intl. Conf. on Neural Information Processing*, ser. NIPS '09, 2008, pp. 1753–1760.
- [33] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.
- [34] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, "Labelme: A database and web-based tool for image annotation," *Int. J. Comput. Vision*, vol. 77, no. 1-3, pp. 157–173, May 2008.
- [35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.
- [36] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, Jan. 2011.
- [37] Z. Jin, C. Li, Y. Lin, and D. Cai, "Density sensitive hashing," *IEEE transactions on cybernetics*, vol. 44, no. 8, pp. 1362–1371, 2014.
- [38] A. N. Gorban and I. Y. Tyukin, "Blessing of dimensionality: mathematical foundations of the statistical physics of data," *CoRR*, vol. arXiv:1801.03421, 2018.
- [39] Y. Delalleau, O. Bengio, "Shallow vs. deep sum-product networks," in *Proc. of the 24th Intl. Conf. on Neural Information Processing Systems*, ser. NIPS '11, 2011, pp. 666–674.
- [40] D. Chen, X. Cao, F. Wen, and J. Sun, "Blessing of dimensionality: High-dimensional feature and its efficient compression for face verification," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '13, June 2013, pp. 3025–3032.
- [41] T. Hige, Y. Aso, M. N. Modi, G. M. Rubin, and G. C. Turner, "Heterosynaptic plasticity underlies aversive olfactory learning in *Drosophila*," *Neuron*, vol. 88, no. 5, pp. 985–998, Dec 2015.
- [42] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen, "A survey on learning to hash," *CoRR*, vol. arXiv:1606.00185, 2016.

Saket Navlakha is an assistant professor in the Integrative Biology Laboratory at the Salk Institute for Biological Studies. He received an A.A. from Simon's Rock College in 2002, a B.S. from Cornell University in 2005, and a Ph.D. in computer science from the University of Maryland College Park in 2010. He was then a post-doc in the Machine Learning Department at Carnegie Mellon University until 2014. His research interests include designing algorithms to study the structure and function of biological networks, and the study of "algorithms in nature".

Jaiyam Sharma received his Bachelor of Technology in Engineering Physics from Indian Institute of Technology Delhi, India. He received a Master of Engineering in Electrical and Electronic Engineering from Toyohashi University of Technology, Japan. His research interests are computer vision algorithms for medical diagnostics. He is currently a doctoral candidate at The University of Electro-Communications, Tokyo and a collaborator with Prof. Saket Navlakha at the Salk Institute.

Improving Similarity Search with High-dimensional Locality-sensitive Hashing

Supplementary Information

Jaiyam Sharma, Saket Navlakha

7 EMPIRICAL ANALYSIS OF LEMMA 1

To support our theoretical analysis of Lemma 1 (main text), we performed an empirical analysis showing that the AUPRC for DenseFly is very similar to that of SimHash when using equal dimensions (Figure 5). DenseFly, however, takes k -times less computation. In other words, we proved that the computational complexity of SimHash could be reduced k -fold while still achieving the same performance.

8 COMPARING PSEUDO-HASHES WITH SIMHASH (LEMMA 4)

Lemma 4 proved that pseudo-hashes approximate SimHash with increasing WTA factor, k . Empirically, we found that the performance of only using pseudo-hashes (not using the high-dimensional hashes) for ranking nearest-neighbors performs similarly with SimHash for values of k as low as $k = 4$ (Figure 6 and Figure 7), confirming our theoretical results.

9 BOUNDS OF $|a'_j - a_j|$

Here we derive a result used in the proof of Lemma 2 (main text). Let $[a, b]$ denote the set of all integers from a to b .

Consider $|a'_j - a_j|$ as defined in the proof of Lemma 2. Since $a'_j = \sum_{i \in S_j} x'_i = a_j + \sum_{i \in S_j} \delta x_i$, then $|a'_j - a_j| = |\sum_{i \in S_j} \delta x_i|$. The problem of finding the maximum value of $|a'_j - a_j|$ is one of constrained optimization and can be solved, generally speaking, by using the Karush-Kuhn-Tucker conditions. In this case the solution can also be found using Lagrange multipliers as we show below.

Let $h(\{\delta x_i | i \in S_j\}) \equiv |\sum_{i \in S_j} \delta x_i|$. The problem is to maximize $h(\{\delta x_i | i \in S_j\})$ such that $\sum_{t=1}^{mk} |\delta x_t|^p = \epsilon^p$.

