BLOOM FILTERS - RECAP

Bloom filters

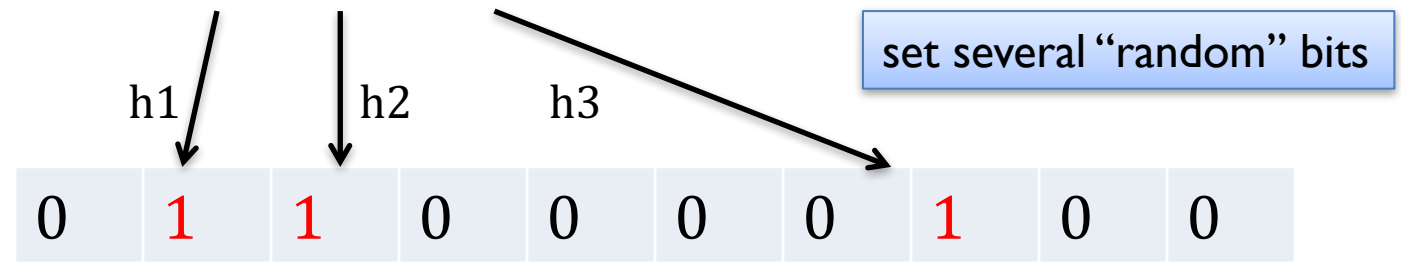
- Interface to a Bloom filter
  - BloomFilter(int maxSize, double p);
  - void bf.add(String s); // insert s
  - bool bd.contains(String s);
    - // If s was added return true;
    - // else with probability at least *1-p* return false;
    - // else with probability at most *p* return true;

  - I.e., a noisy "set" where you can test membership (and that's it)

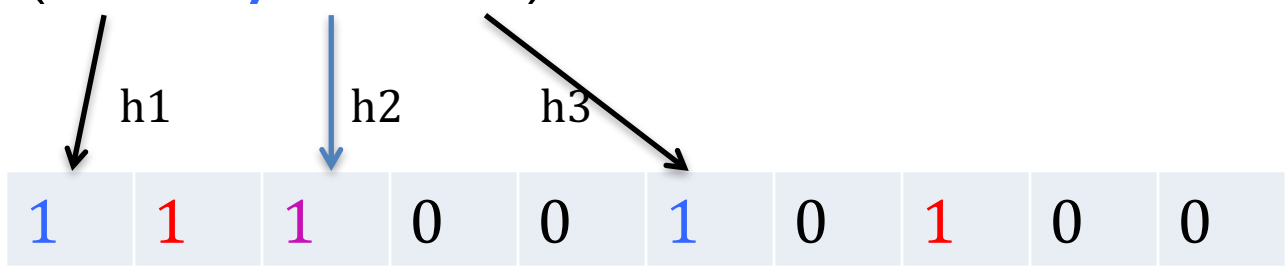
Bloom filters

0	0	0	0	0	0	0	0	0	0

bf.add("fred flintstone"):



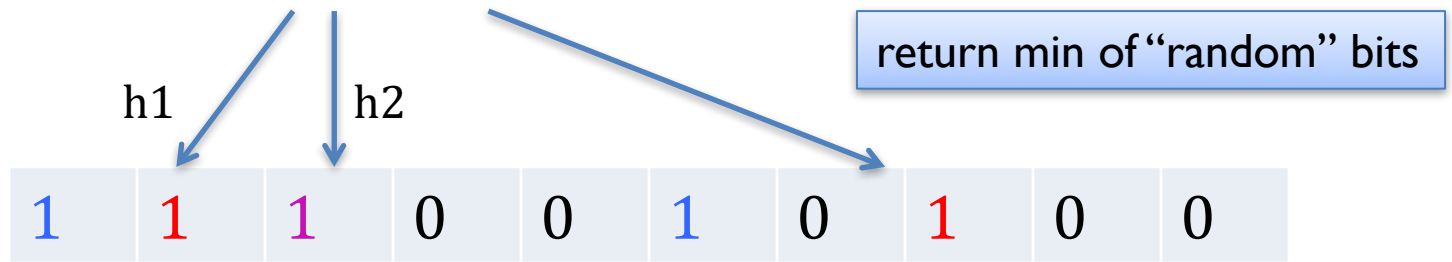
bf.add("barney rubble"):



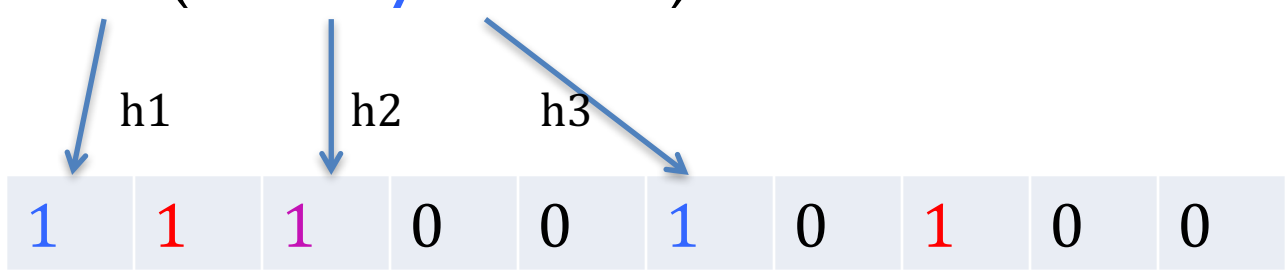
Bloom filters

1	1	1	0	0	1	0	1	0	0

bf.contains ("fred flintstone"):



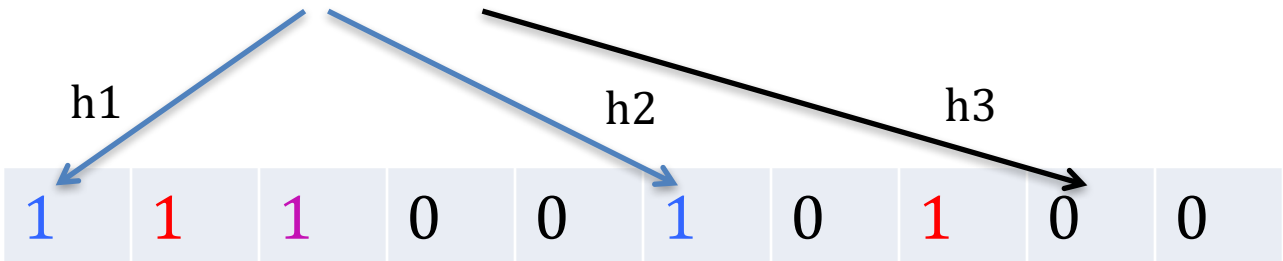
bf.contains("barney rubble"):



Bloom filters

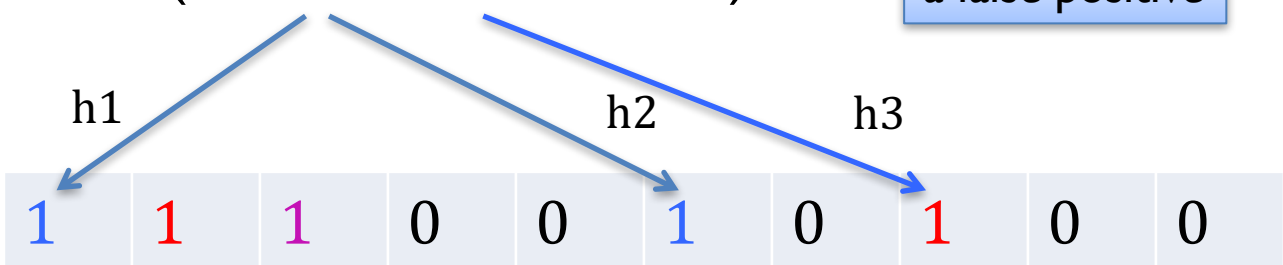
1	1	1	0	0	1	0	1	0	0

bf.contains("wilma flintstone"):



bf.contains("wilma flintstone"):

a false positive



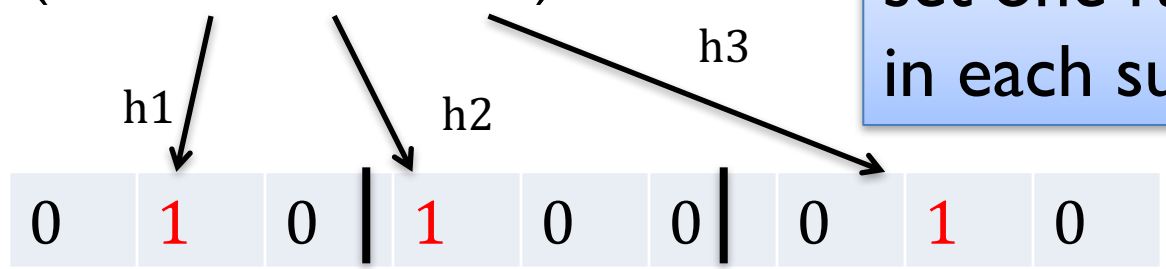
BLOOM FILTERS VS COUNT-MIN SKETCHES

Bloom filters – a variant

split the bit vector into k ranges, one for each hash function

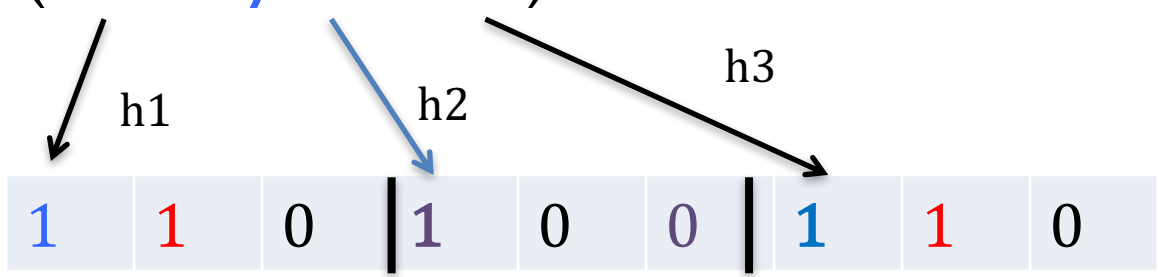
0	0	0	0	0	0	0	0	0

bf.add("fred flintstone"):



set one random bit in each subrange

bf.add("barney rubble"):



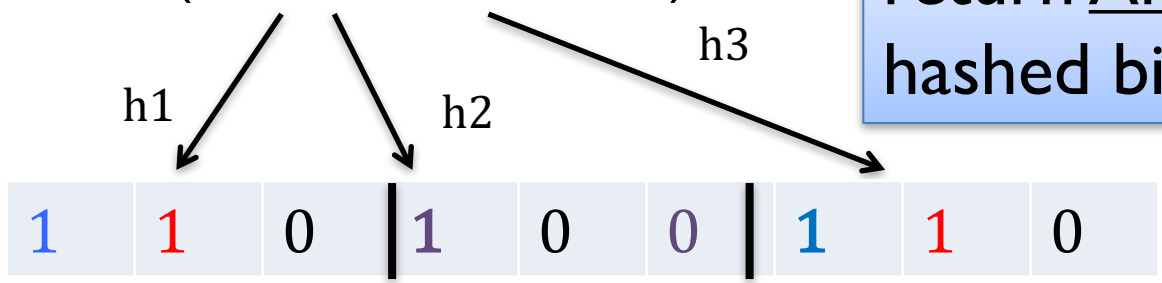
Bloom filters – a variant

split the bit vector into k ranges, one for each hash function

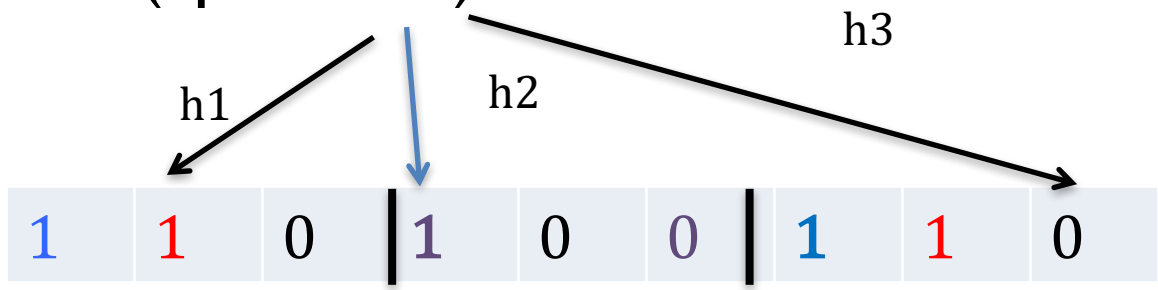
0	0	0	0	0	0	0	0	0

bf.contains("fred flintstore"):

return AND of all hashed bits



bf.contains("pebbles"):



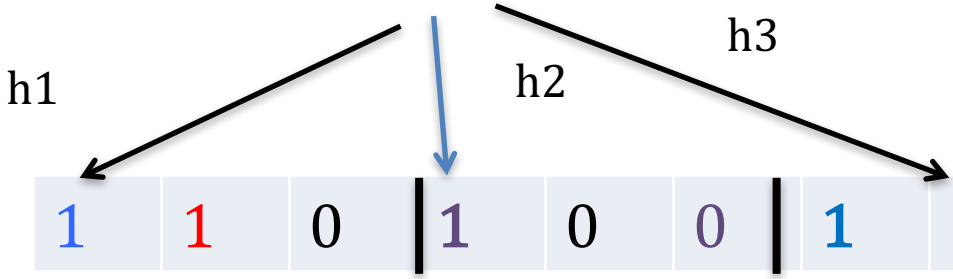
Bloom filters – a variant

split the bit vector into k ranges, one for each hash function

0	0	0	0	0	0	0	0	0

a false positive!

bf.contains("pebbles"):



