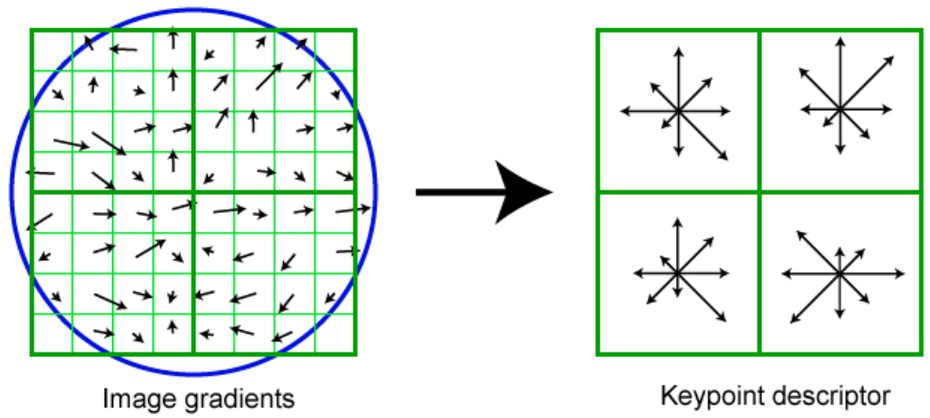
Texture

- Catchup
  - HOG, filter banks
- Texture generation
- Bag-of-words
- (Spatial) pyramid matching

Post-processing



1. Rescale 128-dim vector to have unit norm

$$x = \frac{x}{||x||}, \quad x \in R^{128}$$

“invariant to linear scalings of intensity”

2. Clip high values

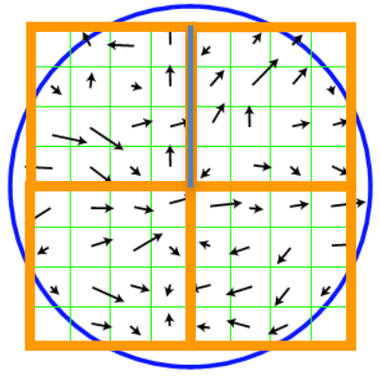
$$x := \min(x, .2)$$

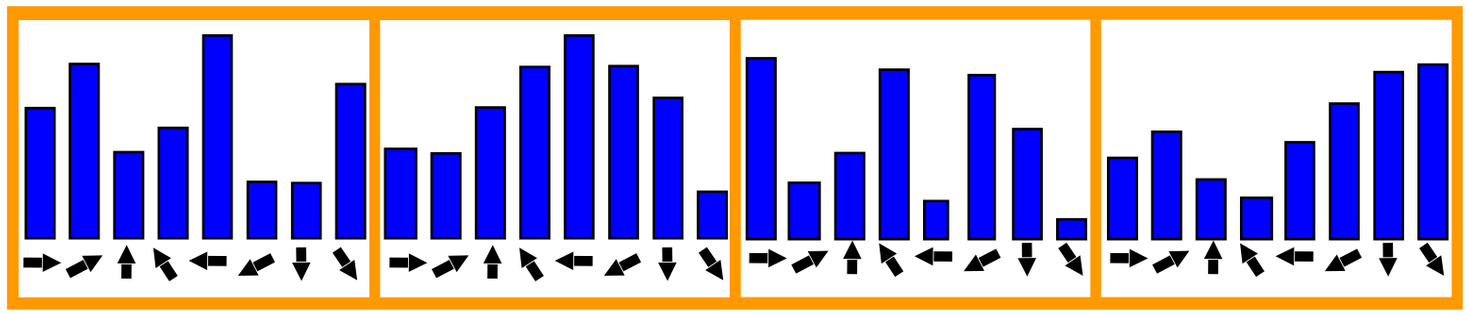
$$x := \frac{x}{||x||}$$

approximate binarization allows for for flat patches with small gradients to remain stable

Background: SIFT image patch descriptor

Histograms of gradient directions over spatial cells



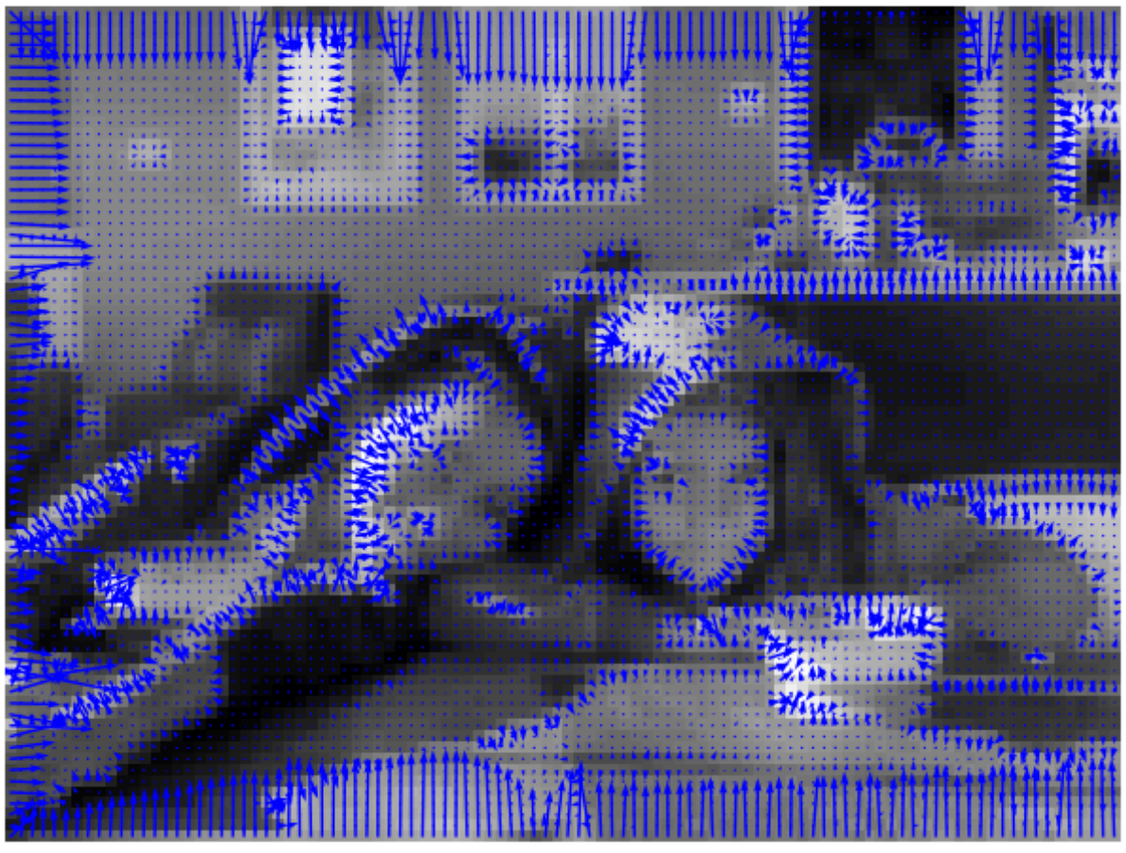


1. Compute gradient vector at each pixel (with [-1 0 1] and $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$ filters)
2. Snap gradient vector to one of 8 orientations (North, West, Northwest,…)
3. Split up patch into 2x2 quadrants and count the number of (North, West, Northwest,…) gradients

Important details for implementation

- To deal with color images, compute gradient on each channel and select the maximum-norm gradient for each pixel. My suggestion; first implement HOG for grayscale
- Natural to use convolve2d, but padding can effect results. Try out different strategies! I found using [-1 1] on boundary pixels and [-1 0 1] on interior works well.
- Finally, each pixel adds a wegihted vote to histogrm, where weight = gradient magnitude

Gradient computation



From demo_canny.ipynb

```
# Compute gradients with finite differencing
d   = np.array([1,0,-1],ndmin=2);
fx  = signal.convolve2d(im2,d,mode='same');
fy  = signal.convolve2d(im2,d.T,mode='same');
```

