Fast Reciprocal Nearest Neighbors Clustering

Roberto J. López-Sastre *, Daniel Oñoro-Rubio, Pedro Gil-Jiménez, Saturnino Maldonado-Bascón

University of Alcalá, GRAM - Department of Signal Theory and Communications, 28805 Alcalá de Henares, Spain

Abstract

This paper presents a novel approach for accelerating the popular Reciprocal Nearest Neighbors (RNN) clustering algorithm, *i.e.* the fast-RNN. We speed up the nearest neighbor chains construction via a novel dynamic slicing strategy for the projection search paradigm. We detail an efficient implementation of the clustering algorithm along with a novel data structure, and present extensive experimental results that illustrate the excellent performance of fast-RNN in low- and high-dimensional spaces. A C++ implementation has been made publicly available.

Keywords: reciprocal nearest neighbors, clustering, visual words, local descriptors

1. Introduction

Many image processing and computer vision tasks can be posed as one of Nearest Neighbors (NN) search. Indeed, the vector quantization techniques, which are used in low-bit-rate image compression techniques (*e.g.* [1, 2]), are directly related to searching NN. Moreover, Bag-of-Words approaches for object recognition [3], need to quantize local visual descriptors, such as SIFT [4], so as to build the so called visual vocabulary. In other words, there is a pressing need for developing fast clustering algorithms that work well with both low- and high-dimensional data. Within the object recognition context, K-means is the most widely used clustering algorithm, even though more efficient alternatives have been proposed (*e.g.* [5]). There are also agglomerative techniques (*e.g.* [6]) that overcome the K-means limitations. In [6], the Reciprocal Nearest Neighbors (RNN) clustering algorithm [7] is used for local descriptors quantization. In this letter, we present an accelerated version of this clustering algorithm: the fast-RNN. In specific terms, we propose a novel method to accelerate the RNN clustering algorithm via a dynamic slicing strategy for the projection search paradigm. The use of this efficient dynamic space partitioning, combined with a novel data structure, improves the performance with both low- and high-dimensional data.

2. Fast Reciprocal Nearest Neighbors Clustering

2.1. Reciprocal Nearest Neighbors: An Overview

The RNN algorithm was introduced in [7]. It is based on the construction of RNN pairs of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, so that $\mathbf{x}_i$ is the NN to $\mathbf{x}_j$, and vice versa. As soon as a RNN pair is found, it can be agglomerated. In [8], it is described an efficient implementation that ensures that RNN can be found with as little re-computation as possible: this is achieved by building a NN chain, which consists of an arbitrary vector, followed by its NN, which is again followed by its NN among the remaining points and so on. Hence, a NN chain of length l can be defined as the sequence of vectors $\{\mathbf{x}_1, \mathbf{x}_2 = \mathrm{NN}(\mathbf{x}_1), \ldots, \mathbf{x}_{l-1} = \mathrm{NN}(\mathbf{x}_l), \mathbf{x}_l = \mathrm{NN}(\mathbf{x}_{l-1})\}$, where $\mathrm{NN}(\mathbf{x}_i)$ is the NN of $\mathbf{x}_i$. Note that the distances between adjacent vectors in the NN chain are monotonically decreasing, and that the last pair of nodes are RNN.

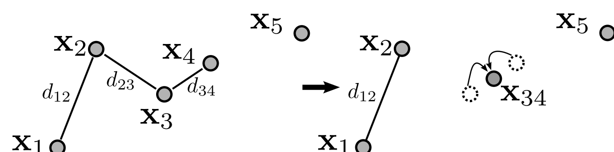


Figure 1: The RNN algorithm is run on this set of vectors. The NN chain starts with $\mathbf{x}_1$, and contains $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$. Note that $d_{12} > d_{23} > d_{34}$. $\mathbf{x}_5$ is not added to the NN chain because the distance $d_{45} > d_{34}$. The last pair of vectors, $\mathbf{x}_3$ and $\mathbf{x}_4$, are RNN. If $d_{34} \leq t$, vectors $\mathbf{x}_3$ and $\mathbf{x}_4$, are agglomerated in the new cluster $\mathbf{x}_{34}$. However, if $d_{34} > t$, then the whole chain is discarded, and each of its elements is considered to be a separate cluster.