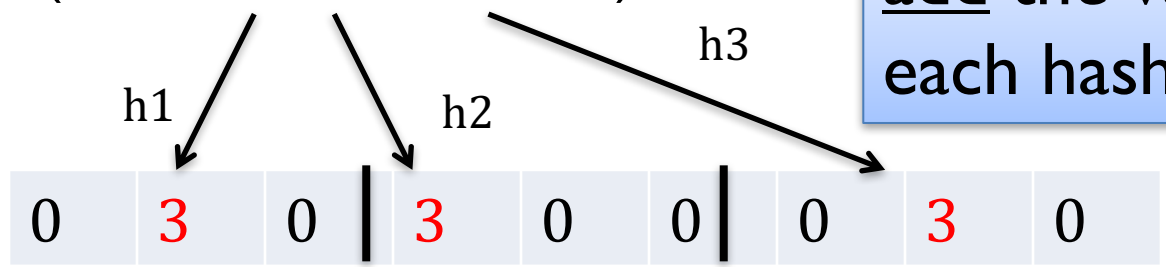
Count-min sketches

split a <u>real</u> vector into k ranges, one for each hash function

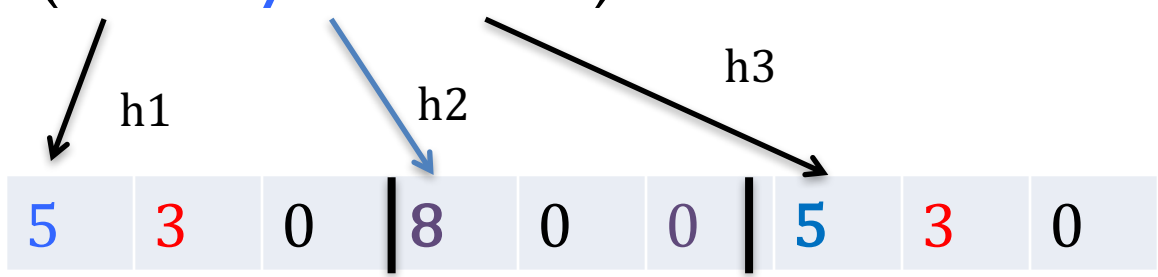
0	0	0	0	0	0	0	0	0

cm.inc("fred flintstone", 3):

<u>add</u> the value to each hash location



cm.inc("barney rubble",5):



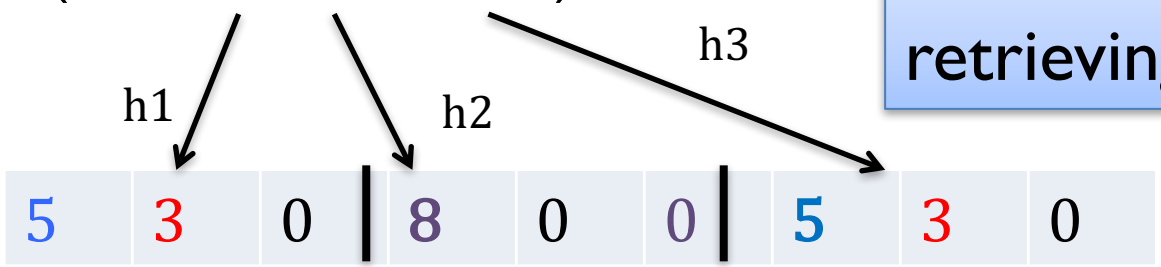
Count-min sketches

split a <u>real</u> vector into k ranges, one for each hash function

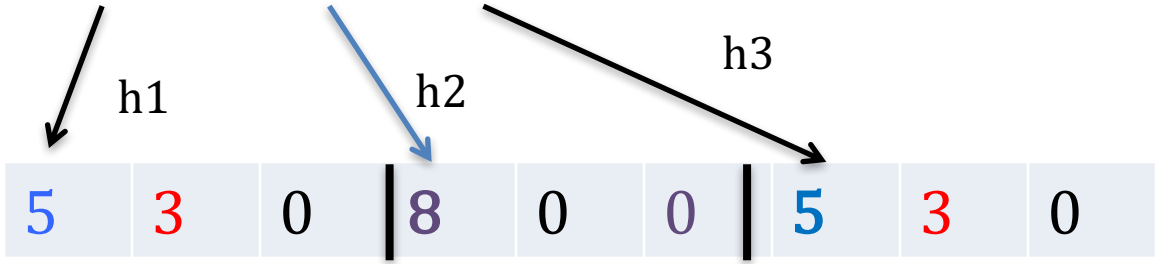
0	0	0	0	0	0	0	0	0

cm.get("fred flintstone"): 3

take min when retrieving a value



cm.get("barney rubble): 5

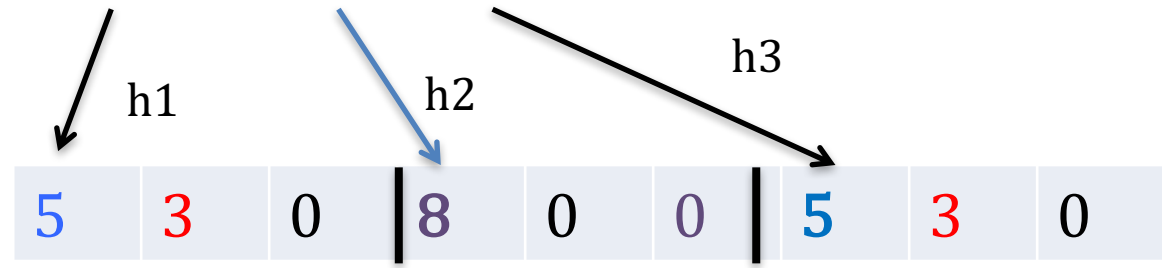


Count-min sketches

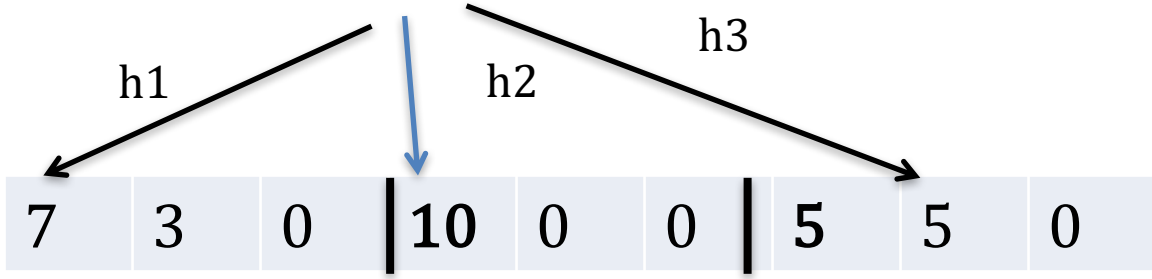
split a real vector into k ranges, one for each hash function

0	0	0	0	0	0	0	0	0

cm.get("barney rubble): 5



cm.add("pebbles", 2):



Count-min sketches

Equivalently, use a matrix, and each hash leads to a different row

cm.inc("fred flintstone", 3):

0	3	0
3	0	0
0	3	0

cm.inc("barney rubble",5):

5	3	0
8	0	0
5	3	0

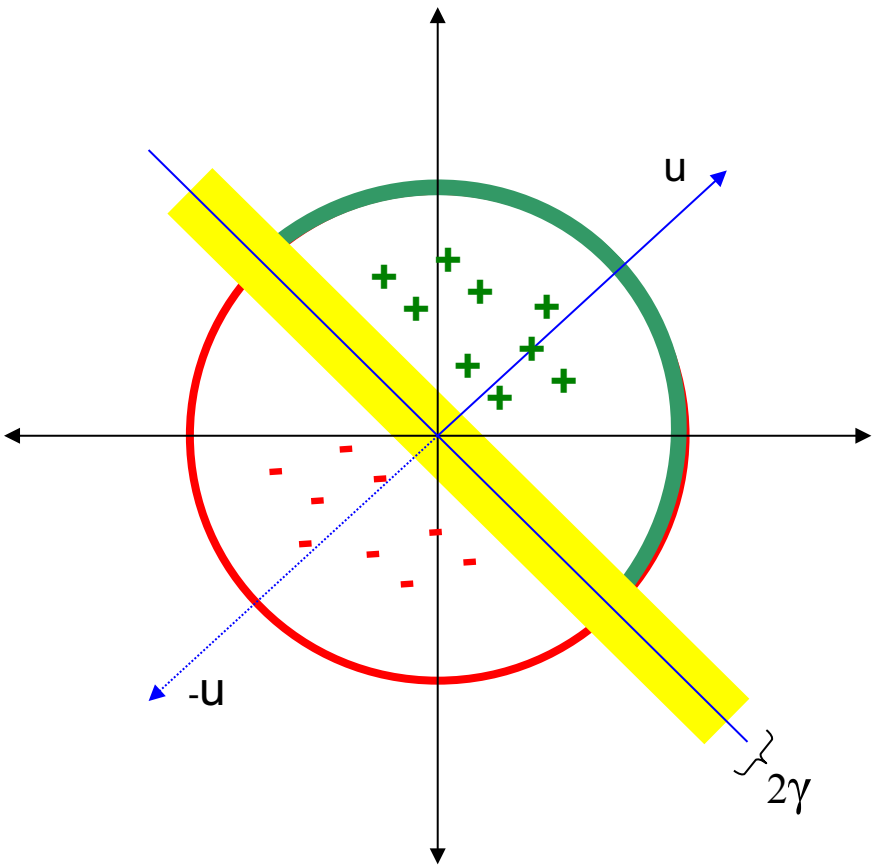
LOCALITY SENSITIVE HASHING (LSH)

LSH: key ideas

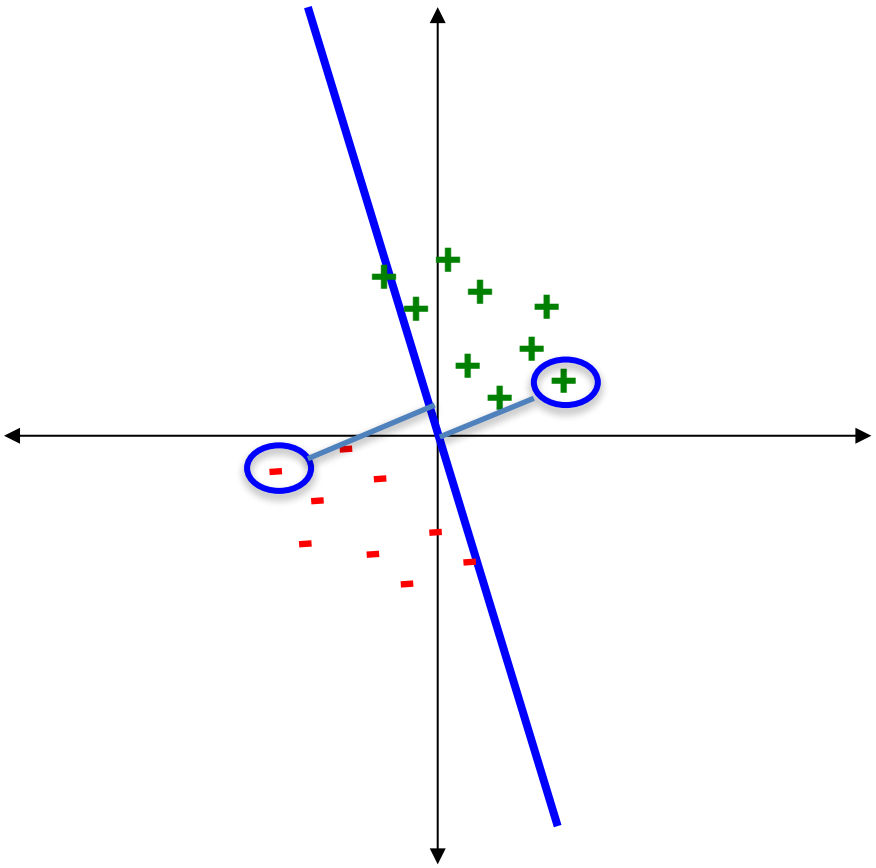
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"

Random Projections

Random projections

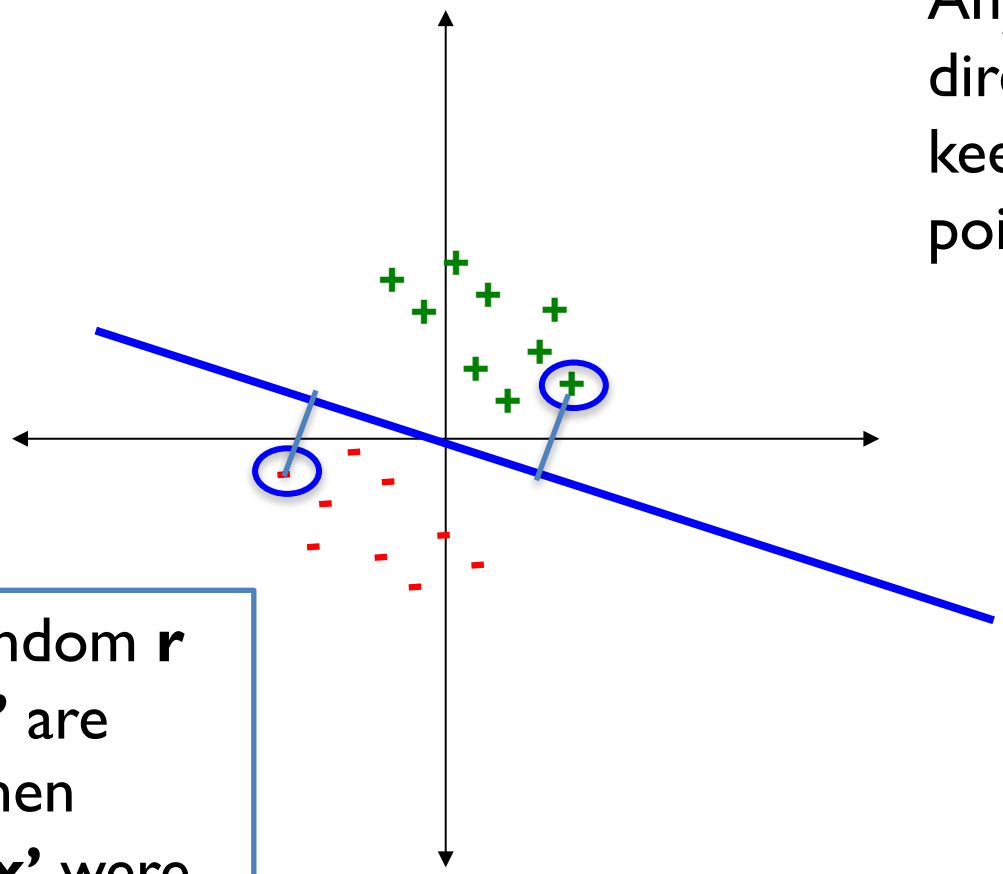


Random projections



To make those points "close" we need to project to a direction orthogonal to the line between them

