Discriminative Learning and Big Data

Approximate Nearest Neighbours

Relja Arandjelović
DeepMind
www.relja.info

and Andrew Zisserman
Big Data means at least …

• Image search, e.g. Facebook
  • 250 billion images = 250 billion descriptors (e.g. 128D based on CNNs)

• Video search: millions of hours of video, e.g. YouTube
  • Billions of audio and video descriptors

• Music search: e.g. Shazam

• The order of magnitude considered in this lecture is millions to billions
Example: 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 classifier 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
  - Linear SVM Classifier

Testing
- Dataset
- Image encoding
- Features
- Scoring
- Scores
- Ranking
  - Ranked List
  - 0.91
  - 0.89
  - 0.65
  - 0.12
Video dataset: BBC TV

- 17K broadcasts from BBC 1, 2, 3, 4 & BBC News 24
- Programmes from 2007 to 2012 in prime time slot (7-12pm)
- 10K hours of video represented by 1 frame per second
- 36M seconds of data, 5M keyframes
- Frames are 480 x 270 pixels
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 and for classification
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
NN Applications: What is this?

Search for the query descriptor in a large database
Copy the meta-data of the nearest neighbour

HOW DOES SHAZAM WORK?

This is a question we get often asked. Here is a quick summary of the three main steps involved from the moment you Shazam until the magic happens.

Let’s say you are in a shop and you like the music you’re hearing. Start the app and tap the Shazam button.

A digital fingerprint of the audio is created and, within seconds, matched against Shazam’s database of millions of tracks and TV shows.

You are then given the name of the track and the artist and information such as lyrics, video, artist biography, concert tickets and recommended tracks. We also let you buy or listen to the song using one of our partners’ services.

From www.shazam.com
NN Applications: Automatic panorama creation

Match local patches, estimate the geometric transformation

(c) SIFT matches 1
(d) SIFT matches 2
(e) Images aligned according to a homography

From Brown & Lowe “Automatic Panoramic Image Stitching using Invariant Features”
NN Applications: Reinforcement Learning

Act in the environment by consulting past memories
Search memories using the current state

From Min & Kim “Learning to Play Visual Doom using Model-Free Episodic Control”
NN Applications: Data Visualization (t-SNE)

Compute nearest neighbours for all points
Flatten the high dimensional data preserving the neighbourhood structure

Taken from https://theconceptcenter.com/simple-research-study-similarity-between-two-words/
NN Applications: Other

- k-NN classification and regression
- Clustering
- Hard negative mining
- Semi-supervised learning

...
Efficient implementations

Cost $O(nd)$ but many optimization tricks

- Scalar product instead of distance if possible
- Special CPU instructions (SIMD in SSE)
- Pipelining / loop expansion
- Multiple queries: clever use of cache

For best performance use standard libraries

- Lower-level: BLAS
- Higher-level: Vectorized MATLAB, numpy, ..
- Good free libraries:
  - yael_nn (http://yael.gforge.inria.fr/)
  - FAISS (https://github.com/facebookresearch/faiss)
How to speed up?

Dimensionality reduction?

- Reasonable first step, but typically insufficient

Use GPU?

- Adds lots of complexity
- Insufficient memory
- Overhead for memory copying between CPU & GPU

Buy more machines?

- Costs money to buy, maintain, ..
- Adds lots of complexity
- For real-time systems: communication overhead
- Still often insufficient, e.g. if all pairwise distances are needed (e.g. building a neighbourhood graph, clustering, ..)
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
Approximate Nearest Neighbours (ANN)

Is finding only approximate nearest neighbours acceptable?

- Often there is no choice
  - Use ANN or not = have Google or not
- Often it is good enough
  - What is this?

?   Big Ben
Approximate Nearest Neighbours (ANN)

Is finding only approximate nearest neighbours acceptable?

- Often there is no choice
  - Use ANN or not = have Google or not

- Often it is good enough
  - What is this?