This translates to an equivalent condition $\sum_{i \in S_j} |\delta x_i|^p \leq \epsilon^p$. Since $|S_j| = \lfloor d\alpha \rfloor$, we reformulate h without loss of generality as $h(\delta x_1, \delta x_2, \dots, \delta x_{d\alpha}) = |\sum_{i=1}^{d\alpha} \delta x_i|$, where we drop $\lfloor \cdot \rfloor$ notation for simplicity. Also, we note that $|\sum_{i=1}^{d\alpha} \delta x_i| \leq \sum_{i=1}^{d\alpha} |\delta x_i|$, where the equality holds if and only if $\delta x_i \geq 0 \forall i \in [1, d\alpha]$. Thus, the absolute value signs can be dropped (if the solution found by dropping the absolute value is indeed ≥ 0).

Next, let $f(\delta x_1, \dots, \delta x_{d\alpha}) \equiv \sum_{i=1}^{d\alpha} \delta x_i$, subject to the constraint $g(\delta x_1, \dots, \delta x_{d\alpha}) \leq 0$ where $g(\delta x_1, \dots, \delta x_{d\alpha}) = \sum_{i=1}^{d\alpha} \delta x_i^p - \epsilon^p$. We note that the global maximum of f lies outside the ϵ -ball defined by g . Thus, the constraint g is active at the optimal solution so that $g(\delta x_1, \dots, \delta x_{d\alpha}) = 0$.

Thus, the optimal solution is calculated using the Lagrangian:

$$\begin{aligned} \mathcal{L}(\delta x_1, \dots, \delta x_{d\alpha}, \lambda) &= \sum_{i=1}^{d\alpha} \delta x_i - \lambda \left(\sum_{i=1}^{d\alpha} \delta x_i^p - \epsilon^p \right). \\ \frac{\partial \mathcal{L}}{\partial \delta x_i} &= 1 - p\lambda \delta x_i^{p-1} \quad \forall i \in [1, d\alpha] \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= \sum_{i=1}^{d\alpha} \delta x_i^p - \epsilon^p \end{aligned}$$

Setting $\frac{\partial \mathcal{L}}{\partial \delta x_i} = 0$, we get $\delta x_i = (\frac{1}{p\lambda})^{1/(p-1)} \equiv \gamma \forall i \in [1, d\alpha]$. Setting $\frac{\partial \mathcal{L}}{\partial \lambda} = 0$, we get $d\alpha \gamma^p = \epsilon^p$.

Thus, $\gamma = \epsilon / \sqrt[p]{d\alpha}$ is the only admissible solution for any p since $\delta x_i \geq 0 \forall i \in [1, d\alpha]$ and $\gamma > 0$. Therefore, $f(\delta x_1, \dots, \delta x_{d\alpha}) \leq d\alpha \epsilon / \sqrt[p]{d\alpha}$ and the proof follows.

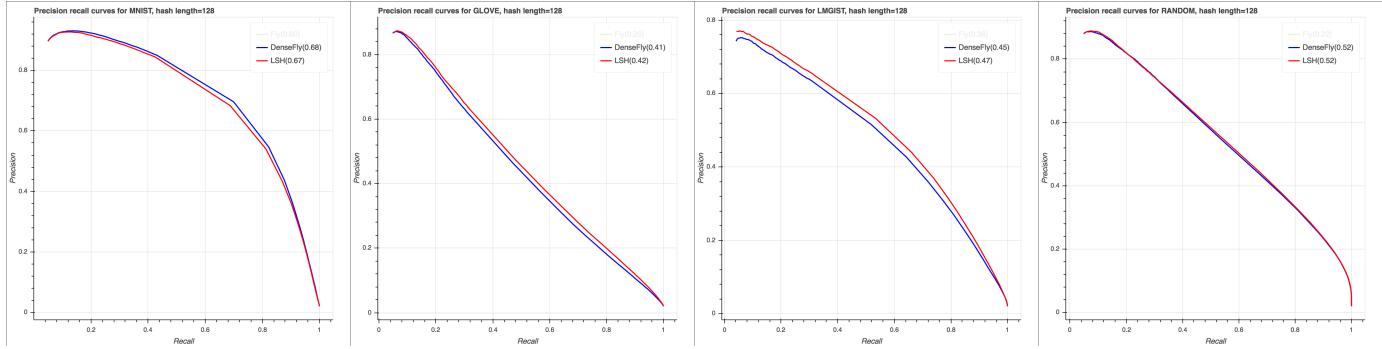


Fig. 5. Empirical evaluation of DenseFly with SimHash when using the same hash dimension. These empirical results support Lemma 1. Results are shown for $k = 20$.