Random projections

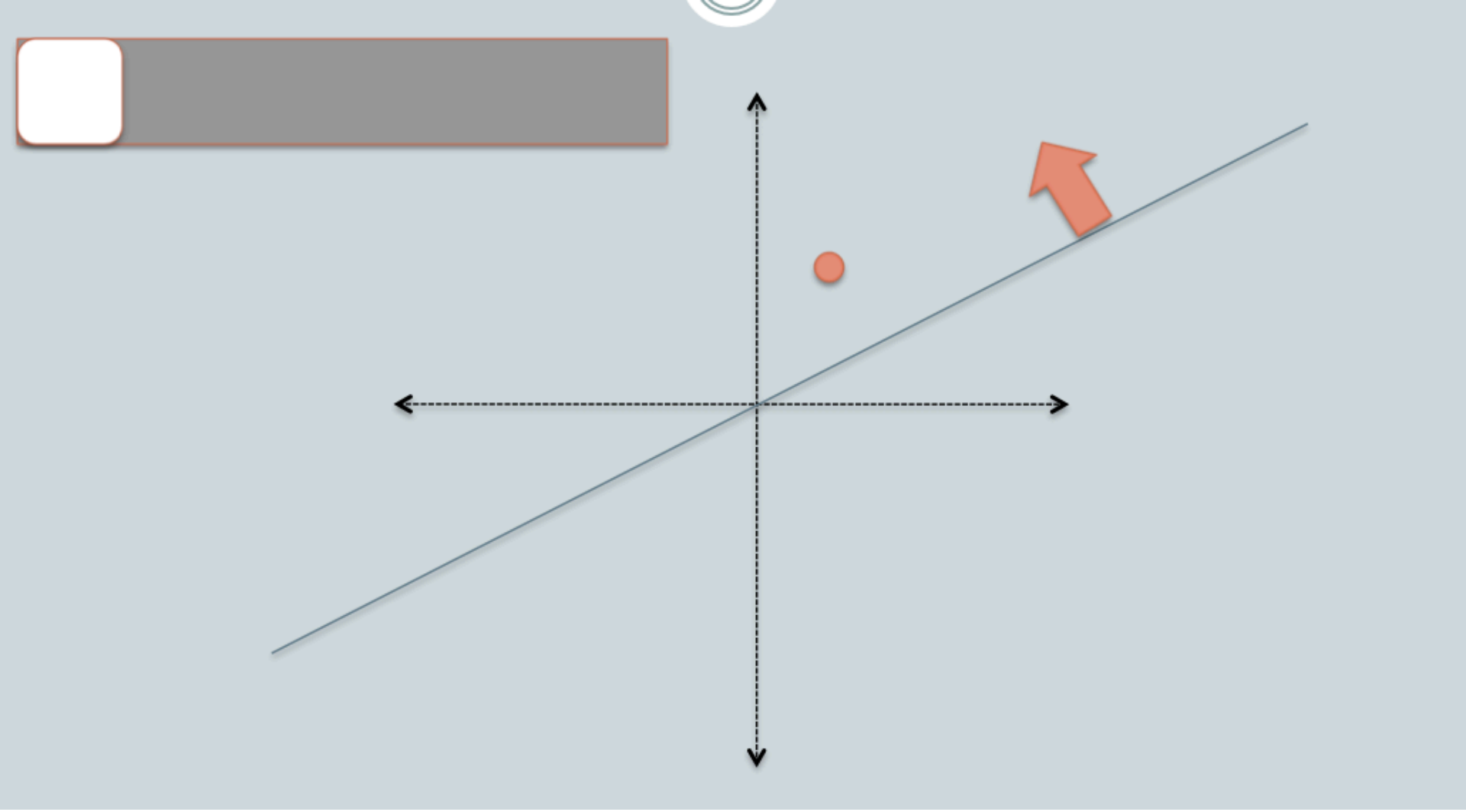


Any other direction will keep the distant points distant.

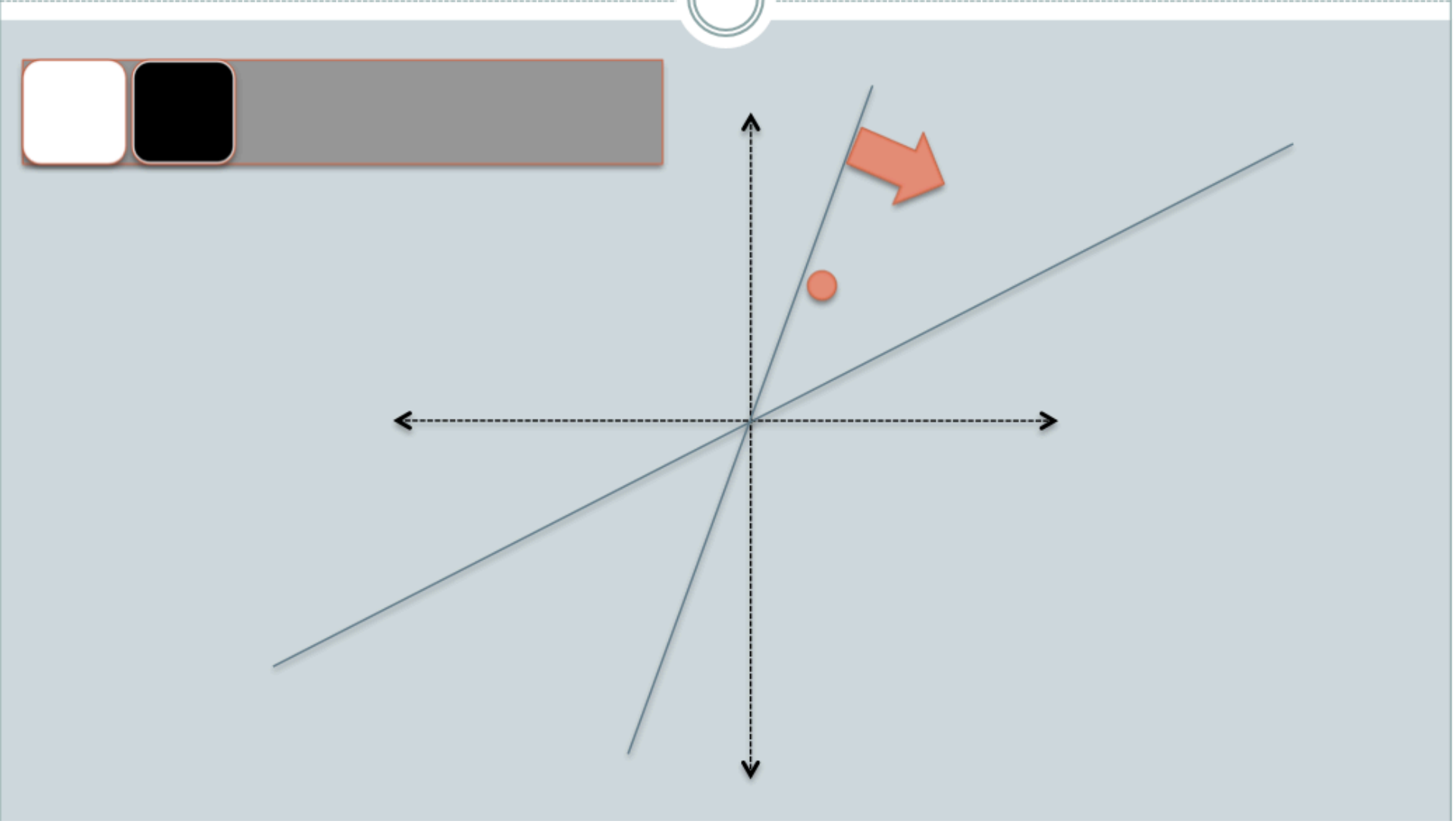
So if I pick a random **r** and **r.x** and **r.x'** are closer than γ then *probably* **x** and **x'** were close to start with.

LSH: key ideas

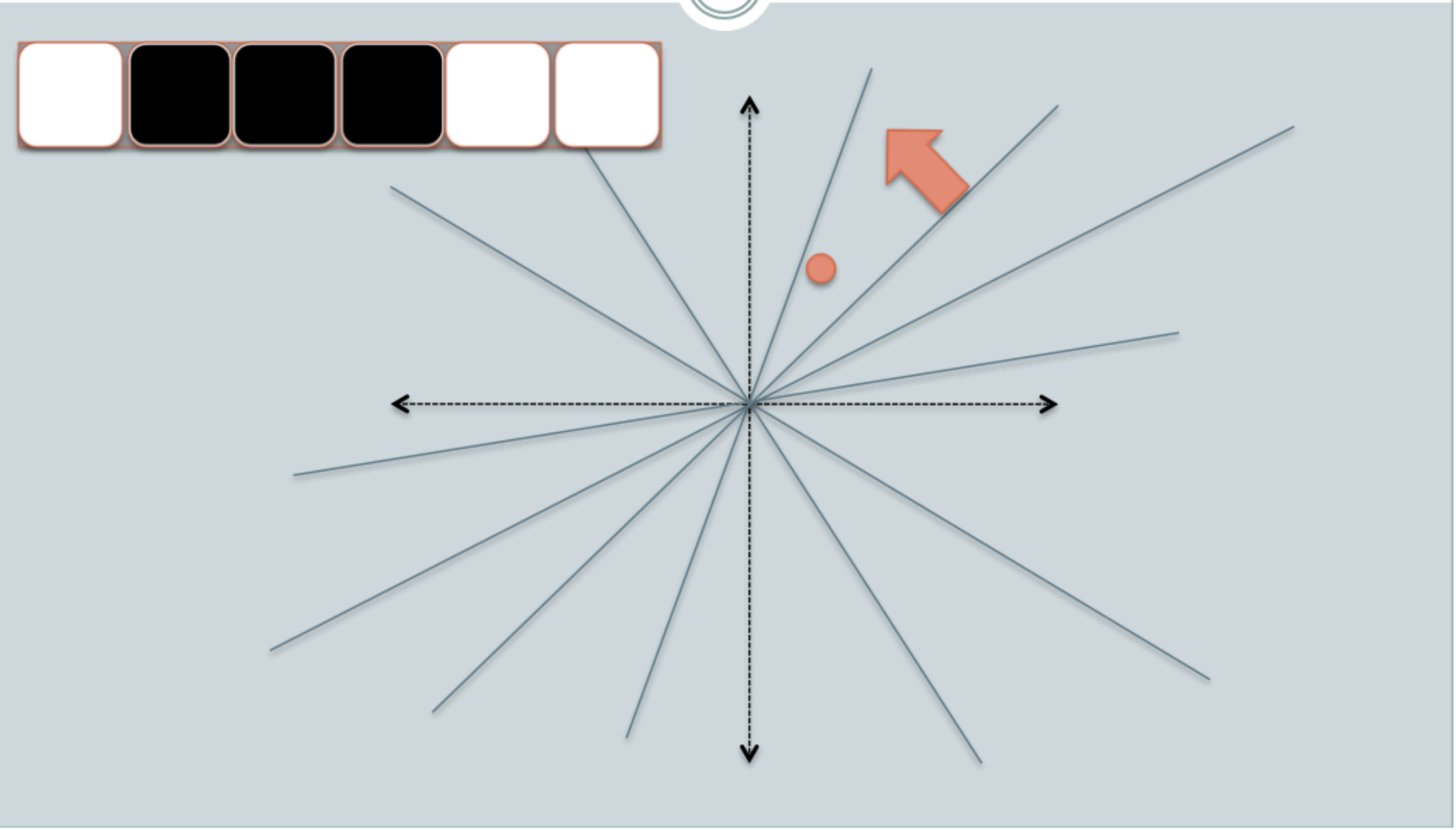
- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves "similarity"
- Basic idea: use *random projections* of **x**
  - Repeat many times:
    - Pick a random hyperplane **r** by picking random weights for each feature (say from a Gaussian)
    - Compute the inner product of **r** with **x**
    - Record if **x** is "close to" **r** (**r.x**>=0)
      - the next bit in **bx**
    - Theory says that is **x'** and **x** have small cosine distance then **bx** and **bx'** will have small Hamming distance



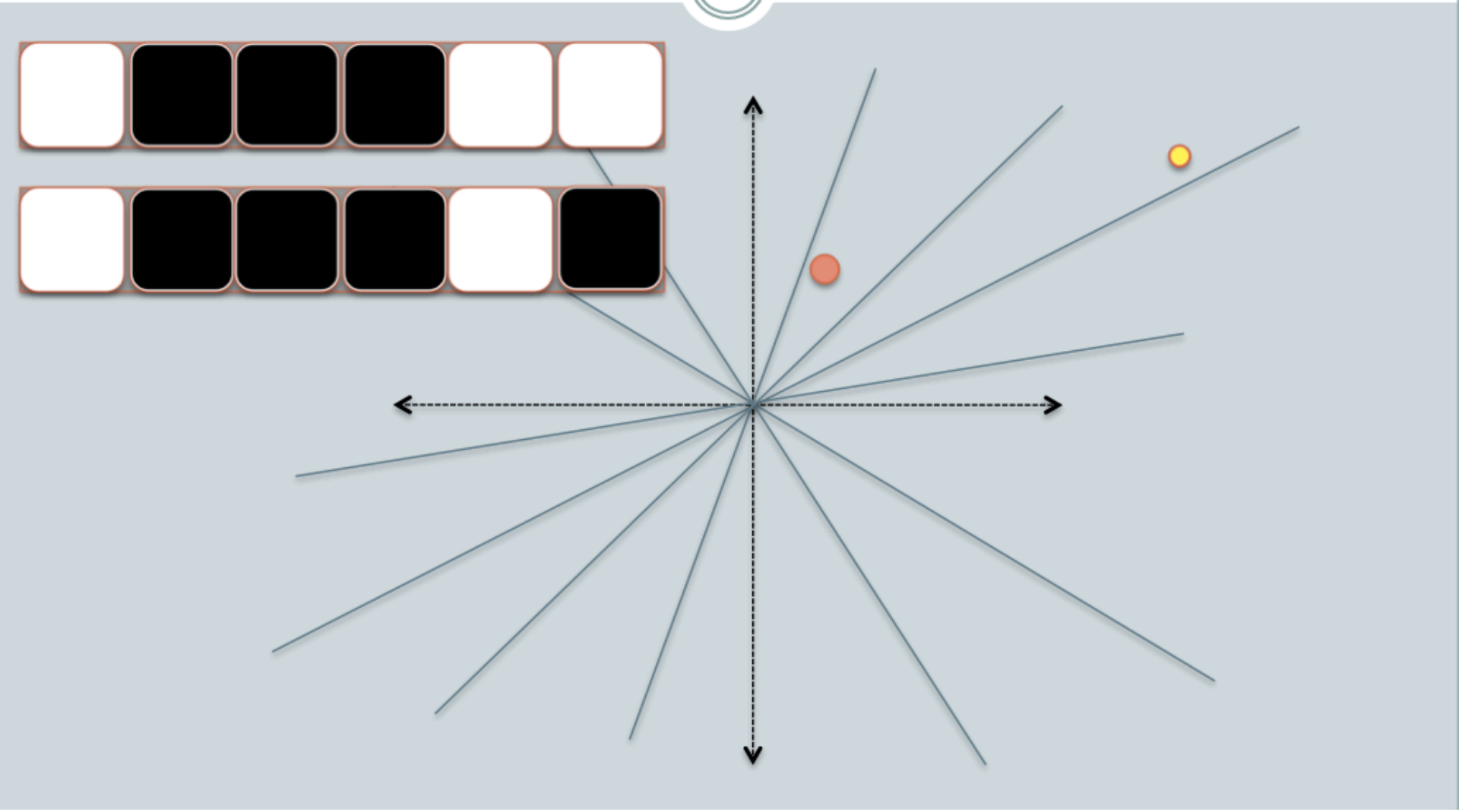
[Slides: Ben van Durme]