The RNN algorithm starts with an arbitrary vector (see Figure 1 for a toy example). A NN chain is then built. When a RNN pair is found, *i.e.* no more vectors can be added to the current chain, the corresponding clusters are merged if their similarity is above a fixed cut-off threshold t, otherwise the algorithm discards the whole chain. This way of merging clusters can be applied whenever the distance matrix D satisfies the reducibility property, $D(C_i, C_j) \leq \min\{D(C_i, C_k), D(C_j, C_k)\} \leq D(C_{i\cup j}, C_k)$, where $D(C_i, C_j)$ is the distance between clusters C_i and C_j, and $C_{i\cup j}$ is the cluster after merging C_i and C_j. This property guarantees that when RNN are merged, the NN relations for the remaining chain members are unaltered, therefore they can be used for the next iteration. When the current chain is empty or has been discarded, a new arbitrary point is selected, and a new NN chain is started.

*Corresponding author: Tel: +34 91 885 67 20. Fax:+34 91 885 66 99
Email address: robertoj.lopez@uah.es (Roberto J. López-Sastre)

The key point is how to recompute the similarity between a new centroid (after merging a RNN pair) and the rest. Leibe *et al.* [6] show that this can be done efficiently if the cluster similarity can be expressed in terms of centroids, which holds for a group average criteria based on correlation or Euclidean distances. The similarities can be computed in constant time, and only the mean and variance of each cluster need to be stored. Moreover, both parameters can be computed incrementally. Let μ_x, μ_y and σ_x^2, σ_y^2 be the means and variances of clusters C_x and C_y respectively. The similarity between clusters C_x and C_y can be computed as similarity$(C_x, C_y) = -((\sigma_x^2 + \sigma_y^2) + (\mu_x - \mu_y)^2)$.

We adopt the RNN clustering algorithm introduced in [6], which has $O(N^2 d)$ time and $O(N)$ space complexity, where N is the number of data points of dimensionality d. Algorithm (1) describes the implementation of the RNN clustering in [6]. The approach in [6] presents a high complexity with high-dimensional data. This is to be expected since the algorithm relies heavily on the search for NN. In Section 2.2, we present an efficient technique for speeding up the NN chain construction in order to further improve the run-time of the clustering algorithm.

Algorithm 1 RNN clustering

```
C = ∅; last ← 0; lastsim[0] ← 0; //C contains a list of clusters
L[last] ← v ∈ V; //Start chain L with a random vector v
R ← V\v; //All remaining points are kept in R
while R ≠ ∅ do
  (s, sim) ← getNearestNeighbor(L[last], R);
  if sim > lastsim[last] then
    //No RNN. Add s to L
    last ← last + 1; L[last] ← s; R ← R\{s}; lastsim[last] ← sim;
  else {//A RNN pair was found}
    if lastsim[last] > thres then
      s ← agglomerate(L[last], L[last − 1]); R ← R ∪ {s};
      last ← last − 2;
    else {//Discard the current chain}
      C ← C ∪ L; last ← −1; L = ∅;
    end if
  end if
  if last < 0 then
    last ← last + 1; L[last] ← v ∈ R; R ← R\v
  end if
end while
```

2.2. fast-RNN

In the RNN clustering, the set of vectors to quantize is continuously changing: vectors are aggregated and/or extracted in each iteration. This makes it unfeasible to apply those NN search algorithms that are designed to work with non-dynamic set of vectors (*e.g.* [9, 1]). Moreover, the experiments in [9] only deal with datasets with dimensionality up to 35.

In order to further accelerate the RNN clustering, we present an efficient technique to speed up the NN chain construction. Since we have to deal with dynamic sets, we propose a novel algorithm for NN search which consists in a new efficient dynamic space partitioning strategy using slices, combined with a novel data structure to accelerate the NN chain construction.

Building NN chains via *slicing*. When building NN chains, the objective is to find the point, in the set of points S, that is closest to a query point $\mathbf{q} \in \mathbb{R}^d$ and within a distance ϵ. Instead of building a hypercube with side 2ϵ [9], we propose finding all the points that lie within a slice of the d-dimensional space of width 2ϵ centred at point $\mathbf{q} = (q_1, q_2, \ldots, q_d)^{\mathrm{T}}$. That is, the i-slice is defined as the region confined by two parallel planes, perpendicular to the ith coordinate axis, separated a distance 2ϵ and centred at q_i. For an ith coordinate, the higher its variance, the more suitable for being used.

