Discriminative Learning and Big Data
Approximate Nearest Neighbours

Andrew Zisserman

Visual Geometry Group
University of Oxford
http://www.robots.ox.ac.uk/~vgg
Big Data means at least …

- Image search, e.g. Facebook
  - 1 billion images = 1 billion descriptors (e.g. 128D based on CNNs)

- Video search: thousands of hours of video
  - Billions of audio and video descriptors

- Music search: e.g. Shazam

- The order of magnitude considered in this lecture is millions to billions
Application: Large scale image/video retrieval

Objective: retrieve images from an image or video dataset that contain an object category

- Represent every image (or keyframe) by a feature vector
- Train an SVM using positive images that contain the category and negative images that don’t
- Score each image in the dataset by the classifier and rank
- Basic operation: nearest neighbour matching/retrieval
Image retrieval

Training
- Training Images
  - Positive (Airplane)
  - Negative (Background)
- Image encoding
- Features
- Learning
  - Classifier Model
    - Linear SVM

Testing
- Test Images
- Image encoding
- Features
- Scoring
  - Scores
    - 0.91
    - 0.12
    - 0.65
    - 0.89
- Ranking
  - Ranked List
    - 0.91
    - 0.89
    - 0.65
    - 0.12
Why a linear SVM?

Advantages of linear SVM:

\[ f(x) = w^\top x + b \]

- Training (Learning)
  - Very efficient packages for the linear case, e.g. LIBLINEAR for batch training and Pegasos for on-line training.
  - Complexity \( O(N) \) for \( N \) training points (cf \( O(N^3) \) for general SVM)

- Testing (Classification)

  Non-linear
  \[ f(x) = \sum_i^S \alpha_i k(x_i, x) + b \]
  \( S = \# \) of support vectors
  \( = (\text{worst case}) \) \( N \)
  \( = \) size of training data

  Linear
  \[ f(x) = \sum_i^S \alpha_i x_i^\top x + b \]
  \[ = w^\top x + b \]
  Independent of size of training data
Nearest Neighbours

Given a query vector $q$ find the closest vector $x$ in the dataset

$$\text{NN}(q) = \arg \min_x ||x - q||^2$$

- $d$ dimensional vectors
- $n$ size of dataset

NN and linear classifiers:

Given a classifier $w$ score each vector $x$ in the dataset as $w.x$ and sort

$$||w - x||^2 = ||w||^2 + ||x||^2 - 2w.x = 2 - 2w.x$$

with $x$ and $w$ having a unit norm.
Nearest Neighbours

Given a query vector $q$ find the closest vector $x$ in the dataset

$$\text{NN}(q) = \arg \min_x ||x - q||^2$$

- $d$ dimensional vectors
- $n$ size of dataset

Problems:
- Costly operation of exact exhaustive (linear) search: $O(n*d)$
- High-dimensional vectors: for exact search the best approach is the naïve exhaustive comparison

Related task: $k$-NN
- find $K$ nearest neighbours
- used in retrieval for ranked lists
The cost of (efficient) exact matching

But what about the actual timings? With an efficient implementation.

Finding the 10-NN of 1000 distinct queries in 1 million vectors
- Assuming 128-D Euclidean descriptors
- i.e., 1 billion distances, computed on a 8-core machine

How much time?
The cost of (efficient) exact matching

But what about the actual timings? With an efficient implementation!

Finding the 10-NN of 1000 distinct queries in 1 million vectors
- Assuming 128-D Euclidean descriptors
- i.e., 1 billion distances, computed on a 8-core machine

5.5 seconds
Need for approximate nearest neighbours

To improve the scalability:

- **Approximate nearest neighbor (ANN) search**

Three (contradictory) performance criteria for ANN schemes

- search quality (retrieved vectors are actual nearest neighbors)
- speed
- memory footprint
Finding *approximate* nearest neighbour vectors

- Approximate method is not guaranteed to find the nearest neighbour.
- Can be much faster, but at the cost of missing some nearest matches
Three ANN methods

- Locality sensitive hashing (LSH)
- Random k-d-trees
- Product quantization
Locality Sensitive Hashing (LSH)

- Choose a random projection
- Project points
- Points close in the original space remain close under the projection
- Unfortunately, converse not true
- Answer: use multiple quantized projections which define a high-dimensional “grid”
Locality Sensitive Hashing (LSH)

- Cell contents can be efficiently indexed using a hash table
- Repeat to avoid quantization errors near the cell boundaries
- Point that shares at least one cell = potential candidate
- Compute distance to all candidates
LSH: discussion

In theory, query time is $O(kL)$, where $k$ is the number of projections and $L$ is the number of hash tables.