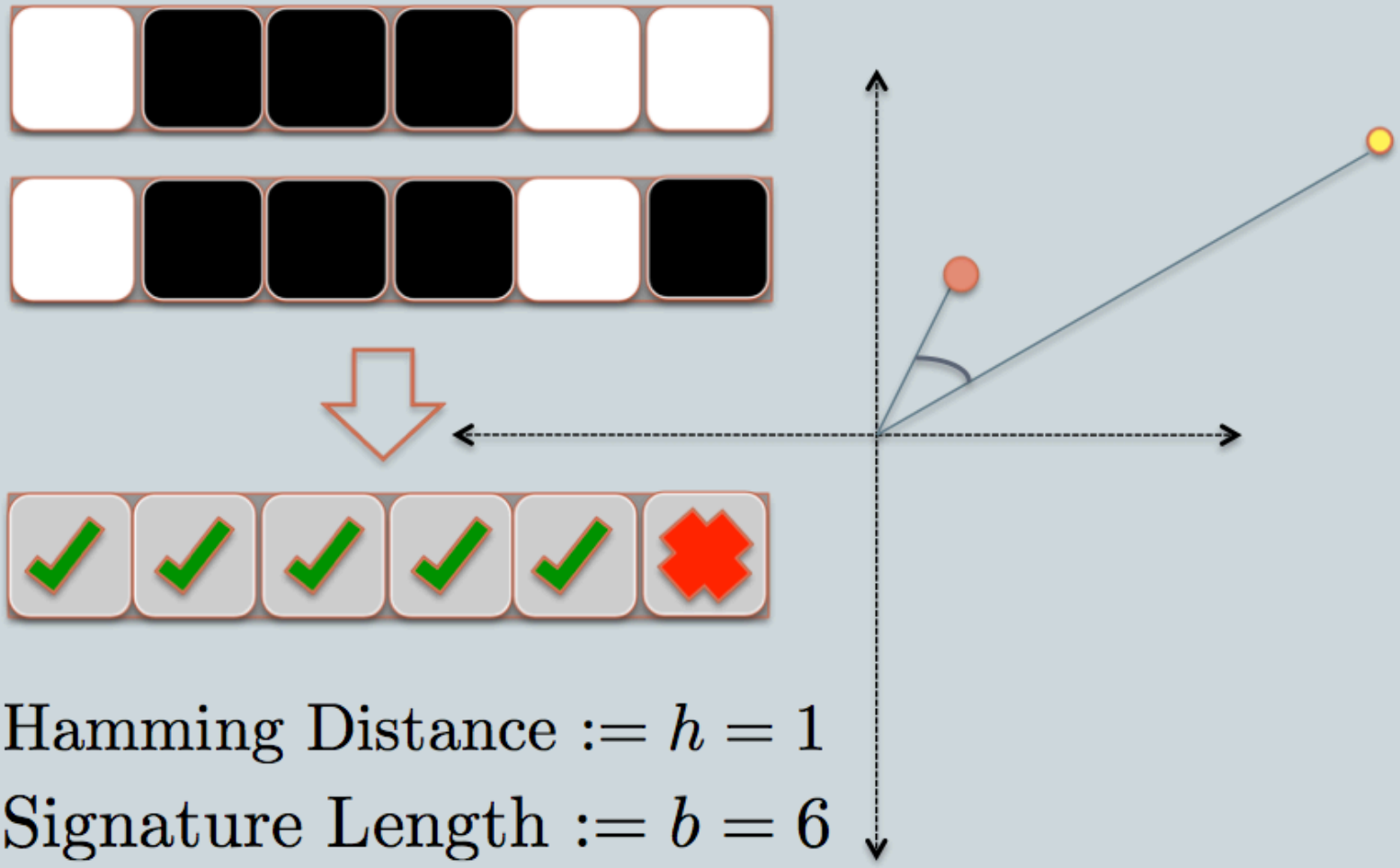
[Slides: Ben van Durme]



[Slides: Ben van Durme]



[Slides: Ben van Durme]



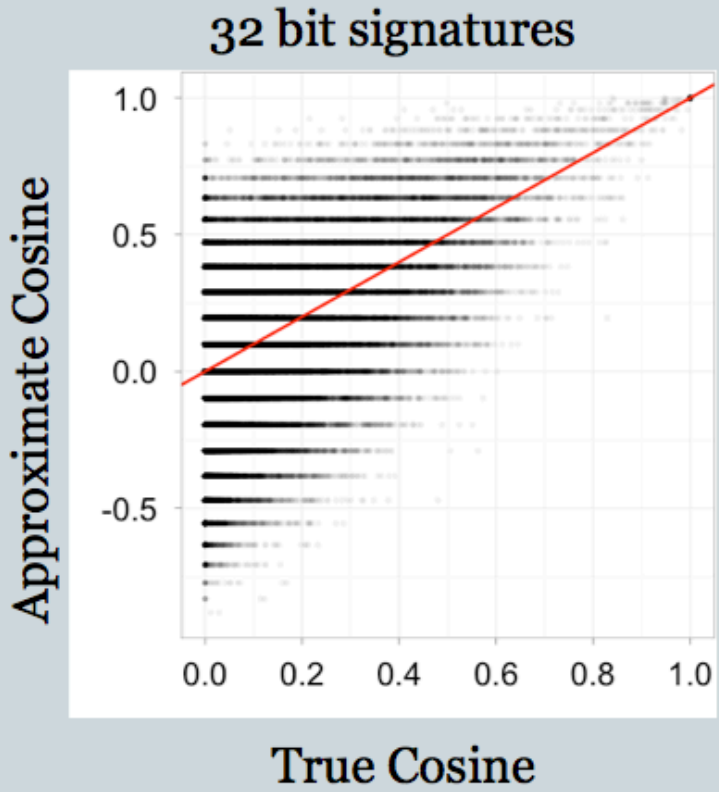
Hamming Distance $:= h = 1$

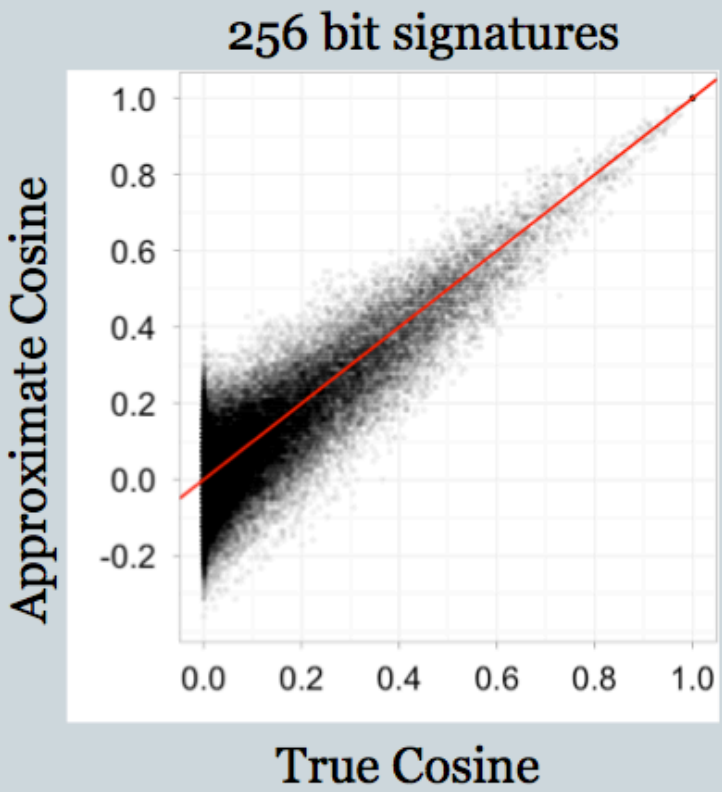
Signature Length $:= b = 6$

$$\cos(\theta) \approx \cos(\tfrac{h}{b}\pi)$$

$$= \cos(\tfrac{1}{6}\pi)$$

[Slides: Ben van Durme]





Cheap

Accurate

[Slides: Ben van Durme]

LSH applications

- Compact storage of data
  - and we can still compute similarities
- LSH also gives very fast approximations:
  - approx nearest neighbor method
    - just look at other items with **bx'=bx**
    - also very fast nearest-neighbor methods for Hamming distance
  - very fast clustering
    - cluster = all things with same **bx** vector

Online LSH and Pooling

Online Generation of Locality Sensitive Hash Signatures

Benjamin Van Durme and Ashwin Lall

human language technology
center of excellence
JOHNS HOPKINS
UNIVERSITY

DENISON
UNIVERSITY

LSH algorithm

- Naïve algorithm:
  - Initialization:
    - For i=1 to outputBits:
      - For each feature *f:*
        - » Draw r(f,i) ~ Normal(0,1)
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] =

      sum(**x**[*f*]*r[i,*f*] for *f* with non-zero weight in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH

LSH algorithm

- But: storing the *k classifiers* is expensive in high dimensions
  - For each of 256 bits, a dense vector of weights for every feature in the vocabulary
- Storing seeds and random number generators:
  - Possible but somewhat fragile

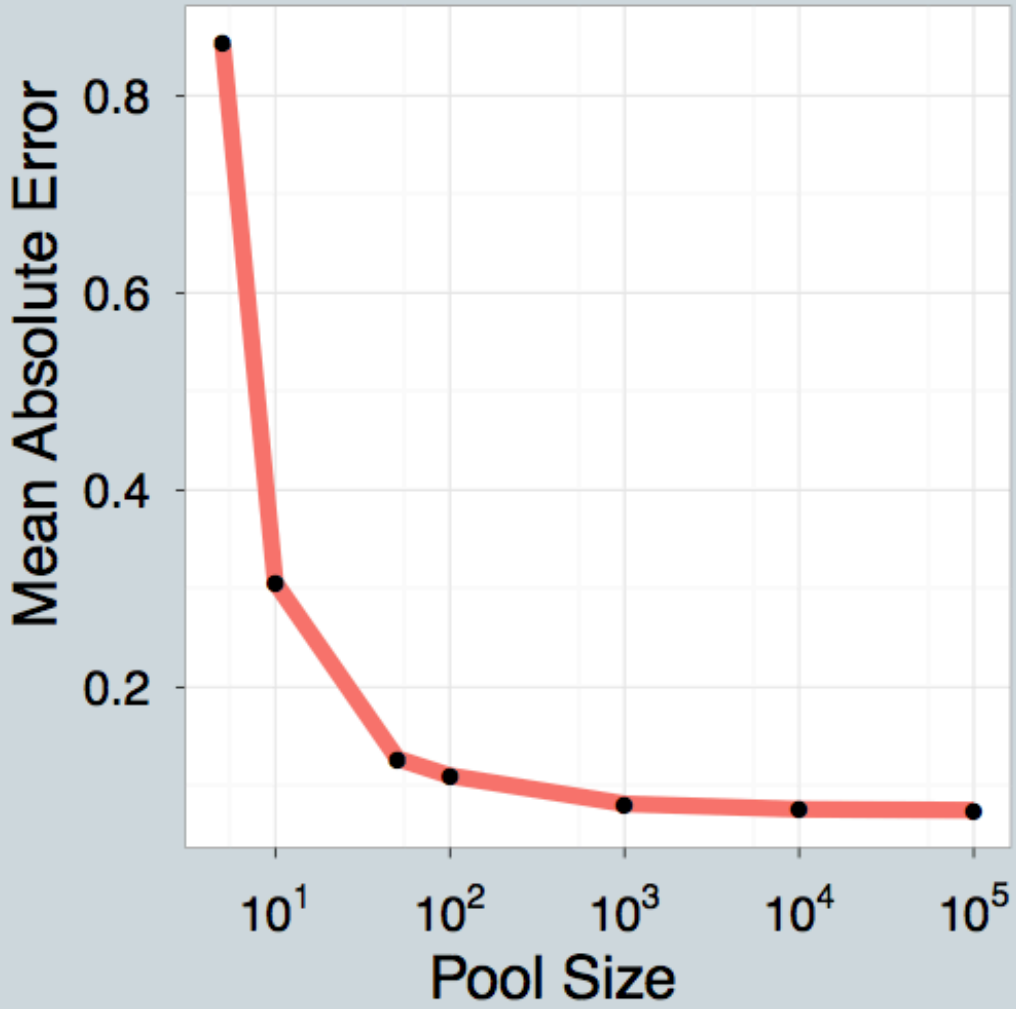
LSH: "pooling" (van Durme)

- Better algorithm:
  - Initialization:
    - Create a pool:
      - Pick a random seed *s*
      - For i=1 to poolSize:
        - » Draw pool[i] ~ Normal(0,1)
    - For i=1 to outputBits:
      - Devise a random hash function hash(i,*f*):
        - » E.g.: hash(i,f) = hashcode(f) XOR randomBitString[i]
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] = sum(

      x[f] * pool[hash(i,f) % poolSize] for *f* in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH

The Pooling Trick



LSH: key ideas: pooling

- Advantages:
  - with pooling, this is a compact re-encoding of the data
    - you don't need to store the **r**'s, just the pool

Locality Sensitive Hashing (LSH) in an On-line Setting

LSH: key ideas: online computation

- Common task: **distributional clustering**
  - for a word *w*, ***v****(w)* is sparse vector of words that co-occur with *w*
  - cluster the *v(w)'s*

…guards at Pentonville prison in North **London** discovered that an escape attempt…

An American Werewolf in **London** is to be remade by the son of the original director…

…UK pop up shop on Monmouth Street in **London** today and on Friday the brand…

v**(London):** Pentonville, prison, in, North, …. and, on Friday

LSH: key ideas: online computation

- Common task: distributional clustering
  - for a word *w*, ***v****(w)* is sparse vector of words that co-occur with *w*
  - cluster the *v(w)*'s

London is similar to:

Milan$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$

in is similar to:

during$_{.99}$, **on**$_{.98}$, **beneath**$_{.98}$, **from**$_{.98}$, **onto**$_{.97}$

sold is similar to:

deployed$_{.84}$, **presented**$_{.83}$, **sacrificed**$_{.82}$, **held**$_{.82}$, **installed**$_{.82}$

LSH: key ideas: online computation

- Common task: distributional clustering
  - for a word *w*, ***v****(w)* is **sparse** vector of words that co-occur with *w*
  - cluster the *v(w)*'s

v(w) is very similar to a word embedding (eg, from word2vec or GloVE)

Levy, Omer, Yoav Goldberg, and Ido Dagan. "Improving distributional similarity with lessons learned from word embeddings." *Transactions of the Association for Computational Linguistics* 3 (2015): 211-225.