¿ Big Ben
Approximate Nearest Neighbours (ANN)

Is finding only approximate nearest neighbours acceptable?

- Often there is no choice
  - Use ANN or not = have Google or not

- Often it is good enough
  - What is this?
  - Geometric matching: Only need sufficient number of matches
  - Data visualization: Rough nearest neighbours are fine
Approximate Nearest Neighbours (ANN)

Three (contradictory) performance criteria for ANN schemes

- Search quality
  - Retrieved vectors are actual nearest neighbors
- Speed
- Memory footprint
  - Representation must fit into memory, disk is too slow
Approximate Nearest Neighbours (ANN): Methods

Broad division

1. Approximate the vectors => do faster distance computations
   • Brute-force search, visit all points: O(n)

2. Approximate the search => do fewer distance computations
   • Do not visit all points

Can (and should) mix and match: approximate both, rerank (1.) with original vectors, etc
Approximate Nearest Neighbours (ANN): Methods

Broad division

1. Approximate the vectors => do faster distance computations
   • Brute-force search, visit all points: O(n)

2. Approximate the search => do fewer distance computations
   • Do not visit all points
ANN via approximating vectors

• Objectives
  • Approximate vectors as well as possible
  • Make the distance computation fast
  • Effectively: quantization, compression

• Do we need all 32 bits in a float?
  • Sometimes yes:
  • Sometimes no:
  • The limit is entropy (amount of information)
  • Betting on exploiting underlying structure
    • Otherwise there is no hope
    • zip, jpeg, mpeg, .. do not always compress!
ANN via approximating vectors: Methods

- Hashing
  - LSH
- Vector Quantization
  - Product Quantization
Hashing (for approximating vectors)

- **Key idea**
  - Convert the input real-valued vector into a binary vector
  - Compare binary vectors using Hamming distance
    
    \[
    \begin{array}{c}
    x: 00100101 \\
    y: 10100111 \\
    \end{array}
    \]
    
    \[
    \begin{array}{c}
    0000110 \\
    \hline
    10000110 \Rightarrow \text{Hamming distance} = \text{sum(1's)} = 3
    \end{array}
    \]

- **Pros**
  - Memory savings: \(d \times 32\) bits -> \(b\) bits
  - Extremely fast distance computation: \(\text{popcnt}(\text{xor}(x,y))\)

- **Cons**
  - Can be hard to design a good hashing function
Choose a random projection

Project points, pick a random threshold

Points close in the original space remain close under the projection

Unfortunately, converse not true

Locality Sensitive Hashing (LSH)
Locality Sensitive Hashing (LSH)

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

• Pros
  • Very simple to implement
  • Very fast
  • Good for embedded systems
Locality Sensitive Hashing (LSH): Discussion

• Pros
  • Very simple to implement
  • Very fast
  • Good for embedded systems

“Choose a random projection”
“Betting on exploiting underlying structure”

• Cons
  • Data-agnostic = blind
  • Need a large number of projections (=bits=a lot of memory) for decent performance compared to alternatives

• The way to go: data-dependent hashing
  • Iterative Quantization [Gong & Lazebnik 2011], Spectral Hashing [Weiss et al. 2009], …
ANN via approximating vectors: Methods

- Hashing
  - LSH

- Vector Quantization
  - Product Quantization
Vector Quantization

Hard to design the conversion real vector -> binary vector, such that distances in binarized space are meaningful

Idea

• Scrap using Hamming distance to compare codes
• Codebook (vocabulary) of $k$ vectors (codewords)
• Code for vector = ID of the nearest codeword ($\log_2 k$-bits)
• Learn codebook to optimize for quantization quality
Vector Quantization: Pros

- Great data dependence
- Easy to learn good codebook: k-means
- Fast search
  - Offline: Pre-compute all distances between codewords
  - Online: \( d(x, y) \approx d(q(x), q(y)) \); lookup table