The NN search is done with just the points inside the i-slice. We are interested in building NN chains. Suppose there is a set of N points $S = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ where $\mathbf{x}_i \in \mathbb{R}^d$. We assume that a metric $d(\mathbf{x}_i, \mathbf{x}_j)$ is defined between points in S. Any NN chain starts with a random point $\mathbf{x}_i$. Our first task is to determine the nearest neighbor of $\mathbf{x}_i$ in S, *i.e.* $\mathbf{x}_j = \mathrm{NN}(\mathbf{x}_i)$. To that end, we build the first slice of width 2ϵ centered at $\mathbf{x}_i$. All the points inside this slice are included on a candidate list. We perform the search for the NN of $\mathbf{x}_i$ considering only the points in the candidate list. Once $\mathbf{x}_j$ is identified, we search for its NN, *i.e.* $\mathbf{x}_k = \mathrm{NN}(\mathbf{x}_j)$, via slicing again, and so on.

As the distances between adjacent elements in a NN chain are decreasing, we can assign the value of the last distance between NN in the NN chain to ϵ. If there are no points within the slice for that ϵ, we can stop building the NN chain. If we proceed in this way, the longer the NN chain, the thinner the slices, therefore the faster the NN search.

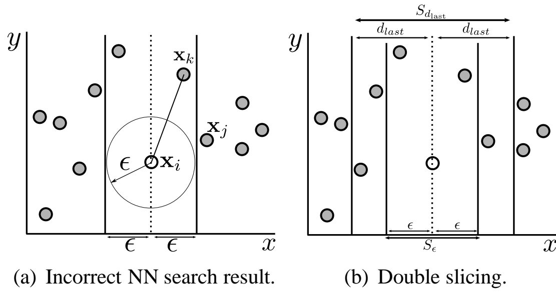
This procedure for updating ϵ is adequate when working with low-dimensional vectors. However, in a high-dimensional space the norm used to define the distance is concentrated [10]. Let $D_{\max_d}$ and $D_{\min_d}$ be the maximum and the minimum distance to the origin of a data point of dimensionality d, respectively. Then,

$$lim_{d\to\infty} \frac{D_{\max_d} - D_{\min_d}}{D_{\min_d}} \to 0\,. \quad (1)$$

This means that the minimum and maximum distances from a query point to points in the dataset become increasingly closer as dimensionality increases. As a result, if we update ϵ with the last distance in the NN chain, we will not trim the number of vectors we have for comparison. Hence for high-dimensional spaces a further study on how to determine ϵ is needed.

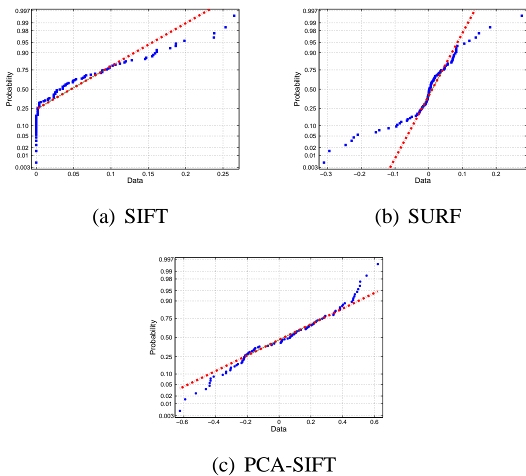
It is important to note that an exhaustive search within the slice does not always find the NN of x_i in S. When performing the linear search with the points inside the slice, we must therefore check whether the distance to the NN, *i.e.* $d(\mathbf{x}_i, \mathbf{x}_j)$, satisfies this condition $d(\mathbf{x}_i, \mathbf{x}_j) \leq \epsilon$. If not, we can not guarantee that $\mathbf{x}_j = \mathrm{NN}(\mathbf{x}_i)$ (Figure 2(a) illustrates this problem with an example).

In such a case, when the NN is not found within the slice, bigger slices must be generated until $\epsilon > d_{\mathrm{last}}$ (with d_{last} being the distance between the last two elements in the NN chain) or



(a) Incorrect NN search result. (b) Double slicing.