$\vec{v} \in \mathbb{R}^d$ *v is context vector; d is vocab size*

$\vec{r}_i \sim N(0,1)^d$ *r_i is i-th random projection*

…guards at Pentonville prison in North **London** c

An American Werewolf in **London** is to be remad

…UK pop up shop on Monmouth Street in **Londo**

v**(London):** Pentonville, prison, in, North, …. and

$$h_i(\vec{v}) = \begin{cases} 1 & \text{if } \vec{v} \cdot \vec{r}_i \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

$h_i(v)$ i-th bit of LSH encoding

$$\text{if } \vec{v} = \Sigma_j \vec{v}_j$$

because context vector is sum of *mention contexts*

$$\text{then } \vec{v} \cdot \vec{r}_i = \Sigma_j \vec{v}_j \cdot \vec{r}_i$$

Break into local products

these come one by one as we stream thru the corpus

Online

$$h_{it}(\vec{v}) = \begin{cases} 1 & \text{if } \Sigma_j^t \vec{v}_j \cdot \vec{r}_i \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Algorithm 1 STREAMING LSH ALGORITHM

Parameters:

m : size of pool

d : number of bits (size of resultant signature)

s : a random seed

$h_1, ..., h_d$: hash functions mapping $\langle s, f_i \rangle$ to $\{0, \ldots, m-1\}$

INITIALIZATION:

```
Initialize floating point array P[0, ..., m − 1]
Initialize H, a hashtable mapping words to floating point
    arrays of size d
for i := 0 ... m − 1 do
    P[i] := random sample from N(0, 1), using s as seed
```

ONLINE:

```
for each word w in the stream do
    for each feature f_i associated with w do
        for j := 1 ... d do
            H[w][j] := H[w][j] + P[h_j(s, f_i)]
```

weight of f_i in r_j

j-th hash of f_i

SIGNATURECOMPUTATION:

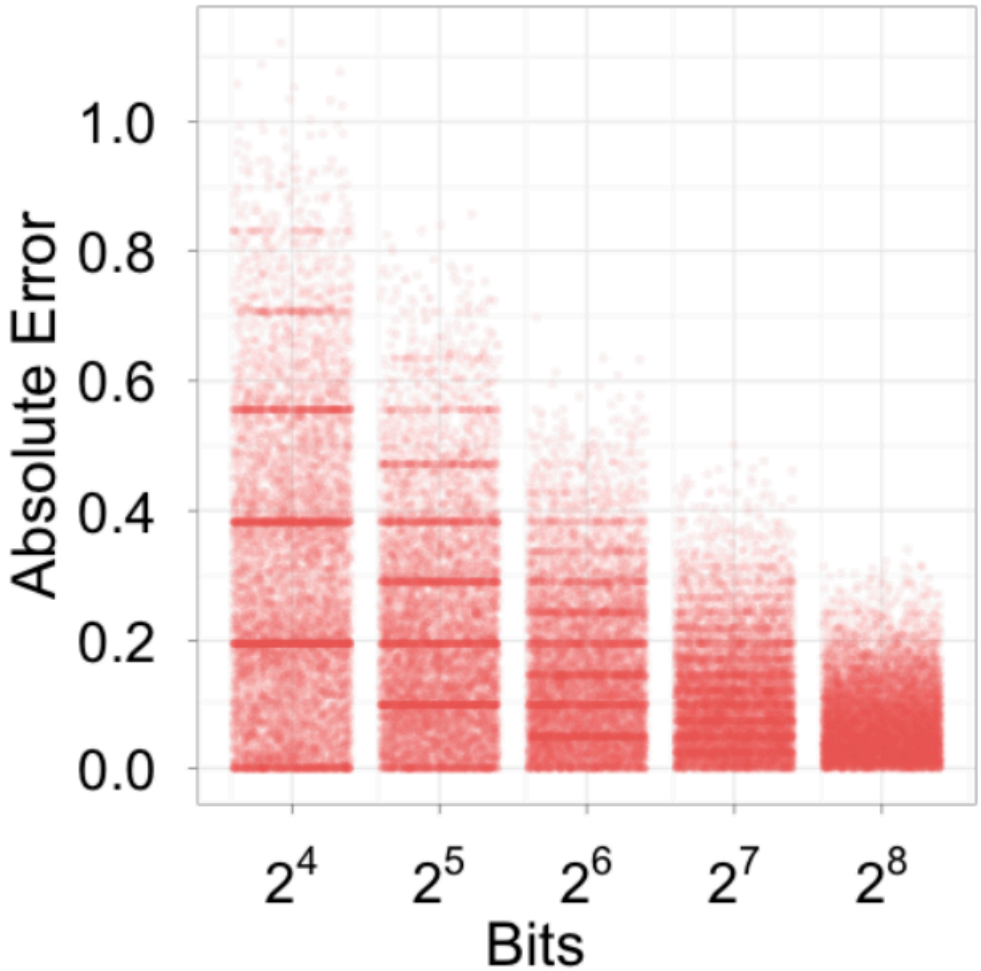
```
for each w ∈ H do
    for i := 1 ... d do
        if H[w][i] > 0 then
            S[w][i] := 1
        else
            S[w][i] := 0
```

$$\sum_j^t \vec{v}_j \cdot \vec{r}_i$$

Experiment

- Corpus: 700M+ tokens, 1.1M distinct bigrams
- For each, build a feature vector of words that co-occur near it, using on-line LSH
- Check results with 50,000 actual vectors

Experiment



Closest based on true cosine

	London
	Milan$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$
16	ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
32	Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$
64	Frankfurt$_4$, Prague$_4$, Taszar$_5$, Brussels$_6$, Copenhagen$_6$
128	Prague$_{12}$, Stockholm$_{12}$, Frankfurt$_{14}$, Madrid$_{14}$, Manila$_{14}$
256	Stockholm$_{20}$, Milan$_{22}$, Madrid$_{24}$, Taipei$_{24}$, Frankfurt$_{25}$

Closest based on true cosine

London

Milan$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$
ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$

London

Milan$_{.97}$, **Madrid**$_{.96}$, **Stockholm**$_{.96}$, **Manila**$_{.95}$, **Moscow**$_{.95}$
ASHER$_0$, Champaign$_0$, MANS$_0$, NOBLE$_0$, come$_0$
Prague$_1$, Vienna$_1$, suburban$_1$, synchronism$_1$, Copenhagen$_2$
Frankfurt$_4$, Prague$_4$, Taszar$_5$, Brussels$_6$, Copenhagen$_6$
Prague$_{12}$, Stockholm$_{12}$, Frankfurt$_{14}$, Madrid$_{14}$, Manila$_{14}$
Stockholm$_{20}$, Milan$_{22}$, Madrid$_{24}$, Taipei$_{24}$, Frankfurt$_{25}$

Closest based on 32 bit sig.'s

Cheap

Points to review

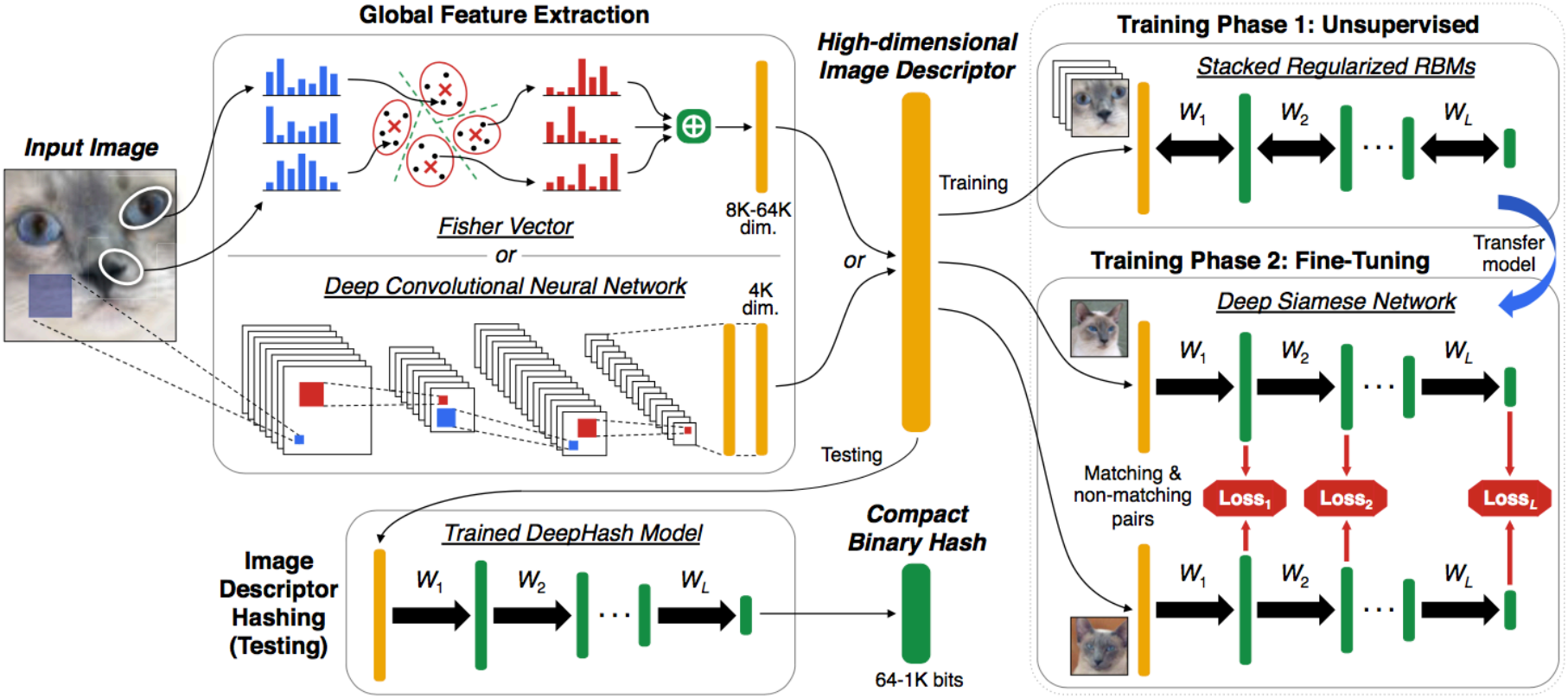
- APIs for:
  - Bloom filters, CM sketch, LSH
- Key applications of:
  - Very compact noisy sets
  - Efficient counters accurate for *large* counts
  - Fast approximate cosine distance
- Key ideas:
  - Uses of hashing that allow collisions
  - Random projection
  - Multiple hashes to control Pr(collision)
  - Pooling to compress a lot of random draws

A DEEP-LEARNING VARIANT OF LSH

DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin*[,1,3], Olivier Morère*[,1,2,3], Vijay Chandrasekhar[1,3], Antoine Veillard[2,3], Hanlin Goh[1,3]
I2R[1], UPMC[2], IPAL[3]