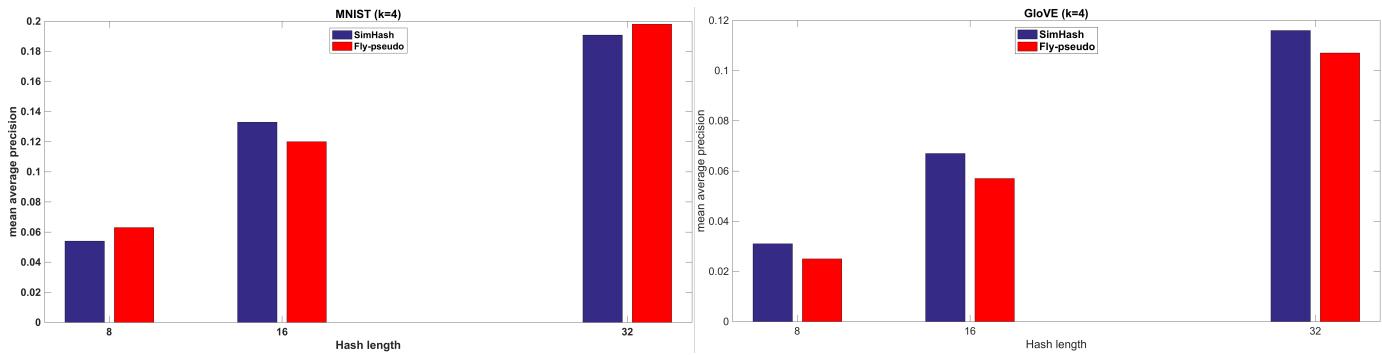


Fig. 6. Mean average precision of pseudo-hashes for $k = 4$ on the MNIST and GLoVE datasets. The mAP scores were calculated over 500 queries. DenseFly pseudo-hashes and SimHash perform similarly.

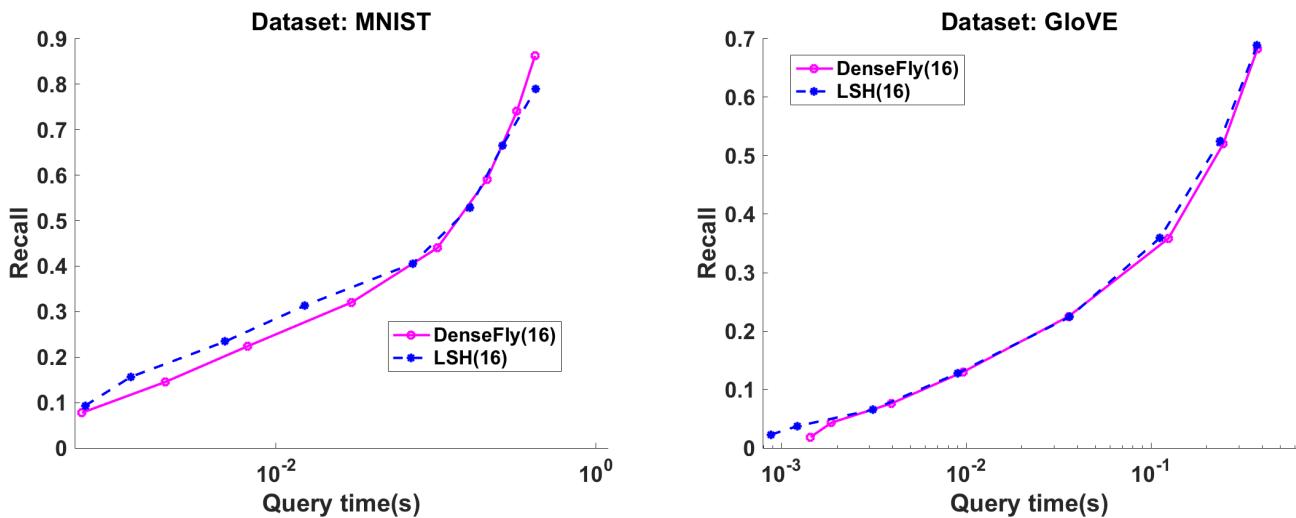


Fig. 7. Recall versus query time for MNIST and GLoVE by SimHash (LSH) and DenseFly. Ranking of candidates for DenseFly is done by pseudo-hashes. The recall for pseudo-hashes is nearly the same as SimHash across all search radii (query times). This supports the theoretical argument that pseudo-hashes approximate SimHash.