I.e. independent of the number of points, $N$.

In practice, LSH has high memory requirements as large number of projections/hash tables are needed.

Code and more materials available online:

http://www.mit.edu/~andoni/LSH/

See also:


Slide: Josef Sivic
Randomized k-d trees

- Use multiple, randomized k-d trees for search

- A k-d tree hierarchically decomposes the descriptor space

- Points nearby in the space can be found (hopefully) by backtracking around the tree some small number of steps

- Single tree works OK in low dimensions – not so well in high dimensions
K-d tree

• K-d tree is a binary tree data structure for organizing a set of points in a K-dimensional space.

• Each internal node is associated with an axis aligned hyper-plane splitting its associated points into two sub-trees.

• Dimensions with high variance are chosen first.

• Position of the splitting hyper-plane is chosen as the mean/median of the projected points.
K-d tree construction

Simple 2D example
K-d tree query

Slide credit: Anna Atramentov
K-d tree: Backtracking

Backtracking is necessary as the true nearest neighbor may not lie in the query cell.

But in some cases, almost all cells need to be inspected.

Figure 6.6
A bad distribution which forces almost all nodes to be inspected.

Figure: A. Moore
Randomized K-d trees

- How to choose the dimension to split and the splitting point?
  - Pick dimension with the highest variance
  - Split at the mean/median

- Multiple randomized trees increase the chances of finding nearby points (shared priority queue)
Randomized K-d trees: discussion

- Find approximate nearest neighbor in $O(\log N)$ time, where $N$ is the number of data points.

- Increased memory requirements: needs to store multiple (~8) trees

- Good performance in practice for recognition problems (NN-search for SIFT descriptors).

- Code available online: http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN
Comparison of approximate NN-search methods


FAST APPROXIMATE NEAREST NEIGHBORS
WITH AUTOMATIC ALGORITHM CONFIGURATION

Marius Muja, David G. Lowe
Computer Science Department, University of British Columbia, Vancouver, B.C., Canada
mariusm@cs.ubc.ca, lowe@cs.ubc.ca

Keywords: nearest-neighbors search, randomized kd-trees, hierarchical k-means tree, clustering.

Abstract: For many computer vision problems, the most time consuming component consists of nearest neighbor matching in high-dimensional spaces. There are no known exact algorithms for solving these high-dimensional problems that are faster than linear search. Approximate algorithms are known to provide large speedups with only minor loss in accuracy, but many such algorithms have been published with only minimal guidance on selecting an algorithm and its parameters for any given problem. In this paper, we describe a system that answers the question, “What is the fastest approximate nearest-neighbor algorithm for my data?” Our system will take any given dataset and desired degree of precision and use these to automatically determine the best algorithm and parameter values. We also describe a new algorithm that applies priority search on hierarchical k-means trees, which we have found to provide the best known performance on many datasets. After testing a range of alternatives, we have found that multiple randomized k-d trees provide the best performance for other
Comparison of approximate NN-search methods

Dataset: 100K SIFT descriptors

Code for all methods available online, see Muja&Lowe’09
Searching using Product Quantization

- Vector split into m subvectors: \( y \rightarrow [y_1 \ldots y_m] \)
- Subvectors are quantized separately
- Toy example: \( y = 8\)-dim vector split into 4 subvectors of dimension 2

\( y_1 \): 2 components

- 8\(^4\)=4,096 centroids induced
- For a quantization cost equal to that of 8 centroids

- In practice: 8 bits/subquantizer (256 centroids)
  - SIFT: \( m = 4 \)-16

Jegou et al, PAMI 2011
Searching using Product Quantization

- Estimate distances in the compressed domain $d(x, y)^2 \approx \sum_{i=1}^{m} d(x_i, q_i(y_i))^2$

- To compute distances between query $x$ and many codes:

  I-

  ![Diagram showing vector spaces and distances]

  Precompute all distances between query subvectors and centroids:

  $d(x_i, c_{i,j})^2$

  Stored in look-up tables computed per query descriptor

  ![Table of distances]

  II-

  For each database vector: sum the elementary square distances

  $m-1$ additions per distance

Jegou et al, PAMI 2011
Example stats

3 Million key frames

Original descriptors 8K dimension

- Memory footprint: $8k \times 4 \times 3M = 96 \text{ GB}$

Product Quantization: $8k \times 4 \rightarrow 2k$

- Memory footprint: $2k \times 3M = 6 \text{ GB}$

Product Quantization for vector compression, Jegou et al., PAMI 2011