ICMR 2017

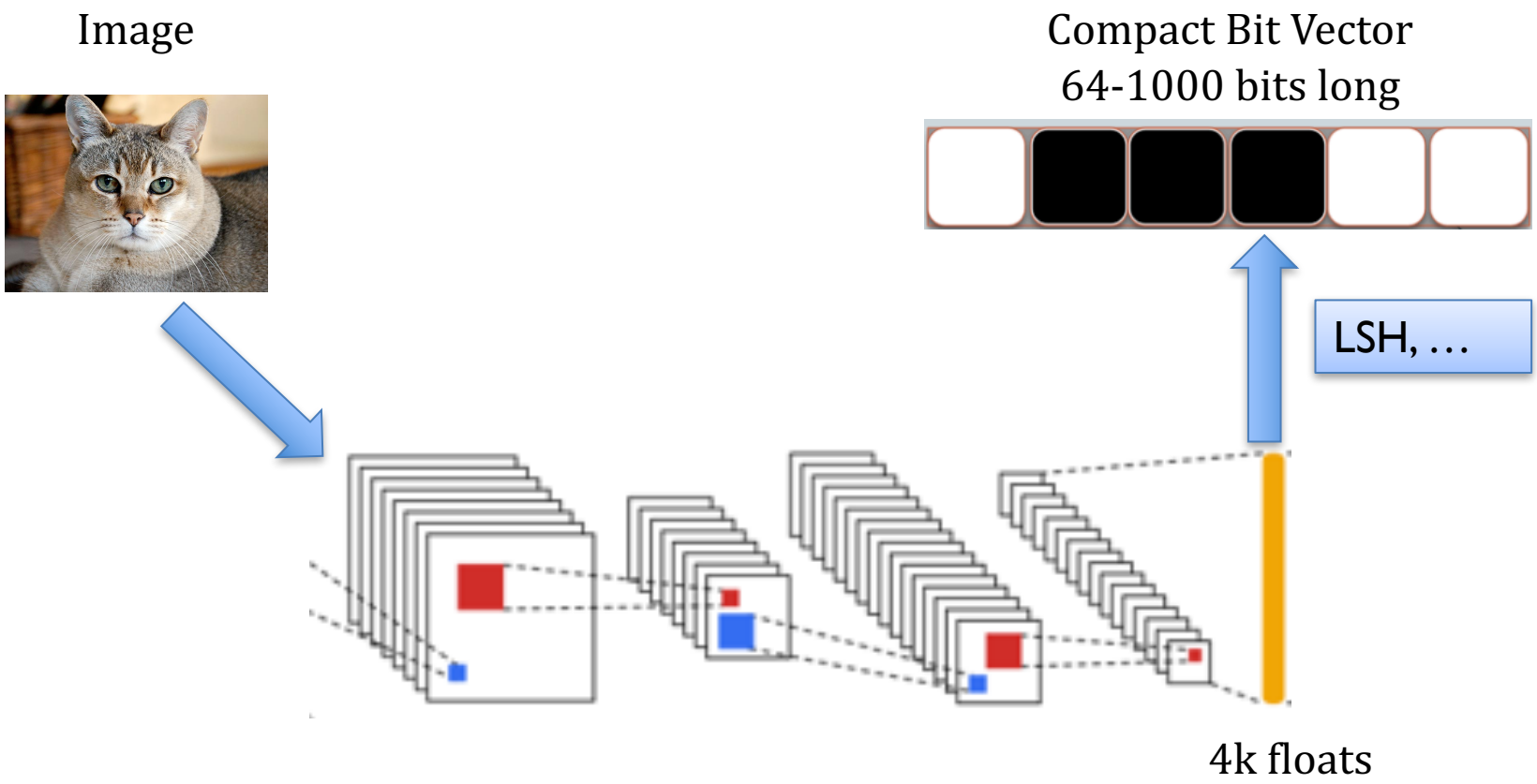


DeepHash

Image

Compact Bit Vector
64-1000 bits long

DeepHash



DeepHash

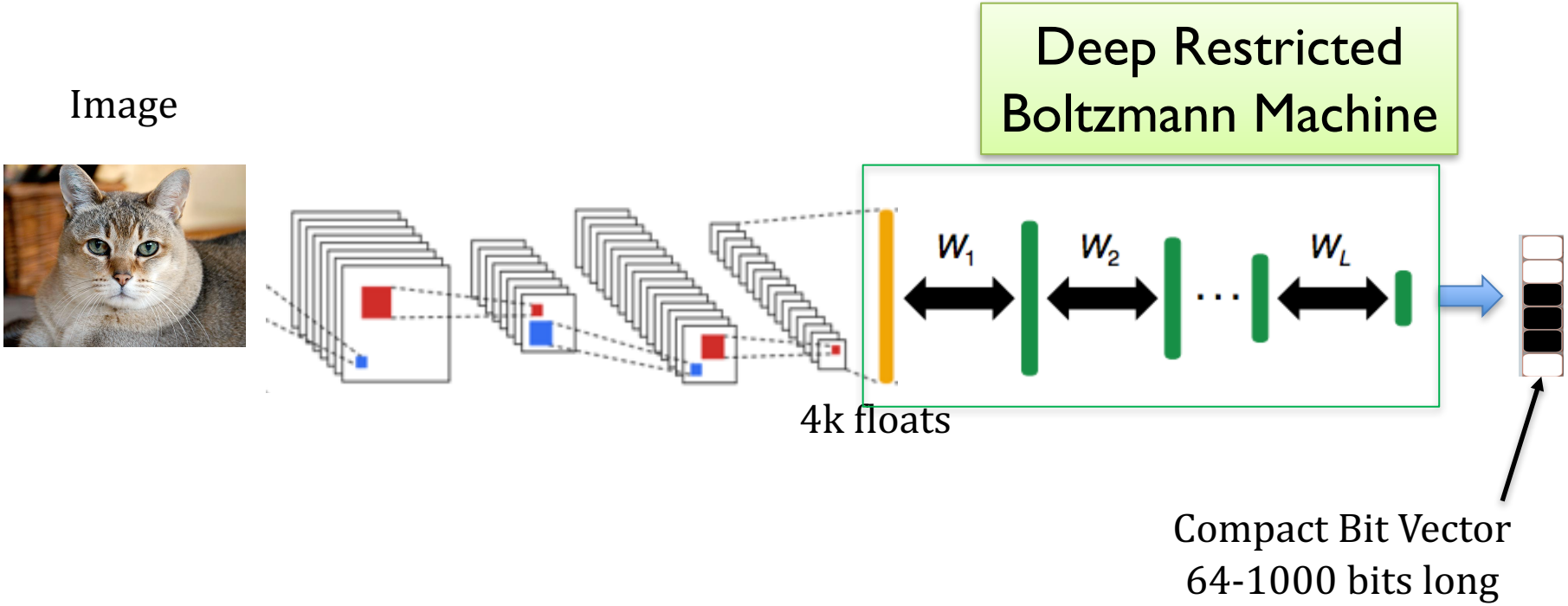
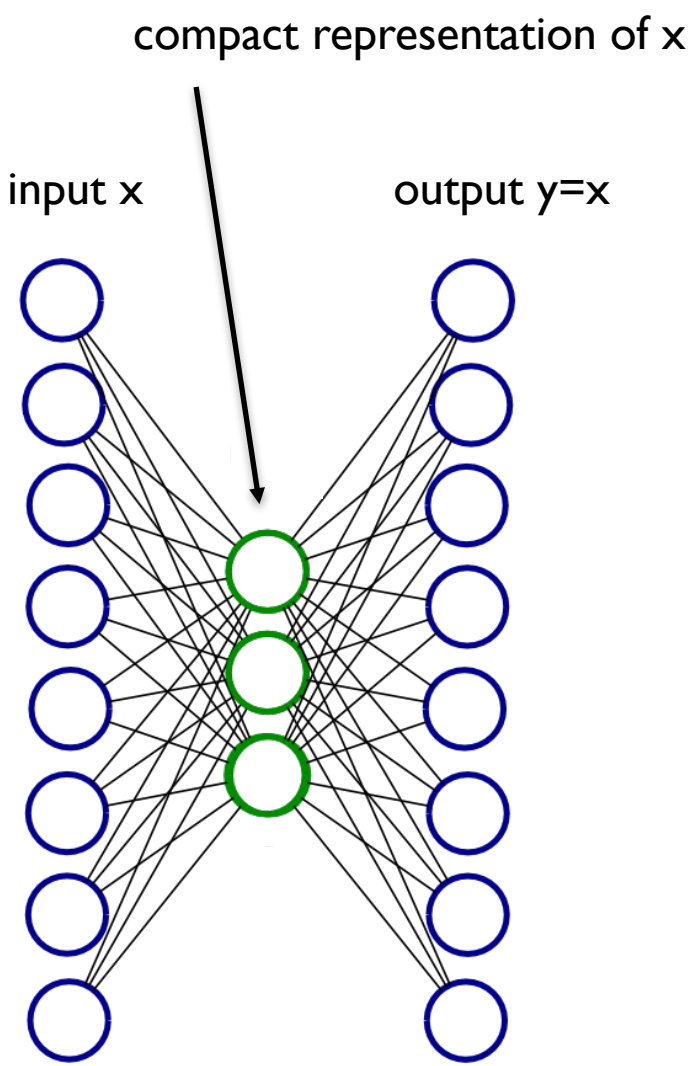


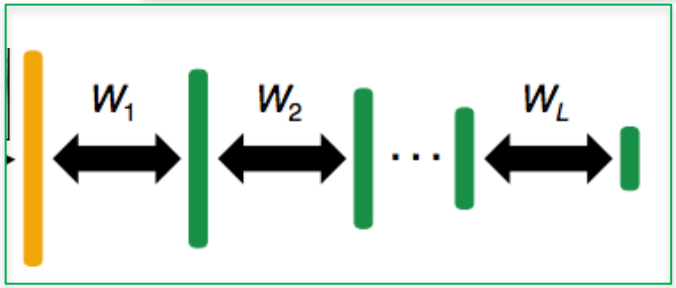
Image
Deep Restricted Boltzmann Machine
W_1
W_2
W_L
4k floats
Compact Bit Vector
64-1000 bits long

DeepHash



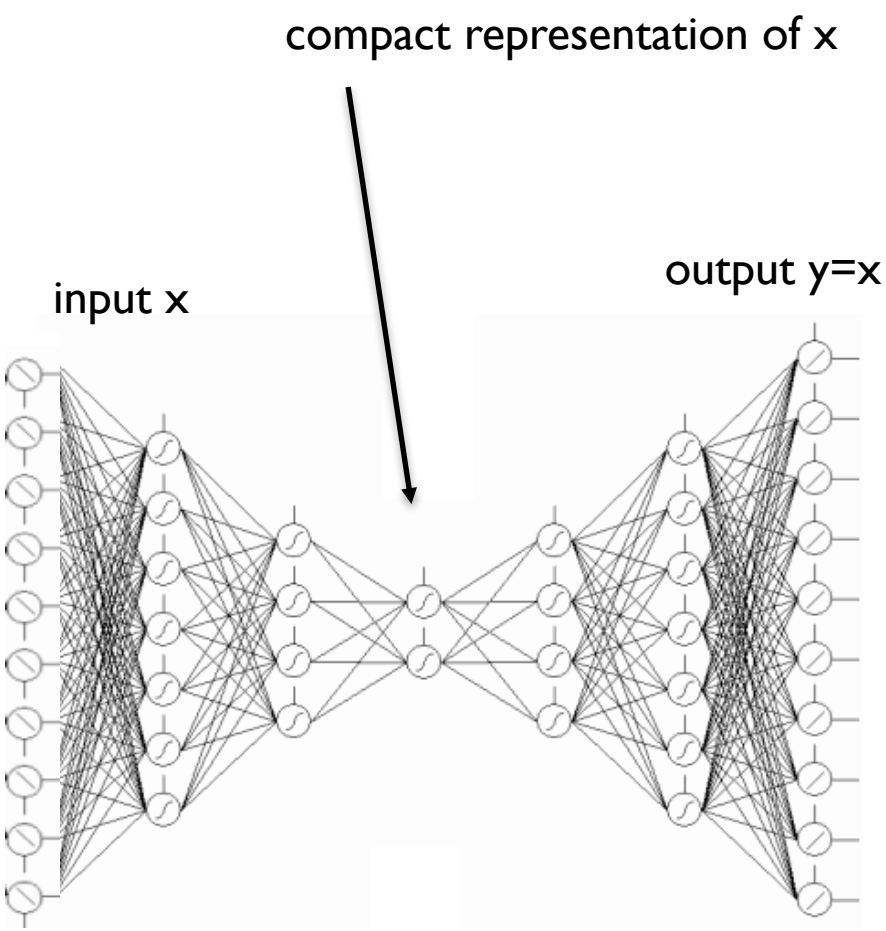
An autoencoder

Deep Restricted Boltzmann Machine



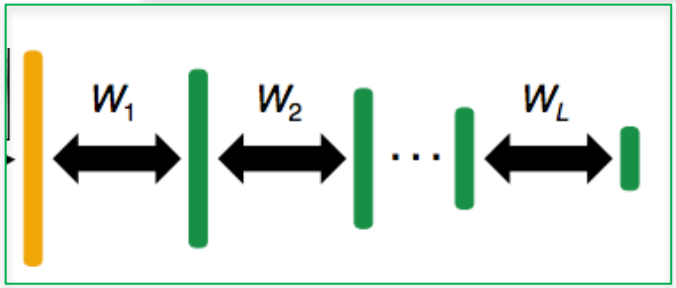
Restricted Boltzmann Machine is closely related to an autoencoder

DeepHash



A deeper autoencoder

Deep Restricted Boltzmann Machine



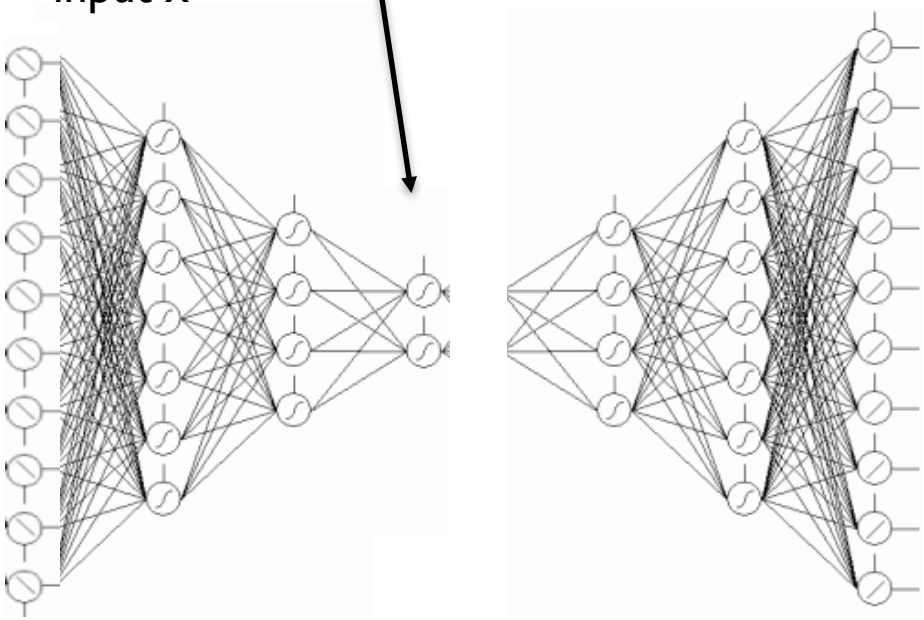
Deep Restricted Boltzmann Machine is closely related to a deep autoencoder

DeepHash

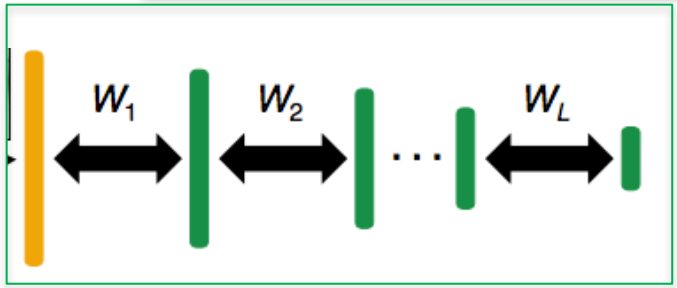
compact representation of x

input x

output y=x



Deep Restricted Boltzmann Machine



Deep Restricted Boltzmann Machine is

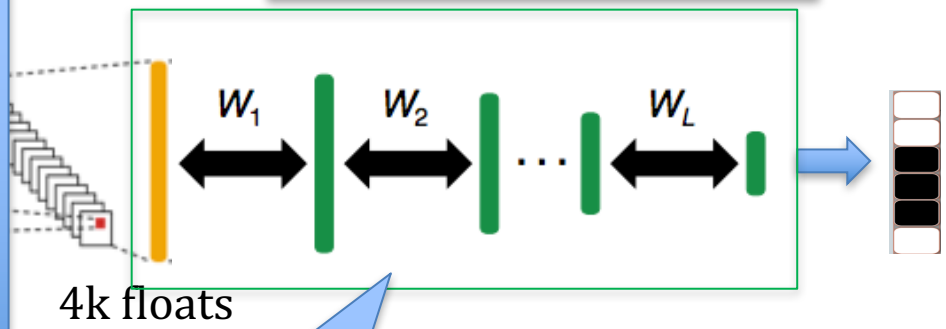
But the RBM is symmetric: weights W from layer *j* to *j+1* are the transpose of weights from *j+1* back to *j*

RBM is also stochastic: compute Pr(hidden|visible), **sample** from that distribution, compute Pr(visible|hidden), **sample**, ...

