Multidimensional hashing
• Current techniques for individual object recognition can only handle up to few thousand objects.

• The challenge is to scale up these techniques by three to five orders of magnitude. Scaling object recognition to millions of objects has a dual in the problem of specific object retrieval in large image collections. When dealing with vocabularies of 1 million words, exact clustering methods such as K-means become infeasible, as their computational cost is $O(NK)$ for $K$ words, $N$ items in the training set.

• Solutions exploit:
  – Kd-trees: they are used canonically to search for approximate nearest neighbors
  – Efficient hash functions: they have been used to approximate nearest neighbor search
  – Hierarchical clustering of descriptors: it has been used to produce a vocabulary which is scored with the vector space model.
Hashing methods

• Use of hashing to group similar images in large image databases, is such that the whole image is represented by a single short descriptor, similar to the bag-of-words representation. The hashing techniques then become an alternative for finding nearest neighbors in the image database.

• Hashing techniques allow broad search:
  – It supports efficient similarity search of high-dimensional data. Similar feature points are mapped to the same buckets with high probability
  – They might be very important for large databases recognition given their $O(Nc), c < 1$ search time, as current methods such as the popular vector space search have $O(N)$ search time.
  – They are especially useful in domains where there is no prior knowledge about how to cluster or model data...
• Types of hashes:
  – Exact hashing: put *Bash* vs *Bush* in different bins
  – (Approximate) similarity sensitive: close matches in same bin

• Nearest Neighbor implementations
  – Pair-wise distance computation: 1,000,000,000,000 comparisons in 2M song database
  – Hash bucket collisions: 1,000,000,000 hash projections
Sub-linear time image search

- Dimension reduction
  - $K$ dimensions
  - $g(v) = (h_1(v), h_2(v), \ldots, h_K(v))$

- Hash family
  - $L$ hash tables
  - $\{g_1(v), g_2(v), \ldots, g_L(v)\}$

The hashing function used should place images with most similar local feature matchings into the same hash buckets.
Several typical distances that can be used with hashing:
- Hamming distance
- L1 distance
- $p$-stable distribution: L2(Gaussian), L1(Cauchy) etc.
- Jaccard (Min-hash)
- Arccos (random projection)
- L2 distance on a sphere
Hamming distance

- Random choose K dimensions
  - If two features are similar, their value in each dimension will be similar, too. Ex, point = 01101
- Choose 1, 3, 5 dimensions
  - $g(p) = (0, 1, 1)$, Inner product projection
  - $h_1(p) = p \cdot (1, 0, 0, 0, 0) = 0$
  - $h_2(p) = p \cdot (0, 0, 1, 0, 0) = 1$
  - $h_3(p) = p \cdot (0, 0, 0, 0, 1) = 1$
  - $h_k(p) = p \cdot (0/1, 0/1, 0/1, 0/1, 0/1)$

- How to convert L1 distance into hamming distance?
  - Find maximum number among features (e.g., $f_v = (6, 2, 3, 1, 5, 3, 2, 0)$)
  - Change integer to the same number of ones
  - Appended zero

- E.g., maximum number is 6
  - 1 $\rightarrow$ 100000
  - 5 $\rightarrow$ 111110
Ls distance (\(\rho\)-Stable)

- a : stable distribution
- L2(Gaussian), L1(Cauchy) etc.
- b : 0 ~ W
Arccos (Random projection)

- Consider the distance measure that is the angle between the two vectors

- Hash function
  - $h(v) = 1, \ a \cdot v \geq 0$
  - $h(v) = 0, \ a \cdot v < 0$

\[
\theta(p, q) = \arccos\left(\frac{p \cdot q}{\|p\|\|q\|}\right)
\]
Jaccard (Min-hash)

- To measure the similarity between two sets

\[
sim(A_1, A_2) = \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|} = P(\min \pi(A_1) = \min \pi(A_2))
\]

\[
\pi: random\ permutation
\]

\[
h_\pi(v) = \min\{\pi(w) | w \in v\}
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>FCAE</td>
<td>D</td>
<td>AEF</td>
</tr>
<tr>
<td>H2</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>ABDF</td>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>H3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>CBAE</td>
<td>F</td>
<td>D</td>
</tr>
<tr>
<td>H4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>EFB</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

\[S(ABC, AEF) = \frac{1}{4} = 0.25\]
Real: \(\frac{1}{5} = 0.2\)

\[S(ABC, BCD) = \frac{3}{4} = 0.75\]
Real: \(\frac{2}{4} = 0.5\)

s-tuples (sketches)
Ex. \(s=2\)
(k hashing functions)
Approximate nearest neighbor

LSH is a fast method for approximate nearest neighbor search in high dimensional space. A problem with LSH is that *signal distance can be dominated by nuisance variables*

- Input
- Rendered (\& hashed)
- Pose DB
- Similar examples fall into same bucket in one or more hash table
Similarity Sensitive Embedding

- A solution is to define a new embedding (hash function) for LSH that is most relevant to some parameter similarity.