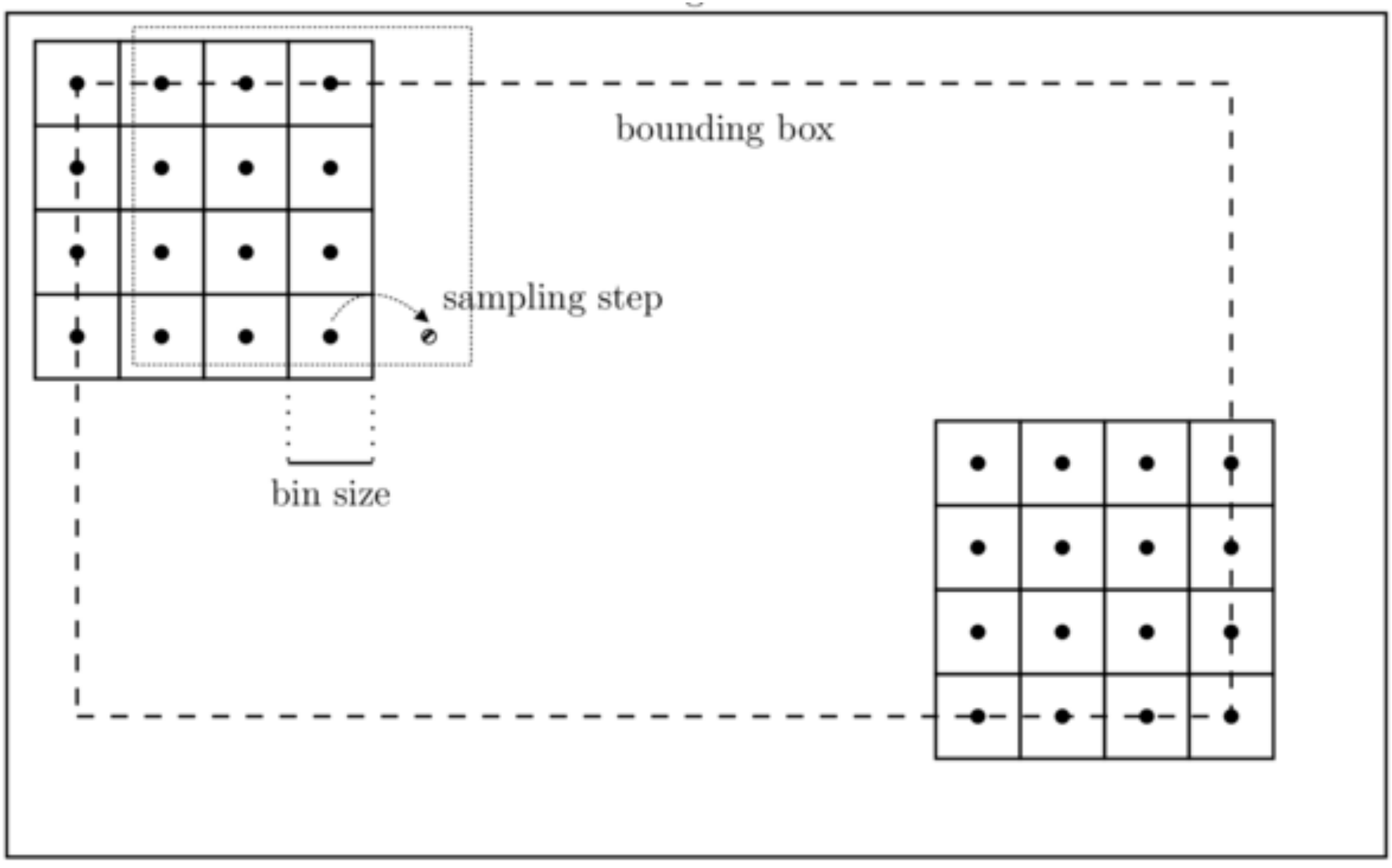
Histograms of Oriented Gradients (45K citations)

Compute 128-dim SIFT descriptors *blocks* densely across image

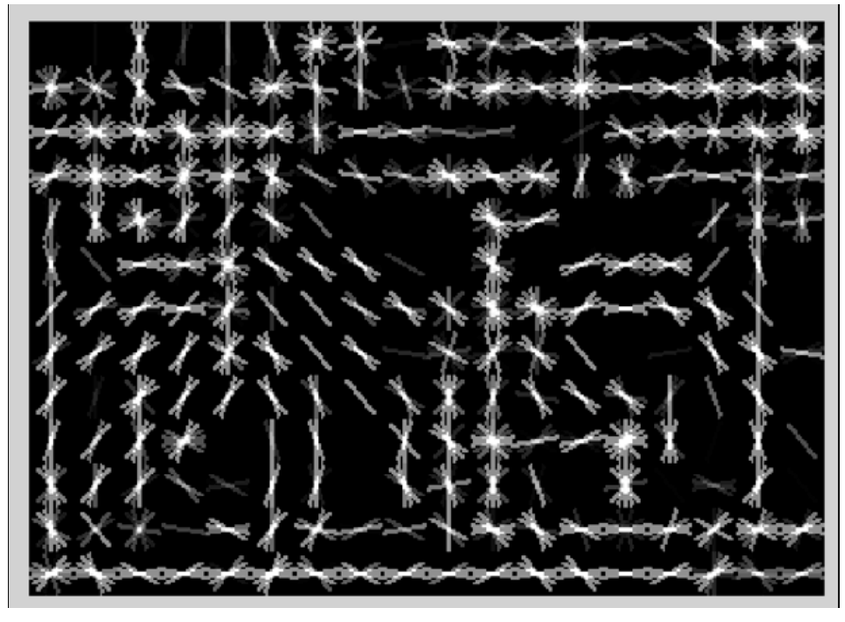
Naive implementation: loop over all possible patches and compute for each
Insight: Compute descriptor blocks at strides of a cell (e.g., 4-pixel shifts).
This allows you to compute gradients *once* on whole image and compute histograms of gradient orientations *once*.

What is output size of compute_gradients() and bin_gradients()?

[My own implementation still iterators over all block positions. Maybe you can find a way to avoid a double-for loop!]



Slick visualization of bin_gradients() output



Contrast HOG with classic (canny) edge detection… what is the philosophical change?

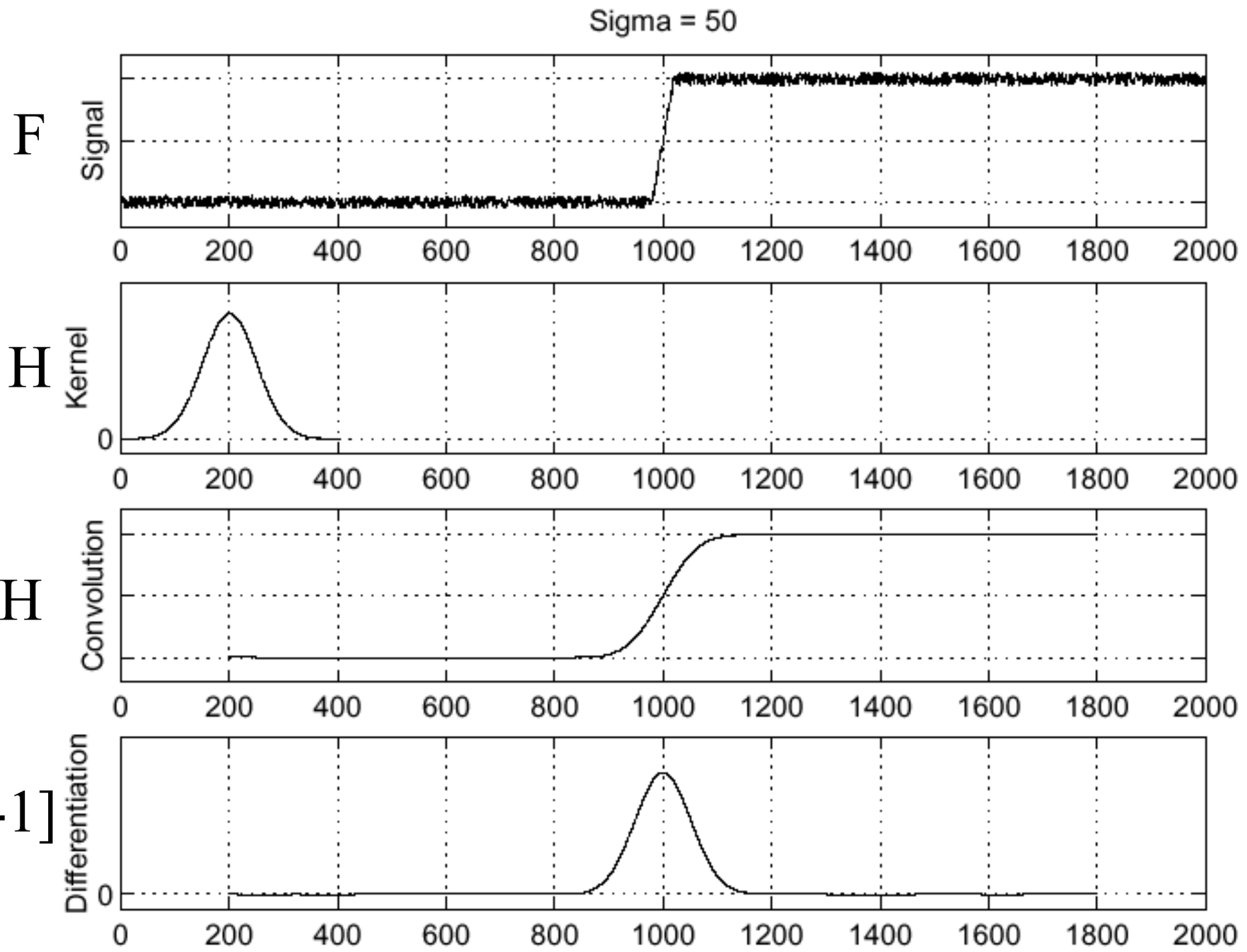
Canny: smooth, compute gradients, thin+threshold