DeepHash

Image

Deep Restricted Boltzmann Machine

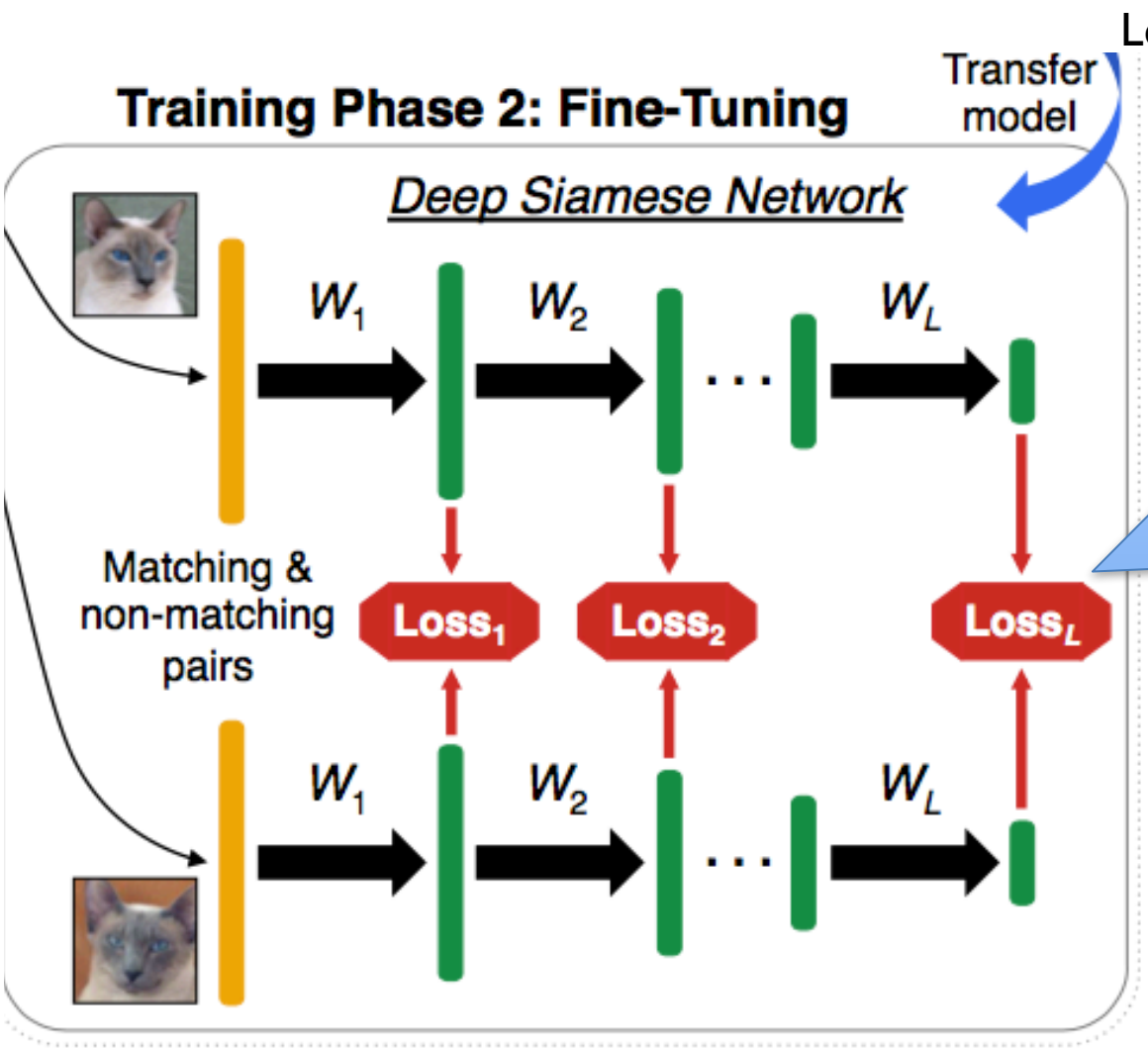


Another trick: regularize so that the representations are dense (about 50-50 "on" and "off" for an image) and each bit has 50-50 chance of being "on"

This model is trained to compress image features, then reconstruct the image features from the reconstructions.

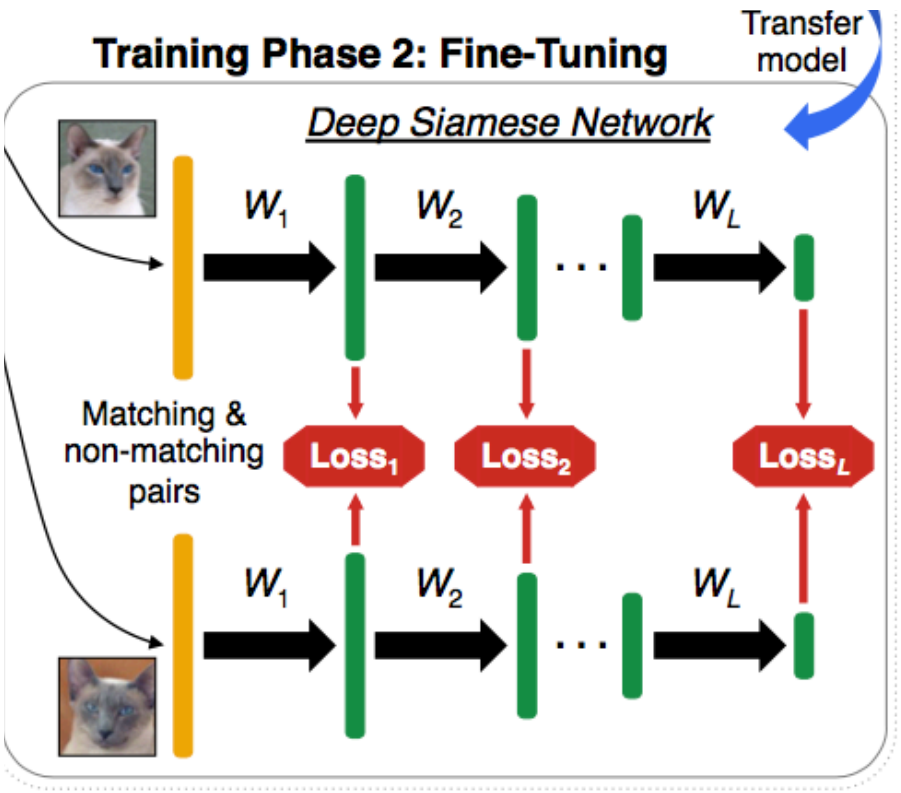
And then more training….

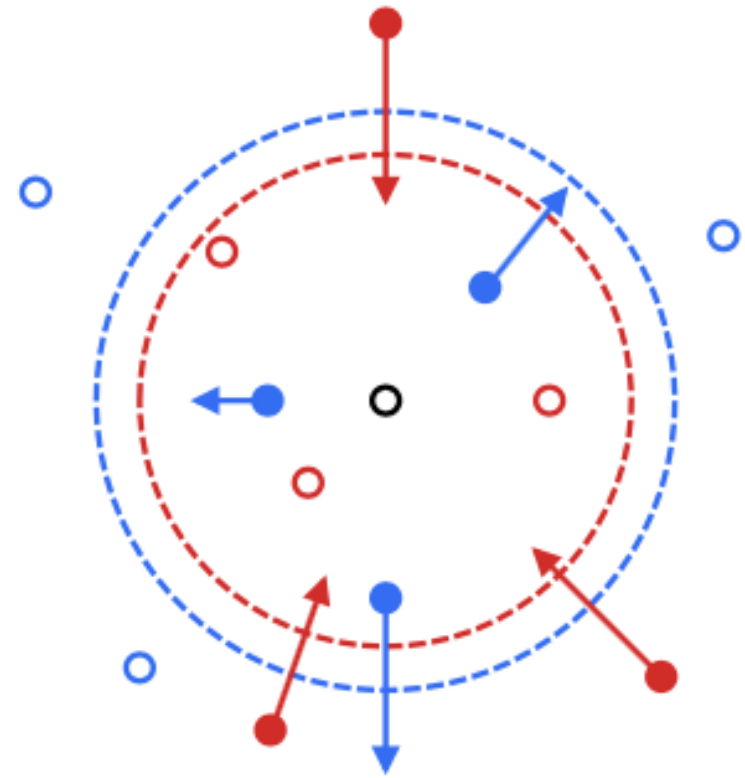
Training on matching vs non-matching pairs



Loss pushes the representation for "matching" pairs together and representation for non-matching pairs apart

Training on matching vs non-matching pairs

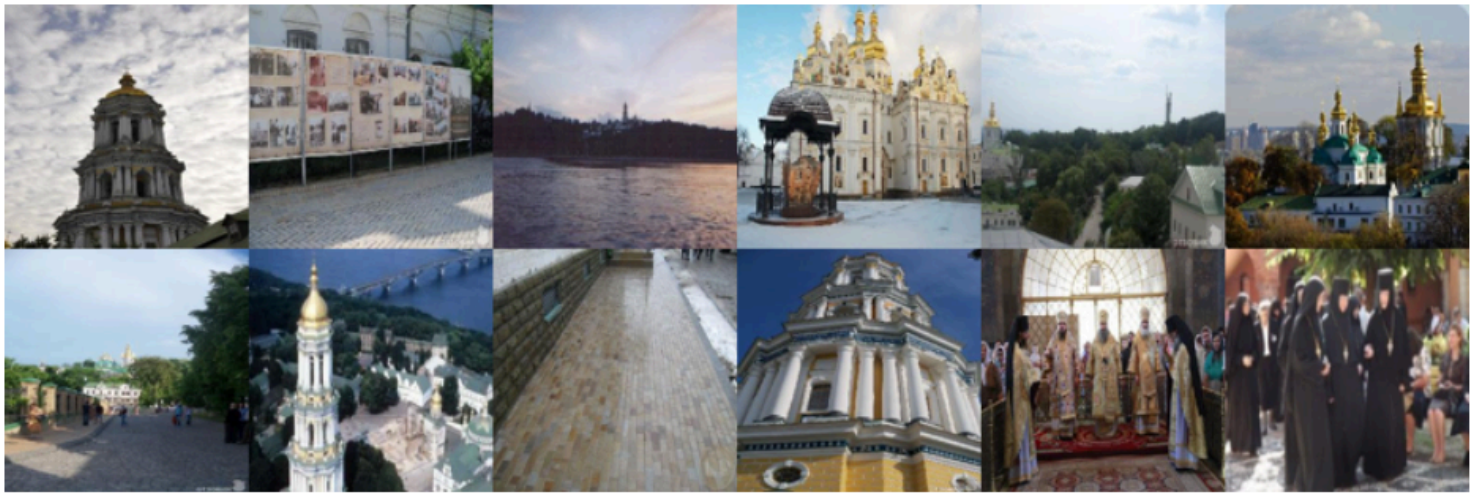




margin-based matching loss

Training on matching vs non-matching pairs

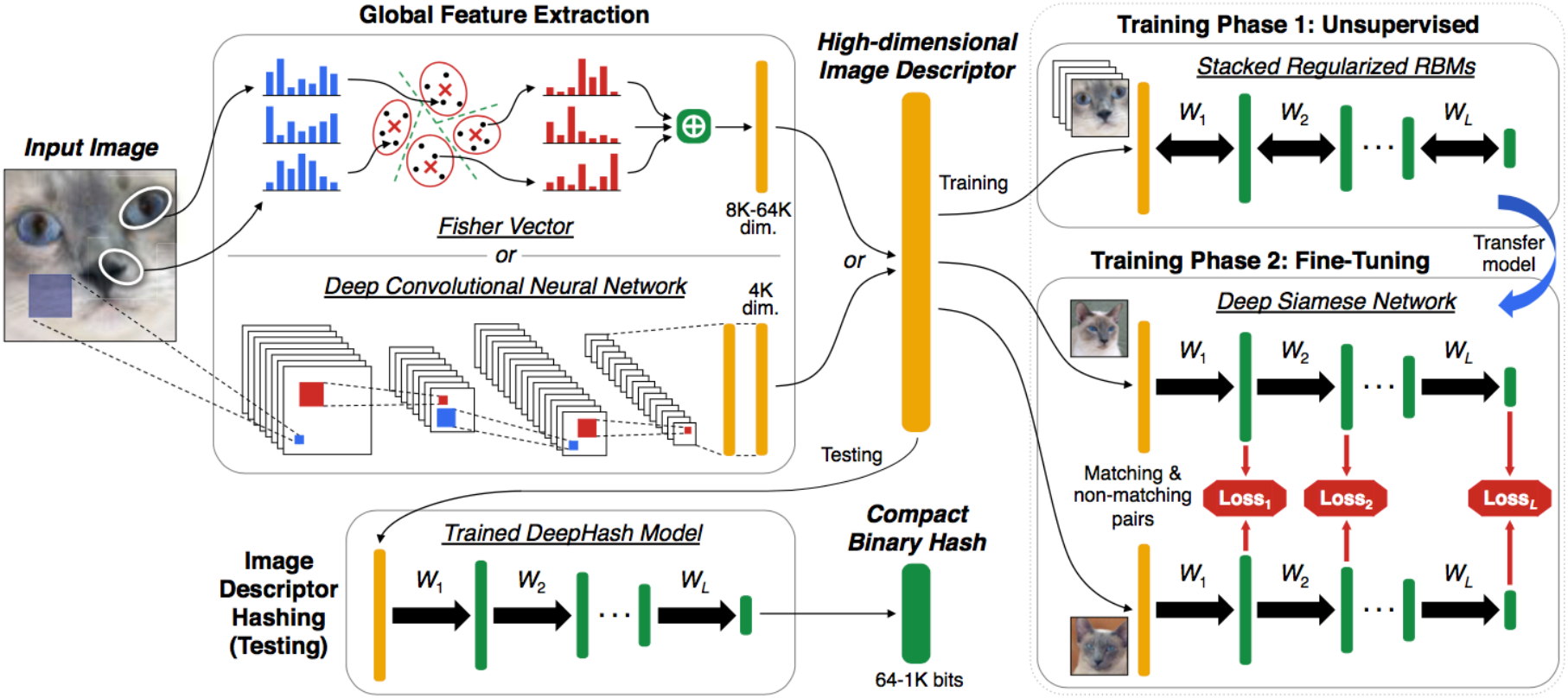
matching pairs were photos of the same "landmark"



DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin*[,1,3], Olivier Morère*[,1,2,3], Vijay Chandrasekhar[1,3], Antoine Veillard[2,3], Hanlin Goh[1,3]
I2R[1], UPMC[2], IPAL[3]

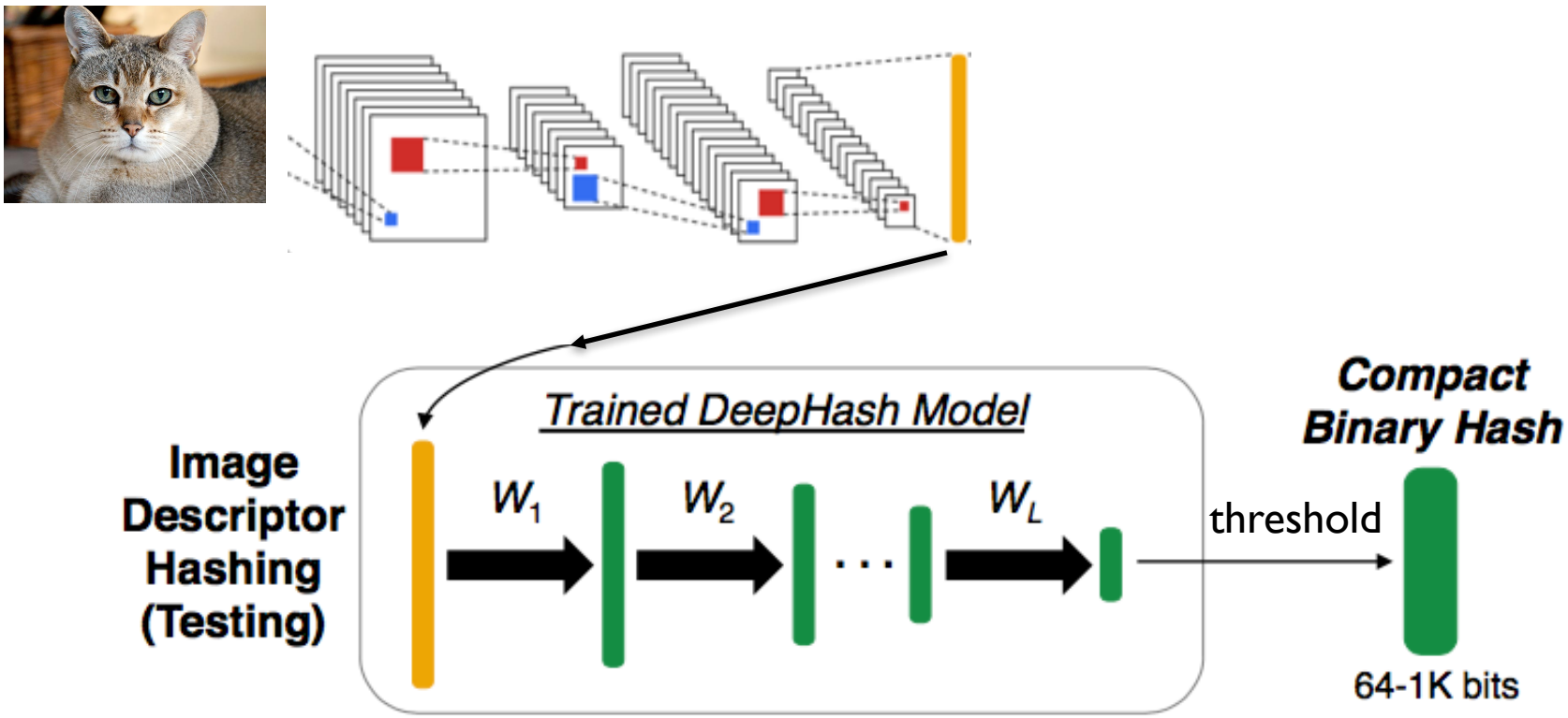
ICML 2017

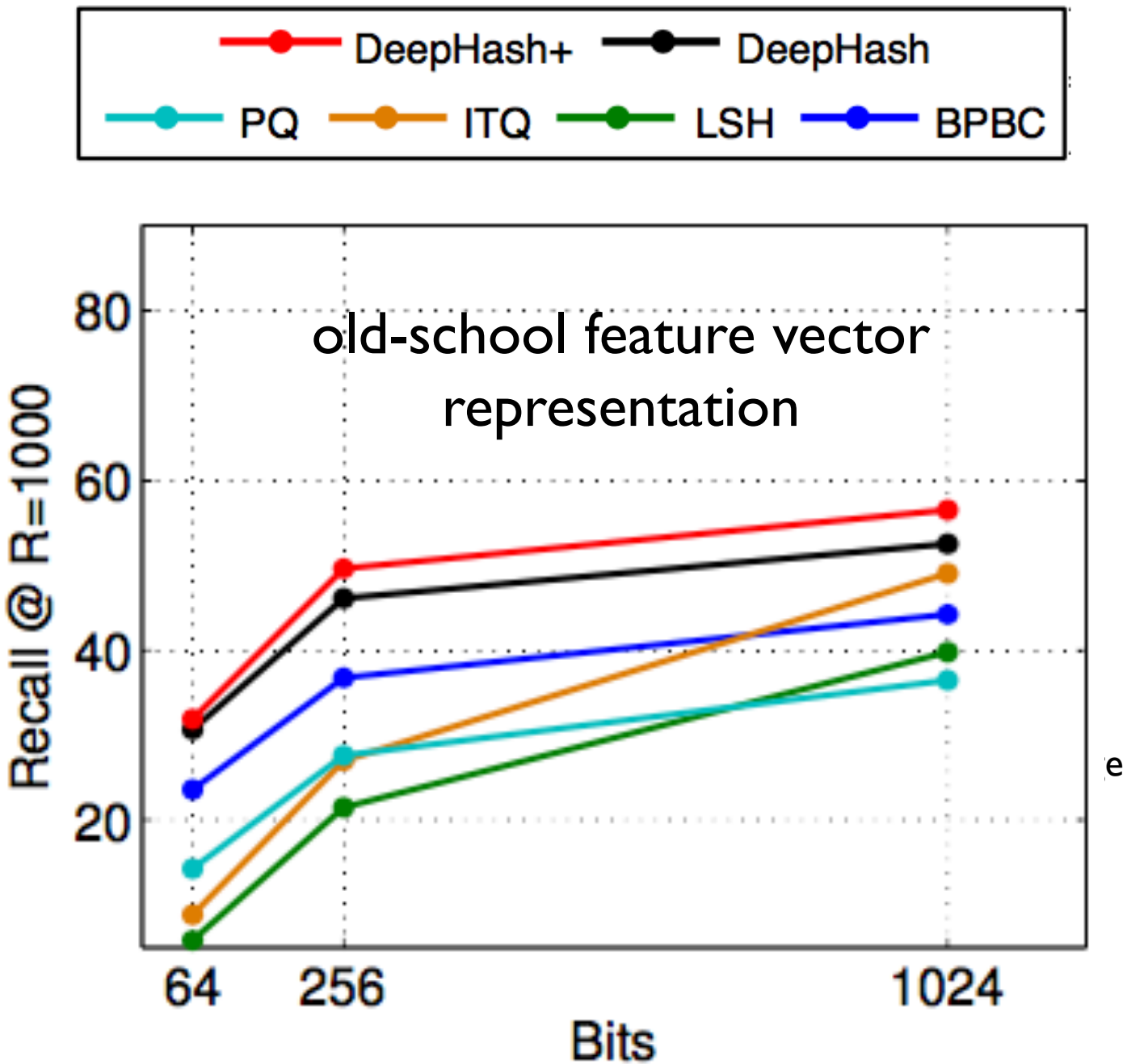


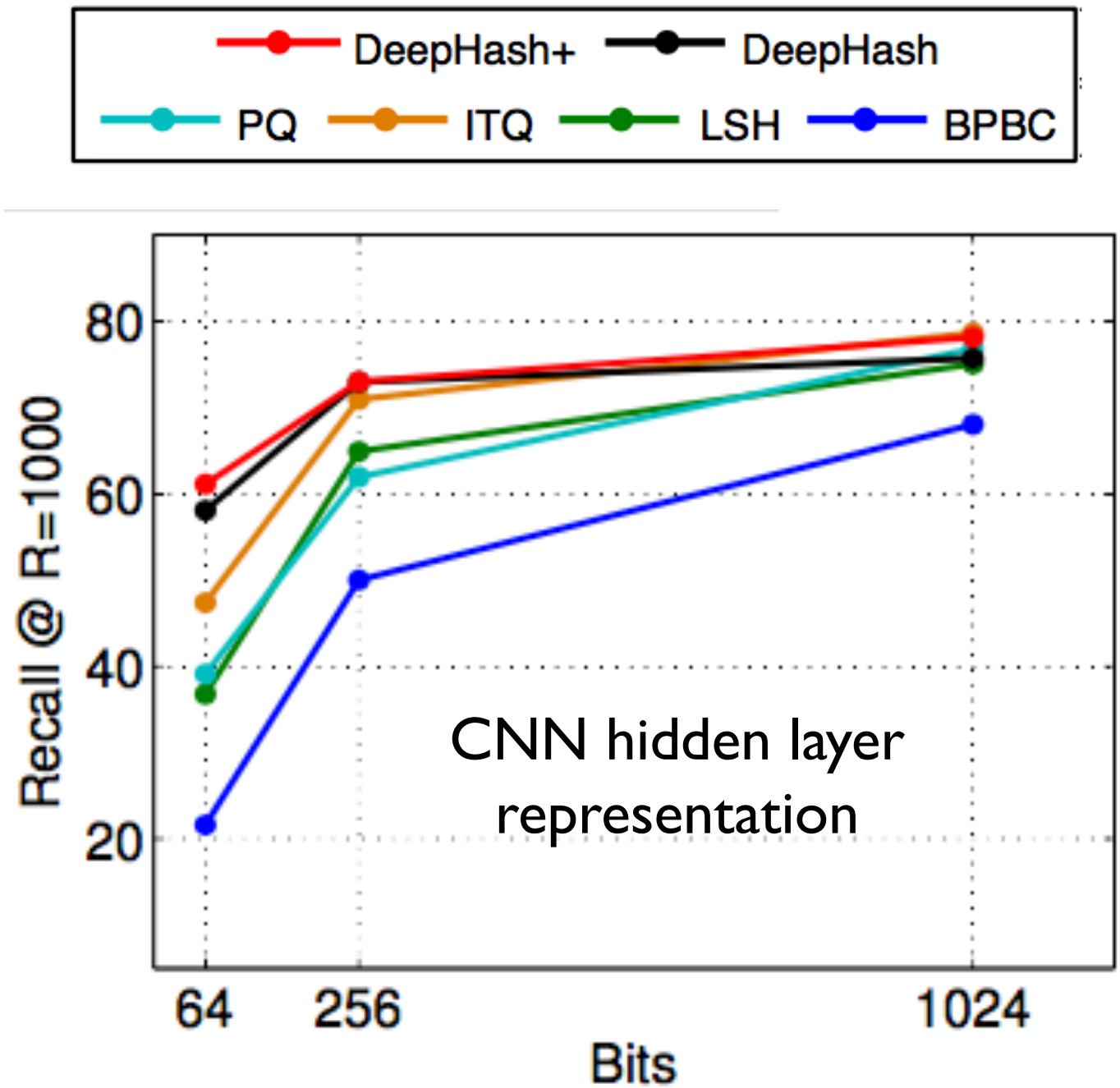
DeepHash: Getting Regularization, Depth and Fine-Tuning Right

Jie Lin[*,1,3], Olivier Morère[*,1,2,3], Vijay Chandrasekhar[1,3], Antoine Veillard[2,3], Hanlin Goh[1,3]
I2R[1], UPMC[2], IPAL[3]

ICML 2017




DeepHash+
DeepHash
PQ
ITQ
LSH
BPBC
old-school feature vector representation
Recall @ R=1000
80
60
40
20
64
256
1024
Bits


DeepHash+
DeepHash
PQ
ITQ
LSH
BPBC
Recall @ R=1000
80
60
40
20
64
256
1024
Bits
CNN hidden layer representation