Figure 2: (a) Slicing a 2-dimensional space. $\mathbf{x}_j$ is closer to $\mathbf{x}_i$ than $\mathbf{x}_k$, but it lies outside the slice. (b) Slicing a 2-dimensional space. We build two slices: S_ϵ of width 2ϵ, and $S_{d_{\text{last}}}$ of width $2d_{\text{last}}$, where d_{last} is the distance between the last two elements in the NN chain.



(a) SIFT (b) SURF

(c) PCA-SIFT

Figure 3: Normal probability plots for a random coordinate of a random selection of (a) SIFT, (b) SURF and (c) PCA-SIFT descriptors.

until we find a NN. To avoid this iterative process, we propose building only two slices, see Figure 2(b). The first slice, S_ϵ, is built using an adequate ϵ that guarantees a significant trim in the number of points. The second slice, $S_{d_{\text{last}}}$, has a width of $2d_{\text{last}}$. If the NN is not found in S_ϵ, we search in $S_{d_{\text{last}}}$. Note that by doing this double search, the clusterings obtained by the fast-RNN and the RNN algorithms are identical.

Determining ϵ when slicing high-dimensional data. The number of points in a slice directly depends on the value of ϵ, so the efficiency of the proposed algorithm critically depends on ϵ too. How to choose ϵ? We focus our study on the specific case in which the set of vectors along each dimension is normally distributed. This assumption can be made if we use SIFT [4], SURF [11] or PCA-SIFT [12] descriptors. The normal probability plots in Figure 3 look fairly straight, at least when the large and small values are ignored.

Our aim is to analytically compute the width of the thinnest slice ($2\epsilon_{\text{min}}$), given that we want to guarantee that the slice is not empty with a probability p. Let N_s be the number of points within a slice of width $2\epsilon_{\text{min}}$. In order to determine the average number of points that lie in the slice, we compute $E[N_s]$. We

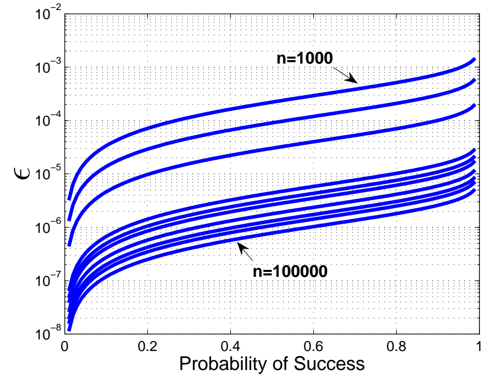


Figure 4: ϵ vs. Probability of success. These results are obtained using a set of SIFT descriptors extracted from random images of the database ICARO ([13]). We fix the number of descriptors n to different values from 1000 to 10^5.

can define Z_i as the distance between $\mathbf{q}_i$ and any point in the slice. P_c is the probability that any point in the set of points is within distance ϵ from $\mathbf{q}_i$, *i.e.* $P_c = P\{-\epsilon \leq Z_i \leq \epsilon | \mathbf{q}_i\}$. Regardless of the distribution of points, N_s is binomially distributed,

$$P\{N_s = k|\mathbf{q}_i\} = P_c^k (1-P_c)^{n-k} \binom{n}{k}. \tag{2}$$

We focus on the scenario where the set of points is normally distributed,

$$f_{Z_i|\mathbf{q}_i}(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(z-q_i)^2}{2\sigma^2}\right), \tag{3}$$

and P_c can then be written as

$$P_c = \int_{-\epsilon}^{\epsilon} f_{Z_i|\mathbf{q}_i}(z)dz = \frac{1}{2}\left(\text{erf}\left(\frac{\epsilon - \mathbf{q}_i}{\sigma\sqrt{2}}\right) + \text{erf}\left(\frac{\epsilon + \mathbf{q}_i}{\sigma\sqrt{2}}\right)\right). \tag{4}$$

The probability p that the slice contains at least one point is

$$\begin{aligned} p &= P\{N_s > 0|\mathbf{q}_i\} = 1 - P\{N_s = 0|\mathbf{q}_i\} \\ &= 1 - (1-P_c)^n \\ &= 1 - \left(1 - \frac{1}{2}\left(\text{erf}\left(\frac{\epsilon - \mathbf{q}_i}{\sigma\sqrt{2}}\right) + \text{erf}\left(\frac{\epsilon + \mathbf{q}_i}{\sigma\sqrt{2}}\right)\right)\right)^n \end{aligned} \tag{5}$$

Using Equation (5), ϵ is plotted against p in Figure 4. For this purpose we obtained a set of 10^5 SIFT vectors extracted from random images from the database ICARO [13]. We set the number of descriptors n at different values from 1000 to 10^5. Note that the value of ϵ required for building non-empty slices is very low for probabilities of success near 0.9, *e.g.* an $\epsilon = 0.012$ guarantees a probability of success of 0.9 when $n = 1000$. In practice, ϵ is fixed to larger values, but always keeping the number of points within the slice small, as we will see in the experiments.