HOG: compute gradients, *sparse multichannel image*, smooth, normalize

Texture

- Catchup
  - HOG, **filter banks**
- Texture generation
- Bag-of-words
- (Spatial) pyramid matching

Alternate approach to edge-finding



Find peaks of above profile be setting *its* gradient equal to 0.

What does the resulting edge filter look like?

Boring math for (finite approximation of) second-derivative filter

"Second derivative filter" G = [1 -1] convolved with [1 -1]

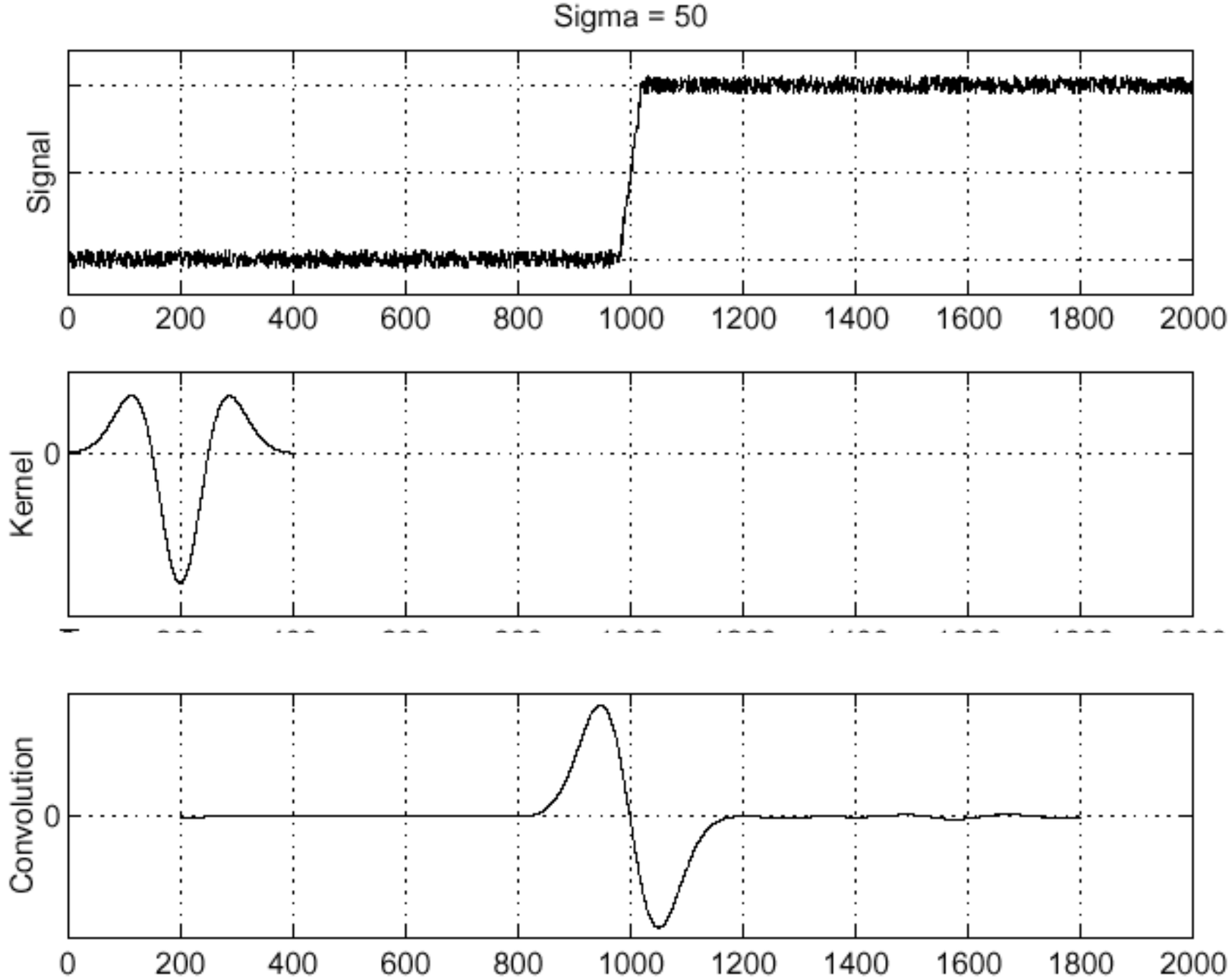
$$G[0] = 0 * -1 + 1 * 1 + -1 * 0 = 1$$
$$G[1] = -1 - 1 = -2$$
$$G[2] = 1 * 0 + -1 * -1 + 0 * 1 = 1$$

[Apologies; on this slide, "*" refers to standard multiplication and not convolution]

$$\mathrm{G} = [1\ \text{-}2\ 1]$$

Look for zero-crossings of second derivative

Consider smoothing (Gaussian) + second derivative [1 -2 1]



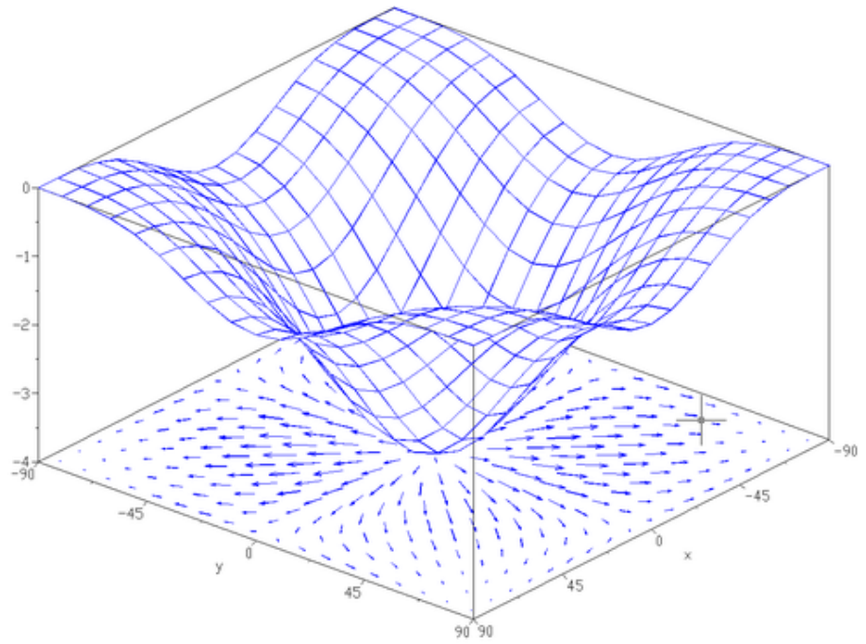
Zero-crossings of second graph

Generalization to 2D

Add second derivatives in each dimension

∇^2 is the **Laplacian** operator: $\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$

[sorrry, switched from ij to xy for filters]