- Parameter similarity sensitive hashing can find distance related to task. It can effectively learn problem-dependent distance measure and is an efficient means to index. Compute embedding $H: \{0, 1\}^N$ such that

$$| H(I(\theta_1)) - H(I(\theta_2)) | \text{ is small if } \theta_1 \text{ is close to } \theta_2$$

$$| H(I(\theta_1)) - H(I(\theta_2)) | \text{ is large otherwise}$$

- Use the embedding with approximate nearest neighbors retrieval (LSH). Find $H$ by training boosted classifier to learn “same-pair” and concatenate resulting weak learners ...

[Shakhnarovich 2005]
**Similarity-sensitive** hashing

![Diagram showing similarity-sensitive hashing process]

- **Input**
- **Pose-sensitive Hash fcns.**
- **Pose DB**
- **Rendered (& hashed)**

... similar in pose, not image

[Shakhnarovich et. al '03, Shakhnarovich '05]
Locality Sensitive Hashing

- Locality Sensitive Hashing (LSH)

The key idea of the LSH approximate nearest neighbor (NN) algorithm is to construct a set of hash functions such that the probability of nearby points being close after transformation with the hash function is larger than the probability of two distant points being close after the same transformation.

- The range space of the function is discretized into buckets and we say that there is a ‘collision’ when two points end up in the same bucket.
- LSH has been shown to work well on high-dimensional datasets
- Has a query time that scales sub-linearly with the dataset size under certain conditions. This scheme could in the worst case make the query time grow like $O(N)$. 
Locality Sensitive Hashing

- Locality sensitive hashing is a method for approximate nearest neighbour computation that is particularly useful for large datasets Gionis, A. & Indyk, P. & Motwani, R. (1999).
  - Take random projections of data
  - Quantize each projection with few bits through hashing

For our N bit code:
- Compute first N PCA components of data
- Each random projection must be linear combination of the N PCA components
Locality Sensitive Hashing

- Hash functions are locality-sensitive, if, for a hash function \( h \), for any pair of points \( p, q \) we have:
  - \( \text{Pr}[ h(p) = h(q) ] \) is “high” if \( p \) is “close” to \( q \)
  - \( \text{Pr}[ h(p) = h(q) ] \) is “low” if \( p \) is “far” from \( q \)

- Given two image points \( p, q \) and a distance function \( D() \):
  - \( D(p, q) \leq R, \text{Pr}[ h(p) = h(q) ] \geq p_1 \)
  - \( D(p, q) \geq cR, \text{Pr}[ h(p) = h(q) ] \leq p_2 \)
  being \( p_1 > p_2 \)

- Due to the linearity of the dot product, the difference between two image points \( ||h(p) - h(q)|| \) has a magnitude whose distribution is proportional to \( ||p - q|| \):
  e.g., \( \text{Pr}[ h(p) = h(q) ] = 1 - \text{dist}(p, q)/d \)

- Similar data have much higher probability to hash into the same bucket than dissimilar data.
  Twisting parameters to increasing \( p_1 \) and reducing \( p_2 \)
k projections

m dictionaries (buckets)

Hash

Count Matches

Query

\[ g_j \]

Hash Bucket

\[ t_1 \]

Fingerprint

\[ t_2 \]
Pyramid hashing

Pyramid hashing is a sub-linear time randomized hashing algorithm for indexing sets of feature vectors according to their partial correspondences: it is concerned with the problem of how, given a set of feature vectors, to efficiently retrieve the most similar sets from a database of sets, with similarity defined in terms of one-to-one correspondences.

It applies to general matchings not only between object instances, but also between textures or categories, which often exhibit stronger appearance variation and may not be isolated from a database on the basis of a few discriminative features alone and instead, the joint matching of all component features may be preferable.

Good for category level comparisons.
Pyramid Match Hashing

• Grauman & Darell, CVPR 2007

• Combines Pyramid Match Kernel (efficient computation of correspondences between two set of vectors) with Locality Sensitive Hashing (LSH) [Indyk & Motwani 98]

• Allows matching of the set of features in a query image to sets of features in other images in time that is sublinear in # images
The method

- Encodes a point set with a weighted multi-resolution histogram in such a way that a dot product between any two such encodings will reflect the similarity of the original point sets according to an approximate, normalized partial matching between their component feature vectors.

- Embeds a locality-sensitive hash function which guarantees that examples with strong matching similarity will (with high probability) hash into the same buckets.

- Approximate similarity search in the Hamming space of the hash keys to identify the approximate nearest neighbors according to the approximate matching score, in sub-linear time in the number of database examples.

- $(1+\varepsilon)$-approximate nearest neighbor images in $O(N^{1/(1+\varepsilon)})$ time for a database of $N$ images, each of which is represented by a set of local features.
What locality hash function

- Given a collection of vectors \( \{v_1, \ldots, v_n\} \) belonging to the unit sphere, and a randomly generated vector \( r \), the probability that any two vectors \( v_i \) and \( v_j \) each has a dot product with \( r \) having an opposite sign is related to the vectors as follows:

\[
Pr [ \text{sgn}(v_i \cdot r) \neq \text{sgn}(v_j \cdot r) ] = \frac{1}{\pi} \cos^{-1}(v_i \cdot v_j).
\]

i.e. the probability a random hyperplane separates two vectors is directly proportional to the angle \( \cos^{-1}(v_i \cdot v_j) \).