Data Structure. Our implementation of the fast-RNN algorithm uses an effective dynamic data structure and 1D binary searches to efficiently find points inside the region defined by two parallel planes. First, we assume that the set of points we

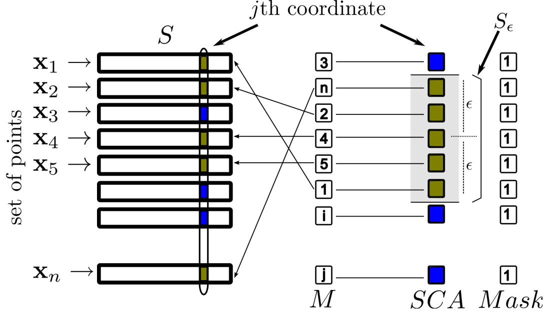


Figure 5: Data structures of fast-RNN. A slice S_ϵ is built using the SCA. Note that only 2 binary searches are needed.

are dealing with is dynamic. For this reason, the data structure needs to be updated whenever the set changes.

Considering d the dimensionality of data, only coordinate j $(0 < j < d)$ is stored as a 1D array. This is called the Sorted Coordinate Array (SCA). In other words, we construct only one slice, which is perpendicular to the jth coordinate axis. Let us assume that our objective is to find the NN, in a set S of n points, of a given point $\mathbf{x}_i$, with coordinates $\mathbf{x}_i = (x_{i_1}, \ldots, x_{i_j}, \ldots, x_{i_d})^{\mathrm{T}}$. In order to construct the candidate list efficiently, we must search for those points that lie between two parallel planes perpendicular to the jth coordinate axis, centered at x_{i_j} and separated by a distance 2ϵ; *i.e.* our aim is to identify the points with jth coordinates in the SCA within the limits $x_{i_j} - \epsilon$ and $x_{i_j} + \epsilon$. The SCA is sorted in order to build the candidate list of points with just two binary searches (each binary search has a complexity of $O(\log n)$ in the worst case).

Figure 5 shows the data structure introduced. In our C++ implementation, the set of points S is built as a *list* of *vectors*, where efficient insertions and deletions of elements can take place anywhere in the *list*. In order to map a coordinate in the SCA to its corresponding point in the set of points S, we maintain an array M of *iterators*, where each element points to its corresponding vector in the *list* S. We maintain a 1D array $Mask$ of boolean elements to deal with the insertions and deletions of points. Every element acts as a mask, indicating whether its position has been deleted (*false*) or not (*true*). When deleting an element, we first mark its mask to *false*, and then we delete the element from the list S. When a new vector has to be inserted, we first insert it in the list S, then with a binary search we determine the corresponding position of its jth coordinate in the SCA, and finally, we update SCA and $Mask$.

Note that SCA does not grow when we insert elements in S. Each insertion is associated with an agglomeration of two vectors that have been extracted beforehand. When an element is inserted, there are at least two elements in the SCA marked as deleted, and the algorithm simply update SCA by moving the elements within the array.

The data structure described is easy and fast to build and maintain. The time complexity for its building is on average $O(n \log n)$, with n being the number of vectors. When elements are deleted or inserted, the data structure is updated at a low cost. Furthermore, the size of the data structure does not grow with dimensionality, which is a desirable property.

With the modified approach for accelerating the NN chain construction, and the efficient data structured detailed, we are able to speed up the RNN clustering. Algorithm 2 summarizes the resulting procedure.