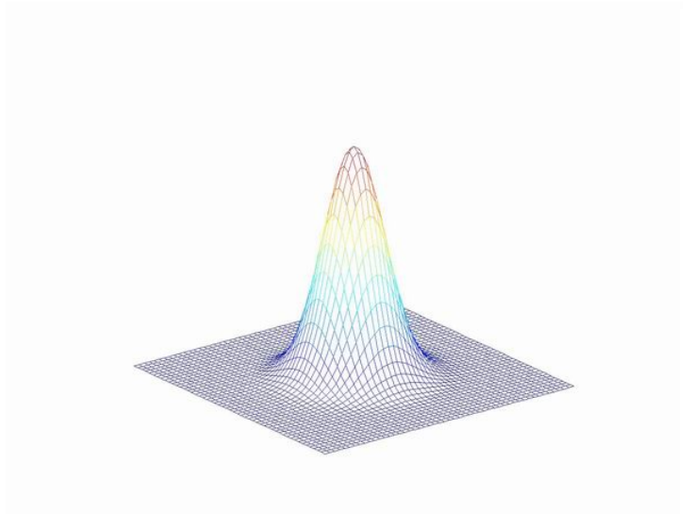
$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Aside: laplacian is equivalent to the divergence (“source-ness” or “sink”-ness) of a gradient of a function (used in fluid mechanics)

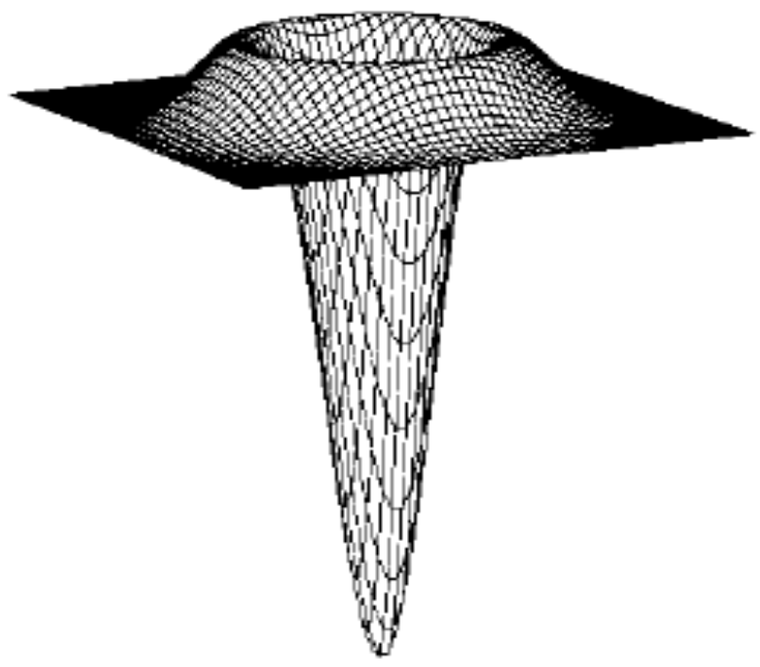
Laplacian of Gaussian (LOG)

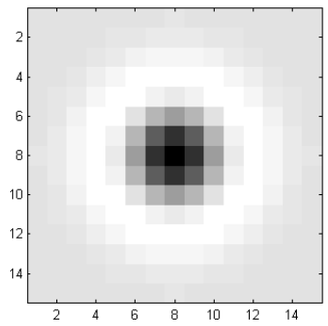
$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right]e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian



