CSE 252C: Computer Vision III

Guest Lecturer: Lawrence Cayton
Scribes: Andrew Rabinovich and Vincent Rabaud
Edited by: Catherine Wah

LECTURE 4
Efficient NN Search: Dimensionality Reduction and Data Structures

4.1. Introduction

4.1.1. Why NN search?

Nearest Neighbor (NN) search is a core operation that is performed in many approaches to object recognition. Once images in the database are represented as vectors or histograms (by whatever means), one way to label the unknown data is by comparing the novel images to the database of training examples; if there is a image that is "close," then it is likely to have the sample label (*i.e.* be the same object).

Although NN is a very simple classification scheme, when coupled with a proper dissimilarity measure, it often outperforms many sophisticated approaches to classification (for instance, SVM or boosting). The main drawback of NN, however, is its complexity, in both time and space. Brute force NN search of a database requires a comparison of the query to all the training

[1]Department of Computer Science and Engineering, University of California, San Diego.

September 20, 2009

examples. In this lecture, we look at speeding up NN search by (i) reducing the dimension of the space the database exists in, and by (ii) using spatial data structures that allow one to calculate distances only to a portion of the database.

4.1.2. Notation and Problem Formulation

Consider some database of points, where each point represents an image, $X = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}^D$. For a novel query, $q \in \mathbb{R}^D$, we want to find a point $x_i \in X$ that is "nearest" or "most similar" to q.

The notion of "nearest" is dependent on the dissimilarity measure used to compare elements in the database. Here we'll focus on the L_2-norm, but the efficient approaches presented later on will also work with other L_p-norms. As a side note, the popular tree-based method for NN searching can be extended to arbitrary metric spaces (*i.e.* can be made to work for any dissimilarity notion as long as it obeys the metric constraints such as the triangle inequality, nonnegativity, symmetry, etc.).

Returning to our problem and using the L_2-norm, we wish to find the shortest distance between the query q and x_i: $argmin_{x_i \in X} ||q - x_i||_2$. The brute force search runs in $\mathrm{O}(nD)$ time, where n is the size of the database and D is the dimensionality. This naive approach is too slow, and often becomes useless for searches in very large databases (*e.g.* bioinformatics, computer vision, and web applications).

To speed up the naive NN search, two general approaches are possible. Since the complexity of NN-search depends on D and n, we'll try to either reduce the dimension of the data (smaller D), or decrease the number of candidate elements in the database (smaller n).

4.2. Dimensionality Reduction

In reducing the dimensionality of a particular dataset, the key objective is to ensure that the distances between all the data points remain unchanged; the absolute position of the point is not important.

As an excellent candidate for dimensionality reduction, consider the MNIST dataset of handwritten digits. While this dataset is fairly simple, it is still of very high dimension. Each image is 28×28 pixels, resulting in a image that is represented as a point in 784-dimensional space. However, most of the pixels in the image are black, resulting in a highly sparse vectorial representation of the image, which may potentially be compressed to reduce the dimensionality.

More formally, we want to find a mapping $f : \mathbb{R}^D \to \mathbb{R}^d$, where $d << D$, s.t. $||f(x) - f(y)|| \approx ||x - y||$, or the distance in the d-dimensional space is approximately equal to the distance in D-dimensional space. We will store only $\{f(x_1), \ldots, f(x_n)\}$ in the database; to compute q's NNs, we map q to $f(q)$ and compare $f(q)$ to $f(x_1), \ldots, f(x_n)$. Next, we show how to find such a mapping f.

4.2.1. Johnson-Lindenstrauss Theorem

The Johnson-Lindenstrauss theorem gives a formal justification for dimensionality reduction. Suppose that $X = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}^D$. Then $\exists f : \mathbb{R}^D \to \mathbb{R}^d$ such that distances are well-preserved:

$$(1-\epsilon)||x_i - x_j||_2 \leq ||f(x_i) - f(x_j)||_2 \leq (1+\epsilon)||x_i - x_j||_2, \quad \forall i, j, \tag{4.1}$$

where $d = O(\frac{\log n}{\epsilon^2})$. Note that subtracting and adding an arbitrary ϵ in Eq. (4.1) only contracts and expands the distances by a small amount. Also, observe that d, the reduced dimension of the dataset, is *completely independent* of the original dimension, D, of the dataset, and in fact is quite small.