- This property may be exploited for locality sensitive hashing. Given a database of vectors in \( \mathbb{R}^d \), a vector \( r \) is chosen at random from the \( d \)-dimensional Gaussian distribution with zero mean and unit variance.

- The corresponding hash function \( h_r \) accepts a vector \( u \in \mathbb{R}^d \), and is defined as:

\[
h_r(u) = 1 \quad \text{if} \quad (r \cdot u) \geq 0; \quad h_r(u) = 0 \quad \text{if} \quad (r \cdot u) < 0
\]

Then, a valid locality sensitive hashing scheme is:

\[
Pr [ h_r(v_i) = h_r(v_j) ] = 1 - \frac{[\theta(v_i, v_j)]}{\pi} \quad \text{where} \quad \theta(v_i, v_j) = \cos^{-1} ((v_i \cdot v_j) \frac{1}{|v_i| |v_j|})^{-2}
\]
• Step 1 take image descriptors
  take a collection of images, each one of which is represented by a set of feature vectors: for example a set of SIFT descriptors extracted at salient points; a set of shape context histograms; a set of geometric blur descriptors extracted at edge points; a set of color distributions, etc.

• Step 2 database preparation
  database items are prepared by mapping every set of vectors to a single high-dimensional vector via the Pyramid match embedding function

• Step 3 hash encoding
  all embedded database examples are encoded as binary hash key strings, with each bit determined with a random hash function designed to probabilistically give similar responses for examples with similar dot products. Hash keys are stored in such a way that they are accessible in sub-linear time.
• **Step 4 query encoding**
  Given a query image, local features of the chosen type are extracted, and the pyramid match embedding function is applied to form the vector encoding for the query set.

• **Step 5 retrieval**
  Rather than compute the dot product between the embedded query and every embedded database item, we apply the same randomized hash functions used for the database items to index into the stored database hash keys. The pyramid match is applied only to a small portion of the database examples. Thereby (with high probability) obtaining in sub-linear time the most similar database neighbors in terms of normalized partial match correspondences between the original local image features.
Pyramid match hashing performance considerations

- **For fixed-size sets**, Locality-Sensitive Hashing [Indyk & Motwani 1998] provides bounded approximate similarity search. Search a database of over 130K images, still with a query time of 1 second. Significant computational advantages for very little loss in accuracy over a brute force linear scan. Only 1-3% of a database needs to be searched.

- **O(N^(1/1+eps))** query time to retrieve similar images within radius r(1+eps) of query (1-nn for Charikar permutation method?)

- **For varying set sizes**: With pyramid hashing index images may be described by varying numbers of features, and the presence of very distant (“outlier”) features in an image cannot significantly skew the correspondence similarity.
Semantic Hashing

- Map images to compact binary codes
- Hash codes for fast lookup
Binary Code requirements

• Size: image binary codes must fit in memory. The appropriate number of bits for the binary code can be defined based on a few technical considerations:
  – Google has few billion images ($10^9$)
  – current computer technology has ~10 Gbytes ($10^{11}$ bits)
This implies that code size must be $\leq 10^2$ bits/image

• Dimensionality: since:
  – One high resolution image is $10^7$ bits
  – One 32x32 color image is $10^4$ bits
semantic hash function must also reduce dimensionality

• Locality structure: in order to account for image semantic similarity the binary code must preserves the neighborhood structure of the input space

• Computational efficiency: since it is designed to deal with very large datasets it must be fast to compute. Hash tables are therefore used.
Binary codes for images

• Semantic Hashing introduced for text documents [Salakhutdinov & Hinton, 2007] can be extended to images in order to permit fast lookup via hashing. With this approach:
  – Each code is a memory address
  – neighbors are found by exploring the neighbourhood around query address
  – Lookup time depends on the radius of the ball, not on the number of data points
Semantic hashing

- The basic idea is that images with similar content should have similar binary codes.

- Hamming distance between codes can be used to measure the number of bit flips, e.g.:
  \[ \text{Ham\_Dist}(10001010, 10001110) = 1 \]
  \[ \text{Ham\_Dist}(10001010, 11101110) = 3 \]
Image representation: Gist vectors

- Pixels are not a convenient representation for images. GIST descriptors can be more appropriate instead. They well account for human perceptual distance.

- Gist vectors: 512 dimensions/image
- Distance between GIST vectors can be computed according to L2 distance.
Further gains in classification rates with large datasets can be achieved by considering:

- Vocabularies that take into account the information gain of features in the vocabulary construction
- Assigning many words to a visual feature, so-called soft word assignment, can further improve the retrieval performance.
- Contextual distance metrics, which adapt the metric for comparing Bag-of-Words vectors to better discriminate between database images (it however requires pre-computing the nearest neighbors to all images in the database, which takes $O(N^2)$ for large databases).
Content-based image retrieval

Even this is far too slow for any web-scale application!

Pyramid match: ~1 second / query

Optimal match: ~2 hours / query