Laplacian of Gaussian

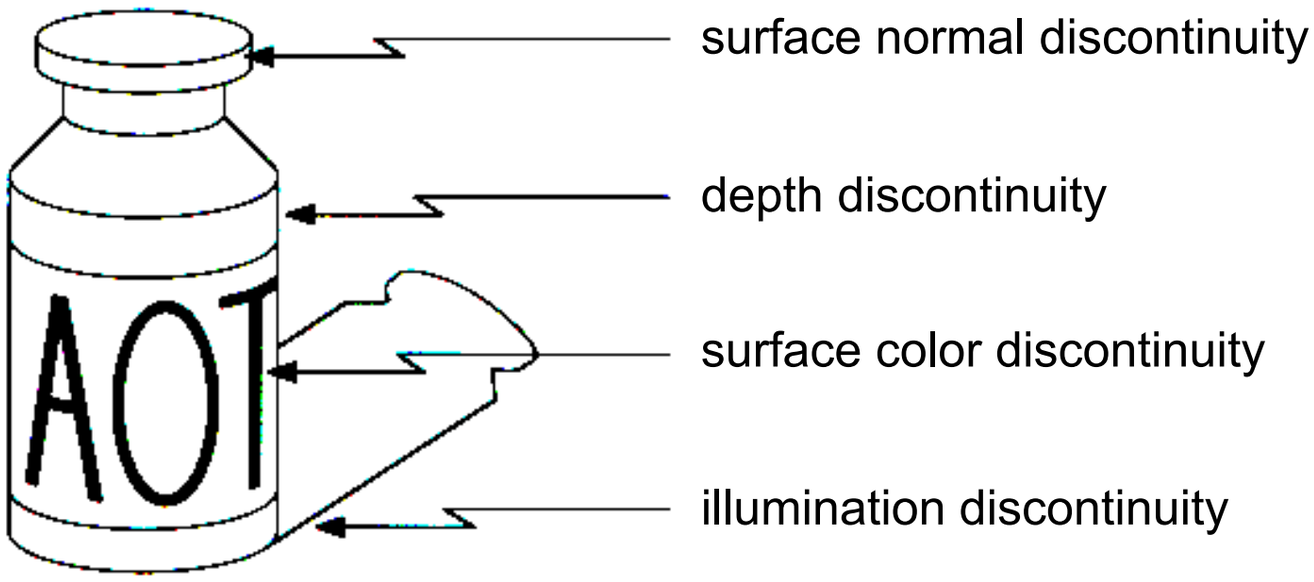


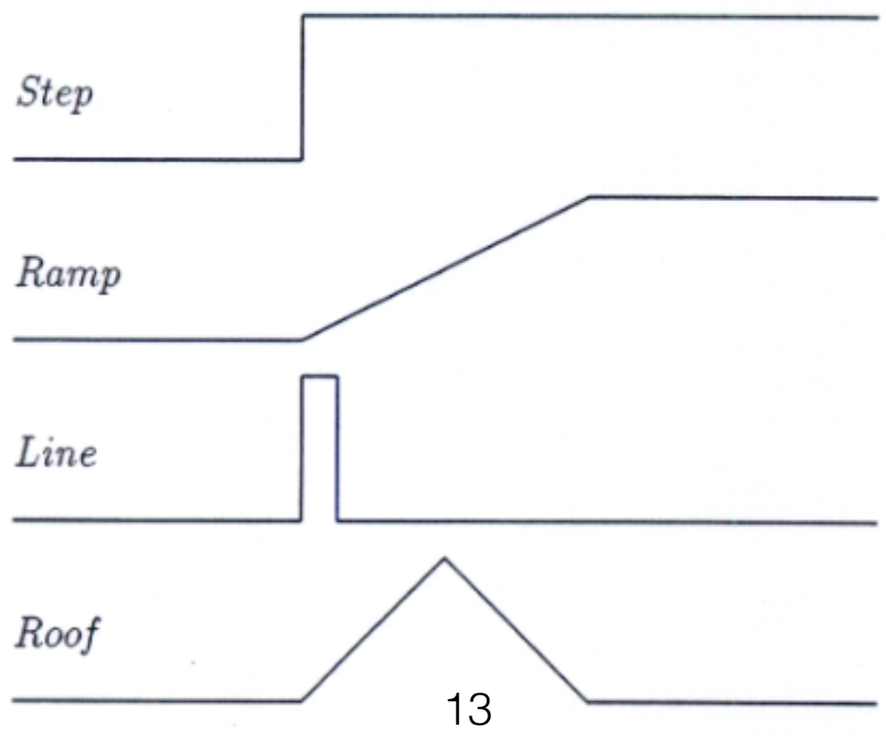


Marr & Hildreth

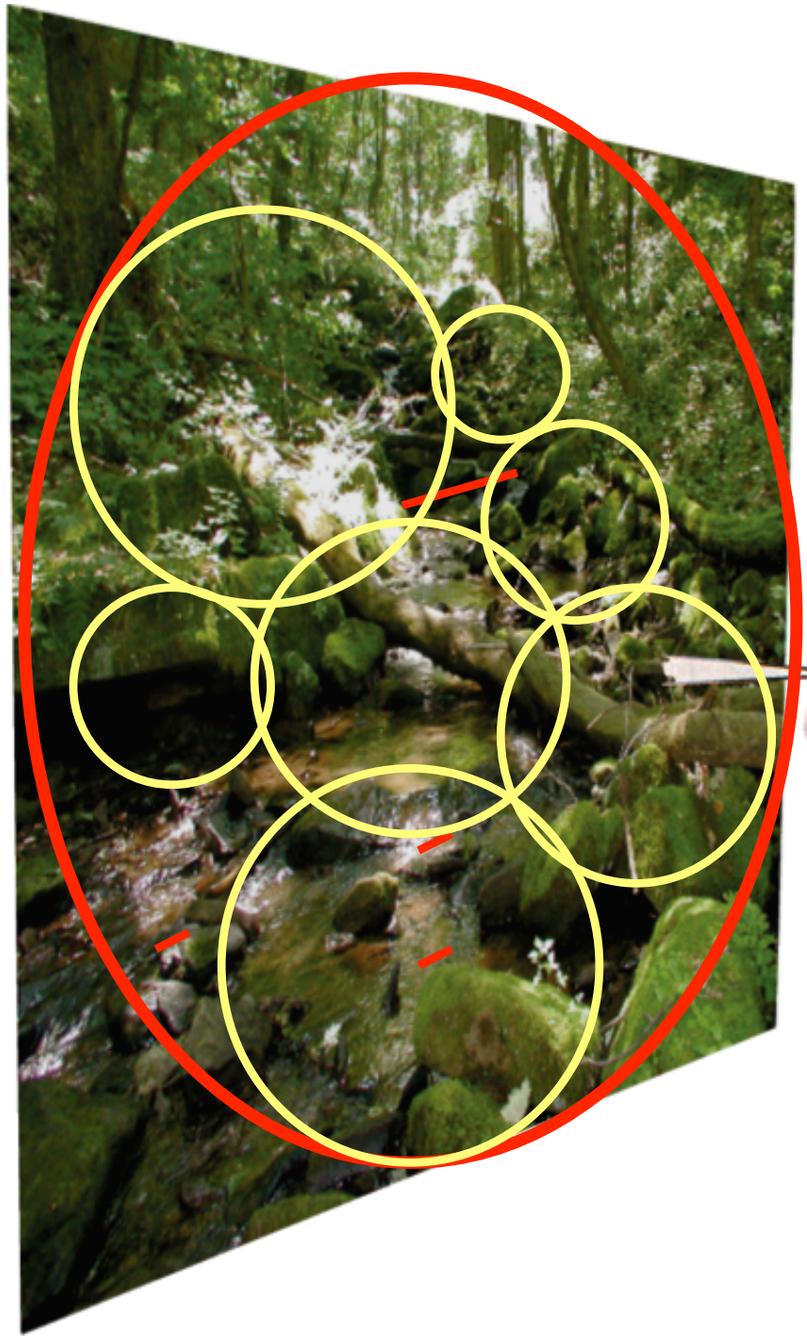
Though we've derived this to be an edge detector (by looking for where responses = 0), one could also use this as a dark-blob detector (by looking for where responses are high)