- Asymmetric distance
  - No need to quantize the query!
  - Online:
    - Compute all \( d(x, \text{codeword}) \)
    - \( d(x, y) \approx d(x, q(y)) \); lookup table
Vector Quantization: Cons

- Not scalable at all
  - E.g. for a standard 64-bit size code
    - Codebook size $k = 2^{64} = 1.8 \times 10^{19}$
  - Not possible to learn
    - Slow
    - Insufficient training data
  - Huge memory requirements for codebook and lookup tables
  - Huge cost of distance pre-computation
Product Quantization (PQ)

Key idea

• Divide vectors into $m$ subvectors (blocks)
• Vector Quantize each subvector independently
• E.g. for a standard 64-bit size code
  • 8 blocks quantized with 8 bits each
  • Sub-codebook size: $2^8=256$
  • Equivalent to vector quantizing with a $2^{64}=1.8 \times 10^{19}$ codebook
• Effective codebook: codewords are the Cartesian product of block-wise codewords
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

\[ 8^4 = 4,096 \] centroids induced
for a quantization cost equal to that of 8 centroids

- In practice: 8 bits/subquantizer (256 centroids)

Jegou et al, PAMI 2011
Searching using Product Quantization

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

- To compute distances between query \( x \) and many codes:

I-

Precompute all distances between query subvectors and centroids:
\[ d(x_i, c_{i,j})^2 \]

Stored in look-up tables computed per query descriptor

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

Use PQ with 4 dim sub-vectors, and 1 byte per sub-vector (256 centres)

Original descriptors 8k dimension

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

Product Quantization: 8k x 4 -> 2k

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

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

Pros

• Large reduction in memory footprint
• Large speedup in NN search (c.f. exhaustive search)
• Good performance in practice

Cons

• Slower than hashing, but usually fast enough
• Assumes subvectors are independent
  • In practice: first decorrelate subvectors (PCA)
• Assumes subvectors contain equal amount of information
  • In practice: balance information across blocks
• See Optimized Product Quantization [Ge et al. 2013] and Cartesian K-Means [Norouzi & Fleet 2013]

Code: FAISS (https://github.com/facebookresearch/faiss)
ANN via approximating vectors: Comparison

Memory vs Quality
recall@100: Proportion of times the true NN is in top 100

Vector Quantization
• PQ, OPQ, CKM

Hashing
• LSH, SH, MDSH, etc

Vector Quantization beats Hashing
ANN via approximating vectors: Comparison

Memory vs Quality
recall@100: Proportion of times the true NN is in top 100

Vector Quantization
- PQ, OPQ, CKM

Hashing
- LSH, SH, MDSH, etc

Vector Quantization beats Hashing
Approximate Nearest Neighbours (ANN): Methods

Broad division

1. Approximate the vectors => do faster distance computations
   • Brute-force search, visit all points: O(n)

2. Approximate the search => do fewer distance computations
   • Do not visit all points
ANN via approximating the search: Methods

- Space partitioning
  - Hashing
  - Vector Quantization
  - K-d trees
Space partitioning

- Partition the vector space into C partitions

- Query
  - Search inside the nearest k partitions

- Does not decrease memory requirements
  - Can combine with quantization

- Performance
  - For C=const - still $O(nd)$, just $O(nd^*k/C)$
  - $k/C$: quality vs speed
Curse of dimensionality

N=1 billion, r=?