So, how do we find this mapping f? As it turns out, we can pick it at random! In particular, $f(x) \overset{def}{=} Gx$, where $G \in \mathbb{R}^{d \times D}$, $G_{ij} \sim N(0, \frac{1}{d})$, will satisfy Eq. (4.1) with high probability. This transformation essentially projects the data onto a random d-dimensional subspace (a hyperplane). An illustration of such a projection is shown in Fig. 1. The matrix G will have

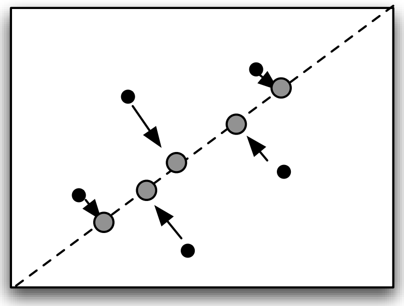

Figure 1. Projection of 2-D points onto a 1-D plane (dashed line).

close to full column rank with high probability if the vectors are chosen at random. Overall, this approach can randomly reduce data dimensionality significantly (to about $\log n$ dimensions) without too much loss of quality.

4.2.2. Other Dimensionality Reduction Methods

There are a number of other approaches to dimensionality reduction. Some assume that the data lies on a low dimensional hyperplane (*e.g.* a linear subspace for PCA), while others relax that assumption and aim to find

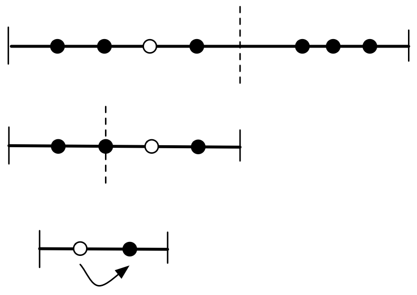

Figure 2. Motivation for kd-tree: 1-D binary search example. Search is accomplished in $O(\log n)$.

a low dimensional manifold (*e.g.* Isomap, MVU, LLE). These approaches are typically expensive compared to random projections, especially for large databases, but produce better results in some situations.

4.3. Data Structures

The basic idea in designing a data structure to be used in NN-searching is to carve up the space $\mathbb{R}^D$ into regions containing only a few points. To find a query q's NNs, we look only at points that fall in some region. Now we consider two of the most common data structures used for NN-search in object recognition.

4.3.1. kd-Trees

The motivation for using this approach is based on a binary search. Suppose we have $X = \{x_1, \ldots, x_n\} \in \mathbb{R}^1$: how would we find a query q's NN? In Fig. 2, we see that performing only $O(\log n)$ comparisons is sufficient to find NN, which is much cheaper than the brute force NN-search. It is very easy to perform this search in 1-D, since we can sort the points. kd-trees generalize binary search to $\mathbb{R}^D$, decomposing the D-dimensional space using median splits over multiple axes.

This procedure (Algorithm 4.1) is essentially a binary search tree approach. The value m refers to the actual median or split value, while i refers to the dimension to compare to. Fig. 3 is an example of 2-D partitioning, where $D = 2$ and MinSize $= 1$. Note that non-leaf nodes correspond to the splits, while the leaves are the regions containing the data points.

In searching this data structure for a NN to q, we take the following approach:

(a) Descend to the leaf node containing q by comparing to all the median lines.
(b) Find the NN to q in that cell.

Algorithm 4.1 BuildTree(S), where $S \subseteq \mathbb{R}^D$

```
if $|S| <$ MinSize then
    return leaf
else
    pick axis $i$ of maximum spread, $i \in \{1, \ldots, D\}$
    $m \leftarrow$ median $(\{s_i \mid s \in S\})$
    LeftTree $\leftarrow$ BuildTree($\{s \in S \mid s_i \leq m\}$)
    RightTree $\leftarrow$ BuildTree($\{s \in S \mid s_i > m\}$)
    return ($\{$LeftTree, RightTree, $m, i\}$)
end if
```