Other local features

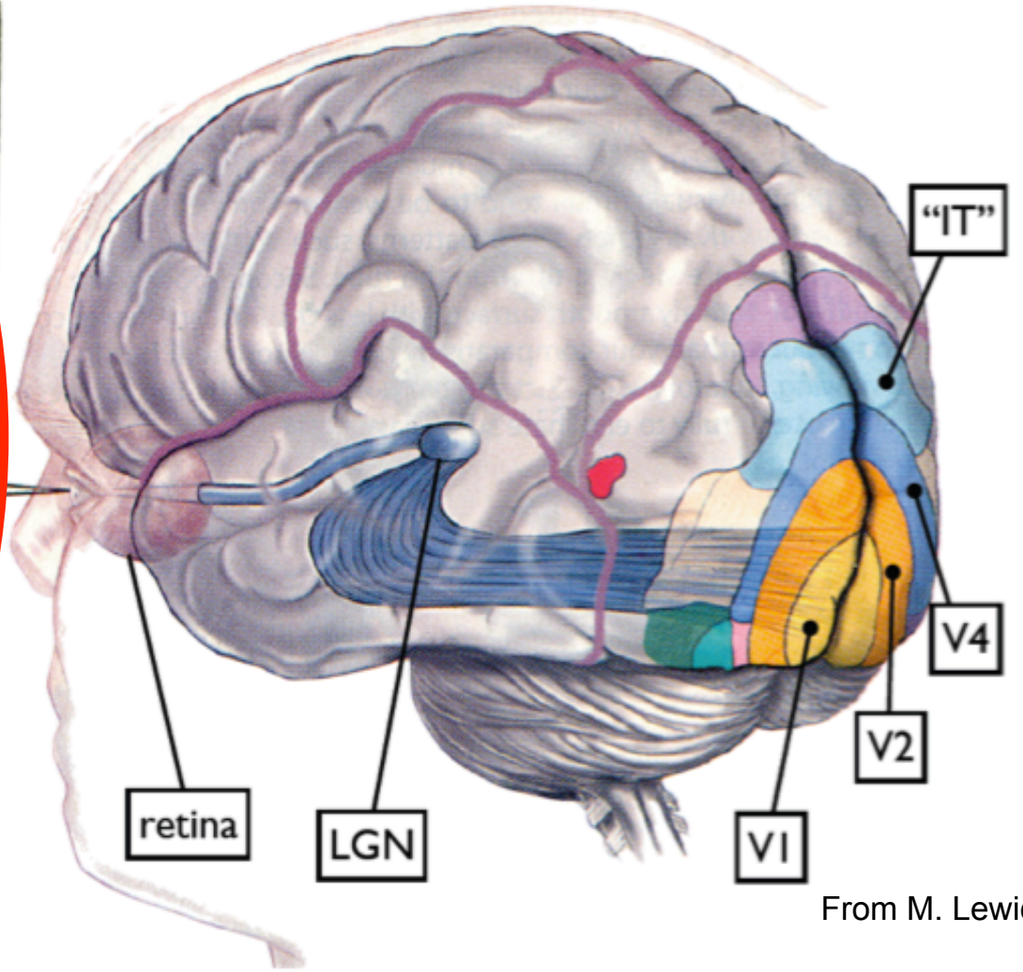




Biological motivation



Some visual areas…



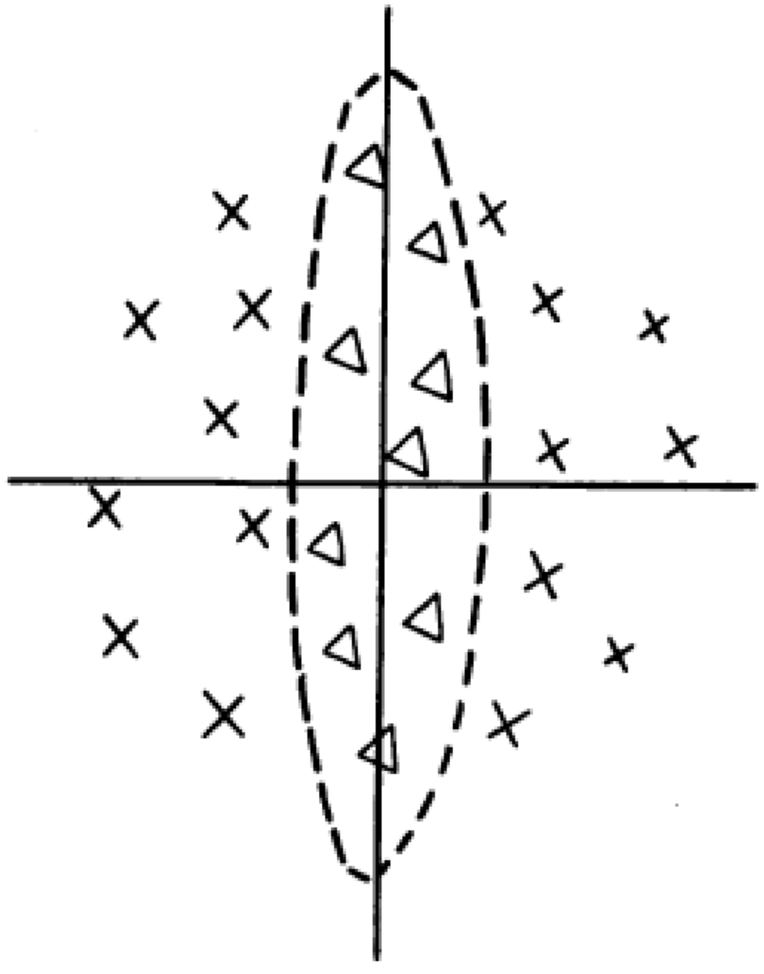
From M. Lewicky

J. Physiol. (1959) 148, 574–591

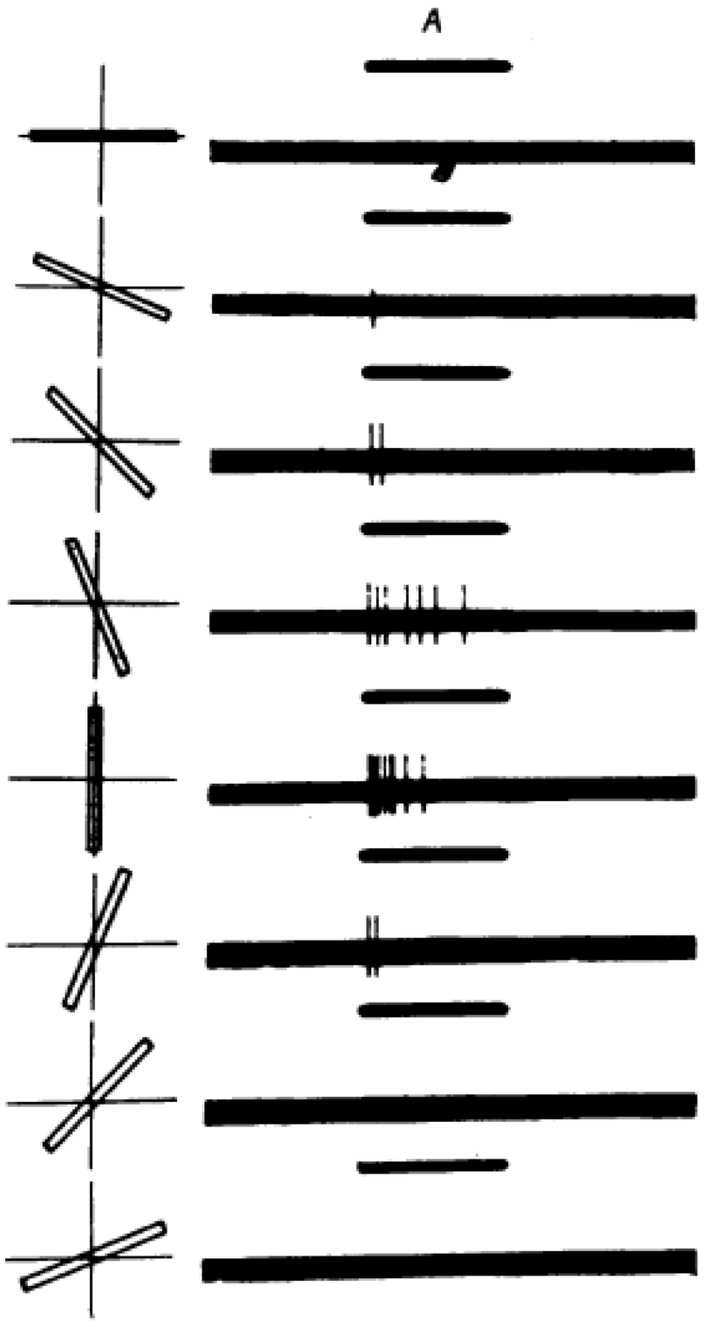
RECEPTIVE FIELDS OF SINGLE NEURONES IN THE CAT'S STRIATE CORTEX

By D. H. HUBEL* and T. N. WIESEL*

From the Wilmer Institute, The Johns Hopkins Hospital and University, Baltimore, Maryland, U.S.A.



Receptive field
of a cell in the cat's cortex



Responses to an oriented bar

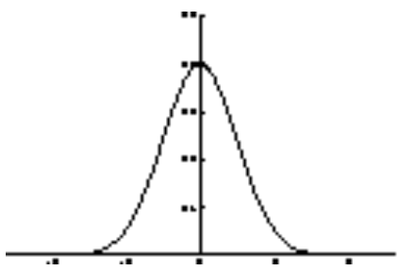
Other filters: Gaussian derivatives

$$G_0(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(\frac{-z^2}{2}) \quad ; \quad z = \frac{x}{\sigma}$$

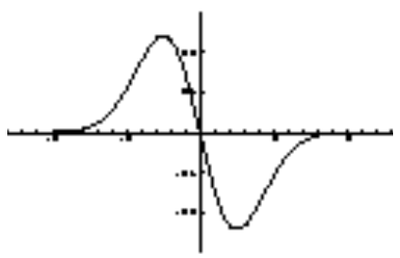
$$G_1(x) = -\frac{1}{\sigma} z G_0$$

$$G_2(x) = \frac{1}{\sigma^2}\left(z^2 - 1\right) G_0$$

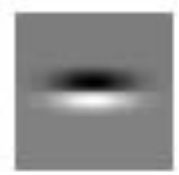
$$G_3(x) = -\frac{1}{\sigma^3}\left(z^3 - 3z\right) G_0$$

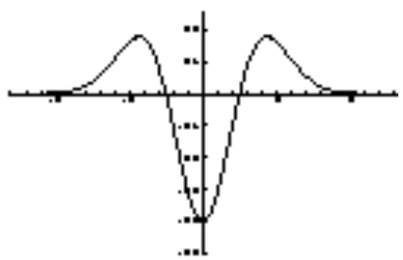


Gaussian

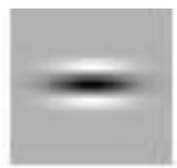


First derivative
“edges”





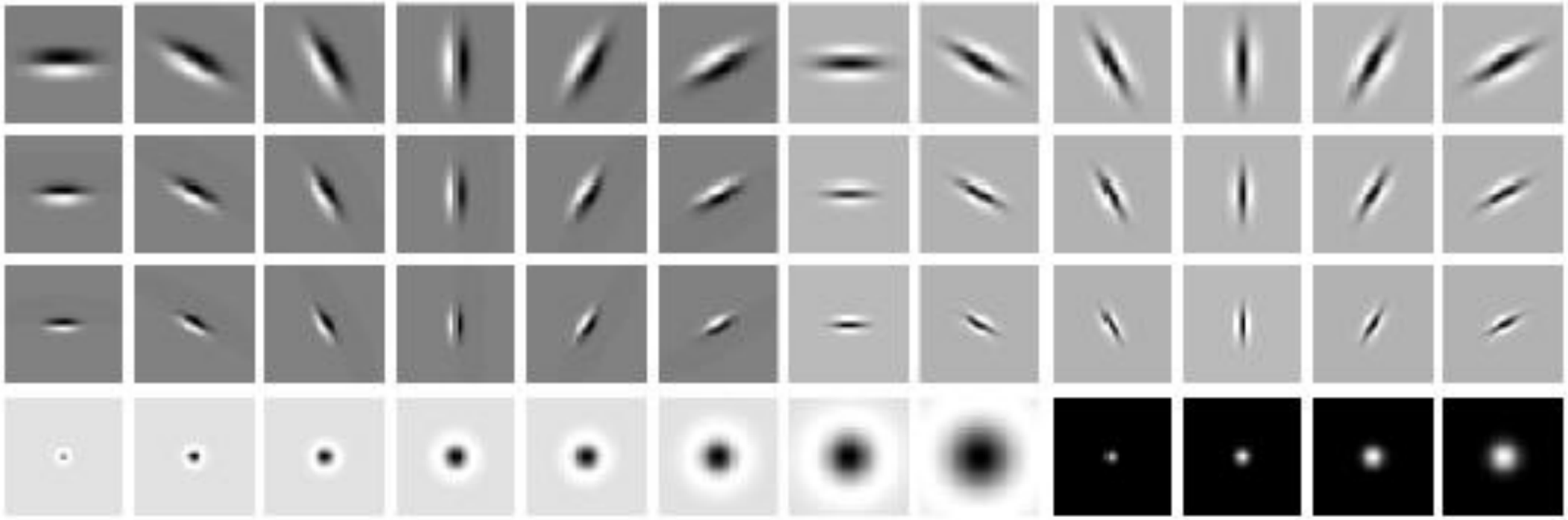
Second derivative
“bars”



$$G_1(x) = \frac{1}{\sigma\sqrt{2\pi}} \frac{-z}{\sigma} e^{-\frac{z^2}{2}}$$

What if I want to find skinny horizontal bars or “fat” 45 degree bars…

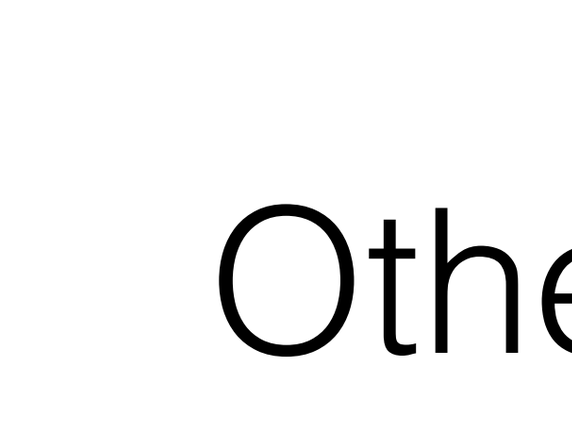
Filter banks



The LeungMalik filter bank has a mix of edge, bar and spot filters at multiple scales and orientations. It has a total of 48 filters - 2 Gaussian derivative filters at 6 orientations and 3 scales, 8 Laplacian of Gaussian filters and 4 Gaussian filters.