Algorithm 2 fast-RNN

```
(M, SCA, Mask) ← initializeDataStructure(V);
C = ∅; last ← 0; lastsim[0] ← 0; //C contains a list of clusters
L[last] ← v ∈ V; //Start chain L with a random vector v
R ← V\v; //All remaining points are kept in R
while R ≠ ∅ do
  (S_ε, S_d_last) ← createSlices(R, ε, L, lastsim, M, SCA, Mask);
  (s, sim) ← getNearestNeighborInSlices(L[last], R, S_ε, S_d_last);
  if sim > lastsim[last] then
    last ← last + 1; L[last] ← s; R ← R\{s};
    lastsim[last] ← sim;
    (M, SCA, Mask) ← eraseElement(s, M, SCA, Mask);
  else {//A RNN pair was found}
    if lastsim[last] > thres then
      s ← agglomerate(L[last], L[last − 1]); R ← R ∪ {s};
      last ← last − 2;
      (M, SCA, Mask) ←
      insertElement(s, M, SCA, Mask);
    else {//Discard the current chain}
      C ← C ∪ L; last ← −1; L = ∅;
    end if
  end if
  if last < 0 then
    last ← last + 1; L[last] ← v ∈ R; R ← R\v
  end if
end while
```

3. Experimental Evaluation

In this section we present an experimental comparison between the RNN and the fast-RNN clustering algorithms. We have used two groups of datasets. On the one hand, 10^4 SIFT [4], PCA-SIFT [12] and SURF [11] descriptors have been extracted from random images in the dataset ICARO [13]. While SIFT descriptors have 128 dimensions, we have extracted PCA-SIFT vectors of dimensionality 36, and SURF descriptor of 128, 64, 36 and 16 dimensions. On the other hand, we have generated 1 set of 50,000 3D vectors from the normal distribution. With these sets of vectors we can show how the fast-RNN performs in both low- and high-dimensional spaces.

We measure the performance of an algorithm A as the number of distance calculations dc_A required. We define the speedup $S = dc_{\mathrm{RNN}}/dc_{\mathrm{fast-RNN}}$. However, fast-RNN may incur overhead to build and update its data structure. Therefore, we also define the time speedup $S_t = t_{\mathrm{RNN}}/t_{\mathrm{fast-RNN}}$, where t_A is the time required by algorithm A. For the experiments, we repeat the measurements 10 times.

Measuring the speedup. Table 1 shows the results obtained as a function of n, the number of vector to quantize. Results

Table 1: Fast-RNN vs. RNN. S and S_t for different datasets.

	Clustering		S			S_t		
Dataset	t	ϵ	$n = 100$	$n = 1,000$	$n = 10,000$	$n = 100$	$n = 1,000$	$n = 10,000$
SIFT	0.9	0.05	1.46	1.61	1.67	1.61	1.77	1.71
SURF-16	0.6	0.05	2.71	3.98	4.12	2	3.67	3.91
SURF-36	0.6	0.05	2.64	2.68	3.03	2.23	2.66	2.96
SURF-64	0.6	0.05	1.83	1.99	2.09	1.74	2	2.02
SURF-128	0.6	0.05	1.58	1.71	1.85	1.54	1.72	1.86
PCA-SIFT	0.6	0.1	2.34	3.13	4.24	2.20	3.78	4.71
			$n = 100$	$n = 1,000$	$n = 50,000$	$n = 100$	$n = 1,000$	$n = 50,000$
NORM-3	0.4	0.1	2.96	3.15	4.15	1.95	3.41	5.91

show that the number of distance calculations always decreases when using the fast-RNN algorithm, *i.e.* S is always > 1. Furthermore, observing S_t, we can conclude that the fast-RNN approach is always faster than the RNN. It is also true that the speedup S increases with n. When the dataset is small, *e.g.* $n = 100$, the efficiency of the NN search algorithm via slicing is comparable to a simple linear search. This is due to the fact that the fast-RNN requires to create and update an auxiliary data structure, therefore the speedup over a linear search decreases when the dataset size drops. Furthermore, when SIFT or SURF-128 descriptors are used, the speedup does not exceed 2, and this is due to the following factors: the value of ϵ chosen and the high-dimensionality of the vectors. SIFT and SURF-128 vectors are concentrated (recall Equation (1)), hence a lower value for ϵ is needed to considerably reduce the number of distance calculations, as it is shown in the following section. Table 1 also shows the excellent performance of the fast-RNN clustering when quantizing low-dimensional vectors (see results for the NORM-3 dataset).