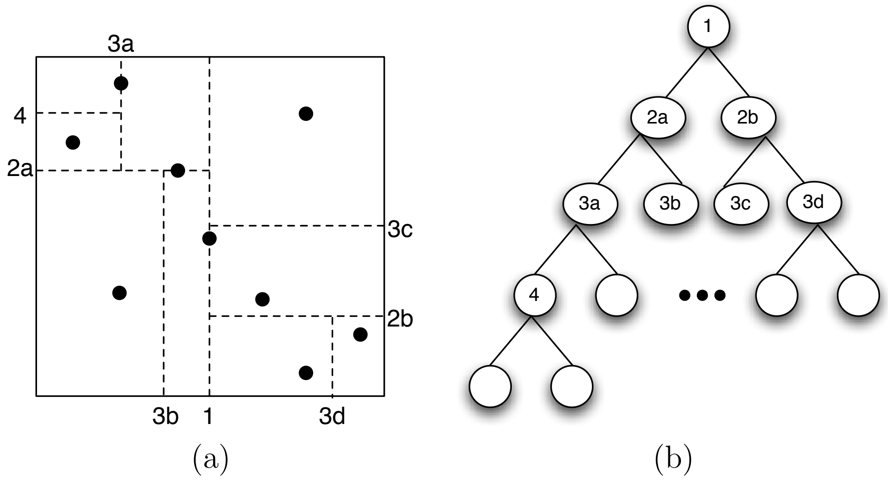

Figure 3. (a) kd-binary partitioning. (b) kd-tree. Non-leaf nodes correspond to the splits, while the leaves are the regions containing the data points.

(c) Backtrack up the tree: at each node, if the distance from q to the current NN is greater than the distance to the split, explore with the other subtree. This is illustrated in Fig. 4.

The complexity of this approach lies between $\log n$ and n distance calculations. Depending on how much backtracking is required, one may end up exploring the entire tree in the worst case, which would be the same as brute force search with bookkeeping. Empirically, excellent results have been reported for this approach for up to $D = 20$, and sometimes as high as $D = 100$. There are a number of variations of kd-trees, for instance, combining them with random projections.

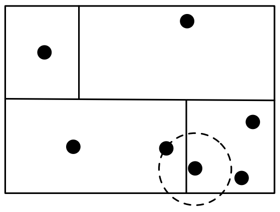

Figure 4. Example of kd partitioning that requires backtracking up the tree. The nearest neighbor of q (at the center of the dotted circle) is not in the same cell as itself, so backtracking is needed, mkaing this worse than binary search.

4.3.2. Locality Sensitive Hashing (LSH)

Based on the idea of approximate NN-search, LSH is the only algorithm that works provably well in arbitrary dimensions. With a user-specified radius R and error tolerance ϵ, LSH will return a point x s.t. $||q - x|| \leq R(1 + \epsilon)$. Note that this is only true under the assumption that there is a point in the database within distance $R(1 - \epsilon)$ of the query. Essentially, R is our best guess as to the distance to q's NN. R can be found exactly with $\mathrm{O}(\log \frac{n}{\epsilon})$ searches.

In constructing the data structure for LSH, all points are randomly projected down to about $d = \log n$ dimensions, and an equi-spaced grid is placed over $\mathbb{R}^d$ where the bin width is a free parameter – see Fig. 5 for an example. To search for q's NN, we calculate q's hash value and return the nearest x with the same hash value.

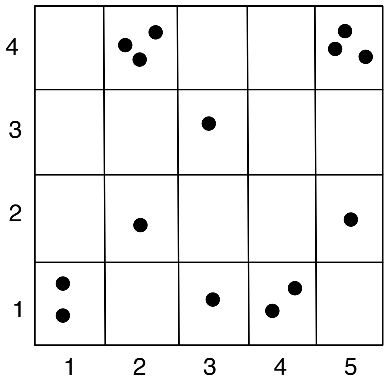

Figure 5. LSH. Uniform binning hashes together nearby points. Bin coordinates are the hash values used in searching. For example, the hash value of the bottom left points is (1,1).

In practice, if the dimensionality is decreased dramatically, too many points will fall in the same bin; another failure mode is when no points fall in the bin. To overcome these issues, we can increase the number of hash functions, or run the search multiple times and vote.

Theoretical results show that the complexity of LSH requires n^p distance calculations, where $p < 1$ (p depends on a variety of factors and is difficult to calculate exactly). This is the only data structure with guaranteed sublinear retrieval time that requires only polynomial space. In theory, for LSH to work, one must use a number of these data structures in parallel (all with different initial conditions), yet in practice, it is sometimes possible to successfully use only one.