Other filters: gabors

Modulate (elementwise multiply) Gaussian with a cosine/sine wave

$e^{\frac{-x^2}{2\sigma^2}}$ 

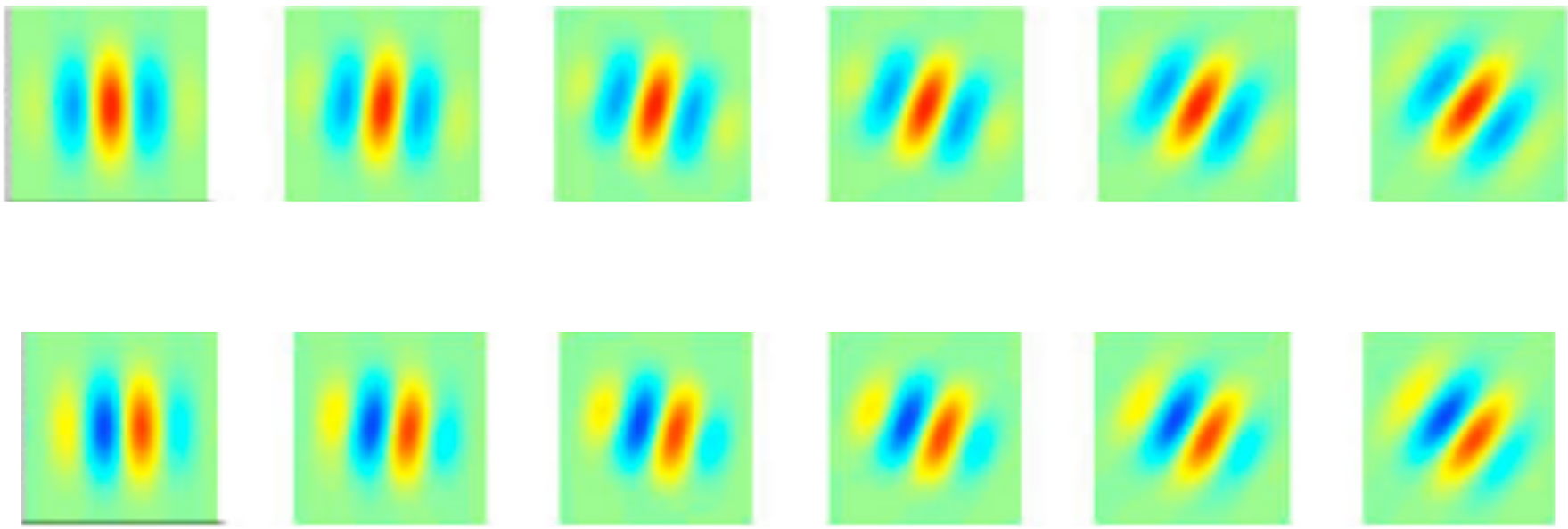
$\cos(\omega x)$ 

x

=

ω = frequency

Gabor filters



Think of as “oriented stripes” match filters

http://en.wikipedia.org/wiki/Gabor_filter

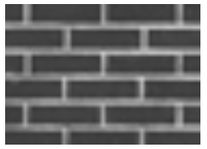
It turns out, we can write cosine + sine modulated gabor filters as real and imaginary parts of a single complex filter (Fourier theory; in 2 lectures)

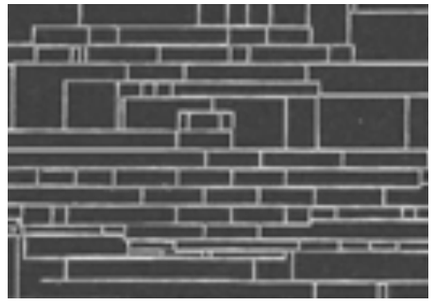
Texture

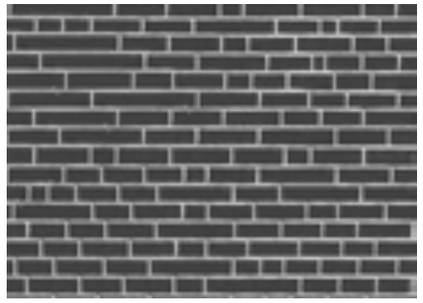
- Catchup
  - HOG, filter banks
- **Texture generation**
- Bag-of-words
- (Spatial) pyramid matching

Three tasks we'll answer today

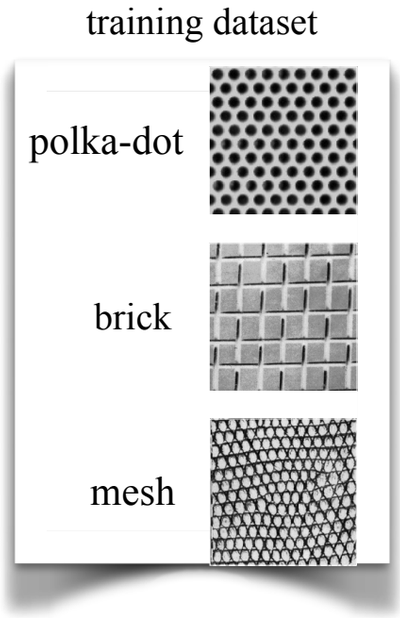
Texture generation:







Texture classification:



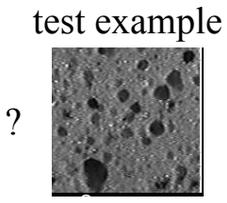
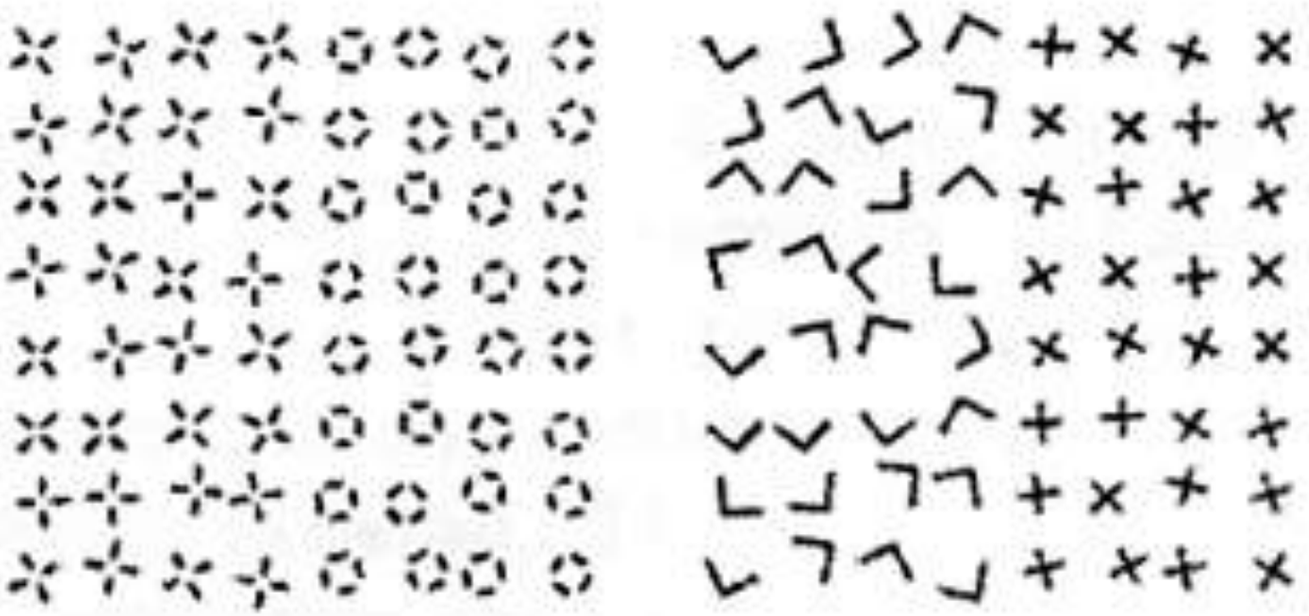