Packing hyper-spheres

1-D
r=1e-9

2-D
r=3e-5

$V_S = k_d r^d$

$V_L = k_d R^d = k_d$

$V_L = NV_S$

$k_d = Nk_d r^d$

$r = \left(\frac{1}{N}\right)^{\frac{1}{d}}$
Space partitioning revisited

• Partition the vector space into C partitions

• Query
  • Search inside the nearest k partitions

• Does not decrease memory requirements
  • Can combine with quantization

• Performance
  • For C=const - still $O(nd)$, just $O(nd \cdot k/C)$
  • $k/C$: quality vs speed
Space partitioning revisited

- Partition the vector space into $C$ partitions

- Query
  - Search inside the nearest $k$ partitions

- Does not decrease memory requirements
  - Can combine with quantization

- Performance
  - For $C=\text{const}$ - still $O(nd)$, just $O(nd*k/C)$
  - $k/C$: quality vs speed
ANN via approximating the search: Methods

- Space partitioning
  - Hashing
  - Vector Quantization
  - K-d trees
Hashing (for approximate search)

Recall: Each bit partitions space into two

Query

- Visit partitions in order of increasing Hamming distance of the query hash
Hashing (for approximate search): Discussion

How many partitions with Hamming distance $r$ for $b$-bits? $\binom{b}{k}$

- For $b=64$, $r=3$: 42k
- For $b=64$, $r=4$: 635k

Necessary solution

- Multiple partitions with smaller number of bits each
- Same vector (or its ID) is stored multiple times

Pros

- If hashing is used for quantization, partitions are directly available

Cons

- Large memory requirements for good performance
- As before: hard to get good hashing functions
ANN via approximating the search: Methods

- Space partitioning
  - Hashing
  - **Vector Quantization**
  - K-d trees
Vector Quantization (for approximate search)

Partition the space using k-means

Query

• Partition visit order: k nearest partitions obtained via NN search for cluster centres

Complexity

• Ordering partitions: $O(Cd)$
• Searching within partitions: $O(ndk/C)$
• Rule of thumb: $C \sim \sqrt{n}$
Vector Quantization (for approximate search)

Pros

• Simple and intuitive
• As before: good data dependence
• Generally good performance

Cons

• Ordering partitions can take a significant chunk of the computation budget
  • Can do ANN for this stage as well, though usually ok

Code: FAISS (https://github.com/facebookresearch/faiss)

Further reading

• Can obtain finer partitioning through PQ, see Inverted Multi-Index [Babenko & Lempitsky 2012]
ANN via approximating the search: Methods

- Space partitioning
  - Hashing
  - Vector Quantization
- **K-d trees**
K-d tree

A k-d tree hierarchically decomposes the vector space

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

- Multiple randomized trees increase the chances of finding nearby points (shared priority queue)

- How to choose the dimension to split and the splitting point?
  - Pick random dimension from high variance ones
  - Split at the mean/median
Randomized K-d trees: Discussion

Pros

- Find approximate nearest neighbor in $O(\log n)$ time, where $n$ is the number of data points
- Good ANN performance in practice for “low” dimensions (e.g. 128-D)

Cons

- Increased memory requirements: needs to store multiple (~8) trees
- Not so good in “high” dimensions

Code available online:

http://www.cs.ubc.ca/research/flann/
Approximate Nearest Neighbour search: Overview

**Approximate the vectors**: fast distances, memory savings
- Hashing
  - LSH, see ITQ, Spectral Hashing, ..
- Vector Quantization
  - **Product Quantization**, see OPQ, Cartesian K-means, ..

**Approximate the search**
- Non-exhaustive through space partitioning
- Hashing
- **Vector Quantization**
- (Randomized) K-d trees
- Mind the dimensionality

In real life – use a **hybrid** approach
- E.g. Google Goggles/Photos 2011: Vector Quantization + OPQ
There is more …

Inverted file indexing (Computer Vision lectures)

Beyond Euclidean distance / scalar product
  - MinHash for Jaccard similarity (set overlap)
  - Winner Take All hashing for rank correlation
  - Other kernels: Explicit feature map

Supervised methods - current and future work
  - Training representations for ANN search
    - Neural networks for hashing
      - “Supervised Hashing for Image Retrieval via Image Representation Learning” [Xia et al. 2014]
    - Neural networks for PQ
      - “SUBIC: A supervised, structured binary code for image search” [Jain et al. 2017]