Determining the best ϵ. The speedup of the proposed algorithm depends critically on ϵ, specially for high-dimensional data (*e.g.* SIFT and SURF-128 descriptors). In the fast-RNN algorithm, the computational efficiency is achieved by limiting the search to a small slice of the d-dimensional space. However, in high-dimensional spaces the distances are concentrated. This concentration is problematic when building the slice of size 2ϵ: an inadequate ϵ can include almost all the points inside the slice, thereby not reducing the NN search time. In Figure 6, we show S for PCA-SIFT, SIFT and SURF descriptors varying ϵ from 0.001 to 0.1. For PCA-SIFT descriptors, S does not depend on ϵ within the interval $[0.001, 0.1]$, *i.e.* the PCA-SIFT coordinates are not concentrated within this interval. However, for SIFT and SURF-128 descriptors, we obtain that $S > 2$ when $\epsilon < 0.02$ and $\epsilon < 0.04$, respectively. SIFT and SURF descriptors, of dimensionality 128, require lower values of ϵ in order to accelerate the NN chain construction within the fast-RNN clustering.

4. Conclusion

This paper details the implementation of the fast-RNN clustering algorithm. To the best of our knowledge, this is the first approach for accelerating the RNN clustering algorithm via the efficient dynamic space partitioning presented. We

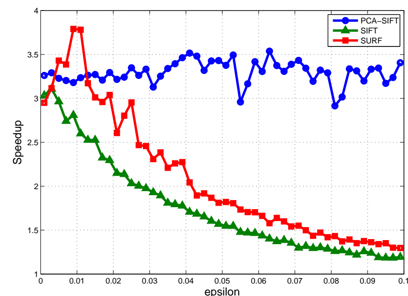


Figure 6: Speedup as a function of ϵ for PCA-SIFT, SIFT and SURF descriptors.

also have developed a novel data structure that improves the performance with both low- and high-dimensional data. Results show that the fast-RNN is faster than the standard RNN. It is worth to mention that the solutions obtained by both clustering algorithms are identical. Finally, with the aim of making our research reproducible, we release a C++ implementation of the fast-RNN clustering, which can be downloaded from `http://agamenon.tsc.uah.es/Personales/rlopez/data/fastrnn`.

Acknowledgements

This work was partially supported by projects TIN2010-20845-C03-03 and CCG10-UAH/TIC-5965.

References

[1] S. Baek, K. Bae, M. Sung, A fast vector quantization encoding algorithm using multiple projection axes, Signal Processing 75 (1999) 89–92.

[2] C.-H. Lee, L.-H. Chen, High-speed closest codeword search algorithms for vector quantization, Signal Processing 43 (1995) 323 – 331.

[3] G. Csurka, C. R. Dance, L. Fan, J. Willamowski, C. Bray, Visual categorization with bags of keypoints, in: ECCV, 2004.

[4] D. G. Lowe, Distinctive image features from scale-invariant keypoints, IJCV 60 (2) (2004) 91–110.

[5] D. Nister, H. Stewenius, Scalable recognition with a vocabulary tree, in: CVPR, 2006, pp. 2161–2168.

[6] B. Leibe, K. Mikolajczyk, B. Schiele, Efficient clustering and matching for object class recognition, in: BMVC, 2006.

[7] C. de Rham, La classification hiérarchique ascendante selon la méthode des voisins réciproques, Cahiers de l'Analyse des Données 2 (5) (1980) 135–144.

[8] J. Benzécri, Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques, Cahiers de l' Analyse des Données 2 (7) (1982) 209–218.

[9] S. A. Nene, S. K. Nayar, A simple algorithm for nearest neighbor search in high dimensions, IEEE TPAMI 19 (9) (1997) 989–1003.

[10] K. S. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft, When is "nearest neighbor" meaningful?, in: Proceedings of the 7th International Conference on Database Theory, 1999.

[11] H. Bay, T. Tuytelaars, L. Van Gool, Surf: Speeded up robust features, in: ECCV, Vol. 3951, 2006, pp. 404–417.

[12] Y. Ke, R. Sukthankar, PCA-SIFT: A more distinctive representation for local image descriptors, in: CVPR, 2004.

[13] R. J. López-Sastre, C. Redondo-Cabrera, P. Gil-Jiménez, S. Maldonado-Bascón, ICARO: Image Collection of Annotated Real-world Objects, `http://agamenon.tsc.uah.es/Personales/rlopez/data/icaro` (2010).