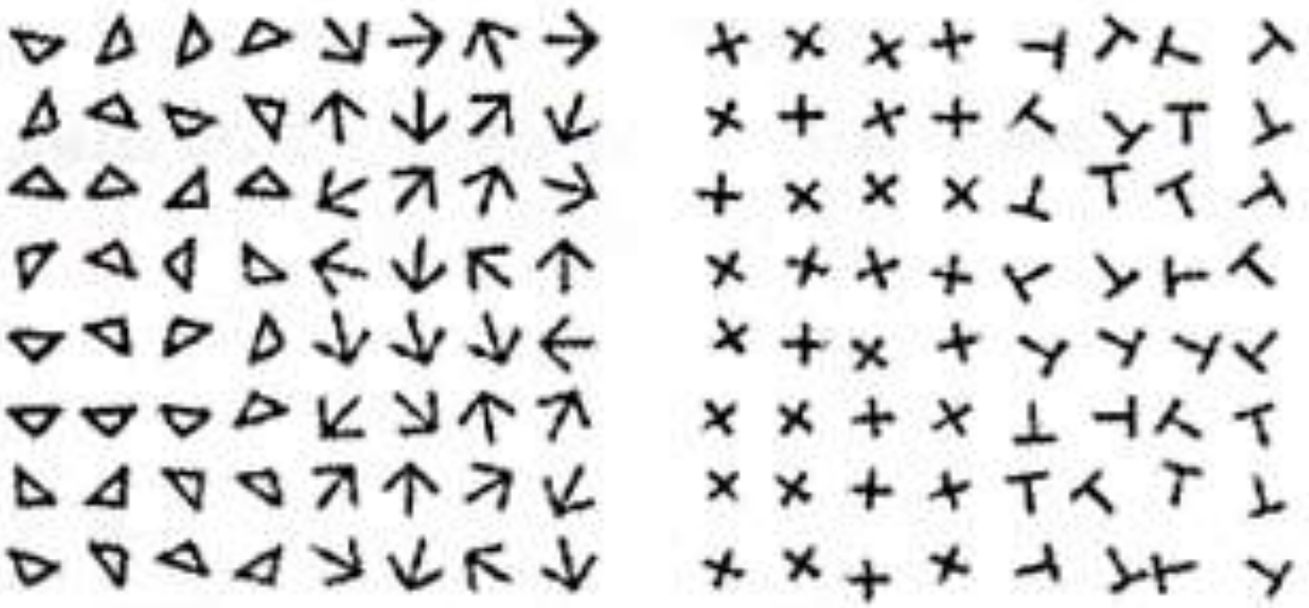
Image classification:

(by approximating images as textures)
"bag of visual words"

Inspiration from human vision: pre-attentive texture discrimination



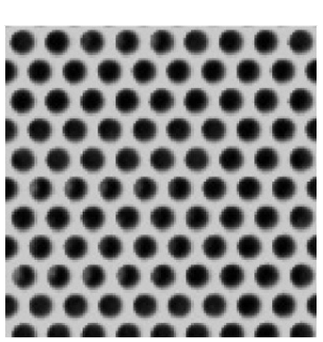
(Julesz,1981)



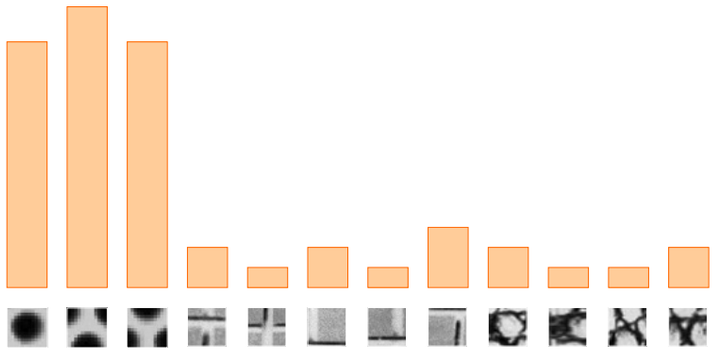
“textons”

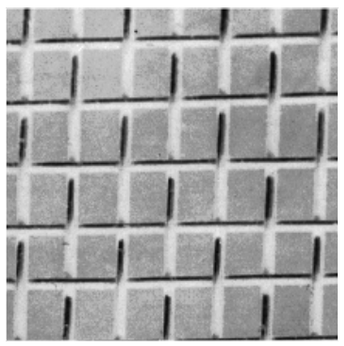
160 ms, outside foveal gaze
Instantenous, or effortless texture discrimination

Where we are headed…
(Old HW1)

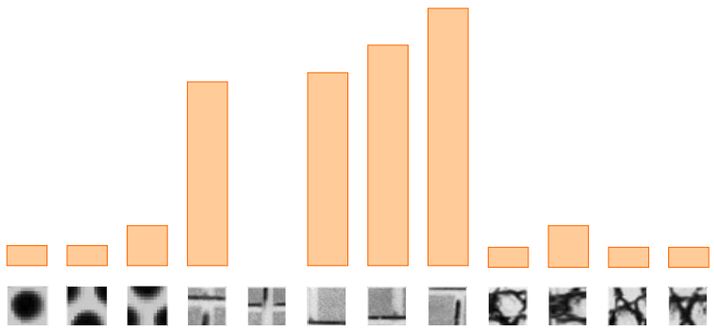


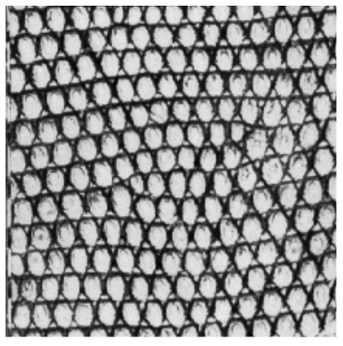
polka-dot



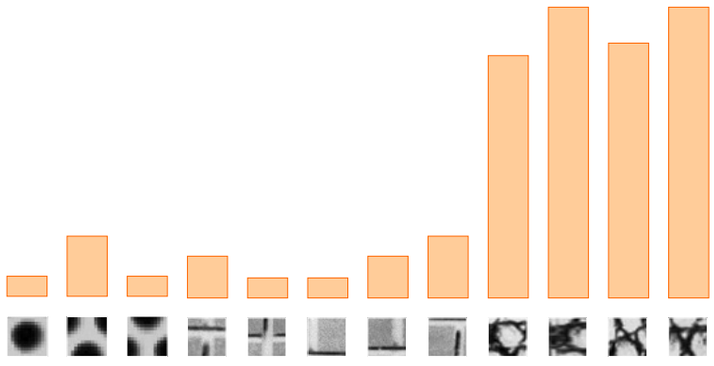


brick

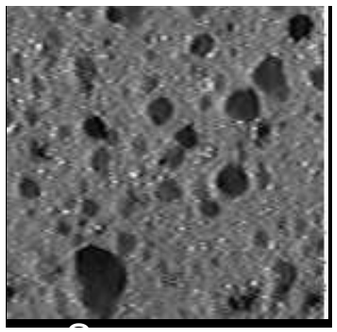


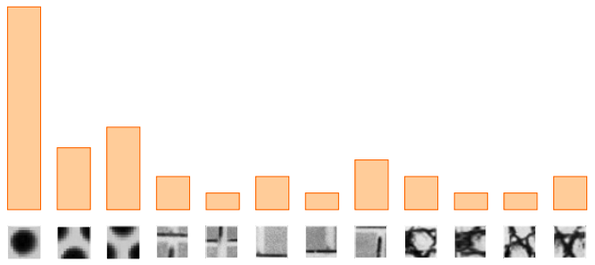


mesh



match





Representing textures

Let’s encode the texture as a distribution over localized visual elements, or “textons”

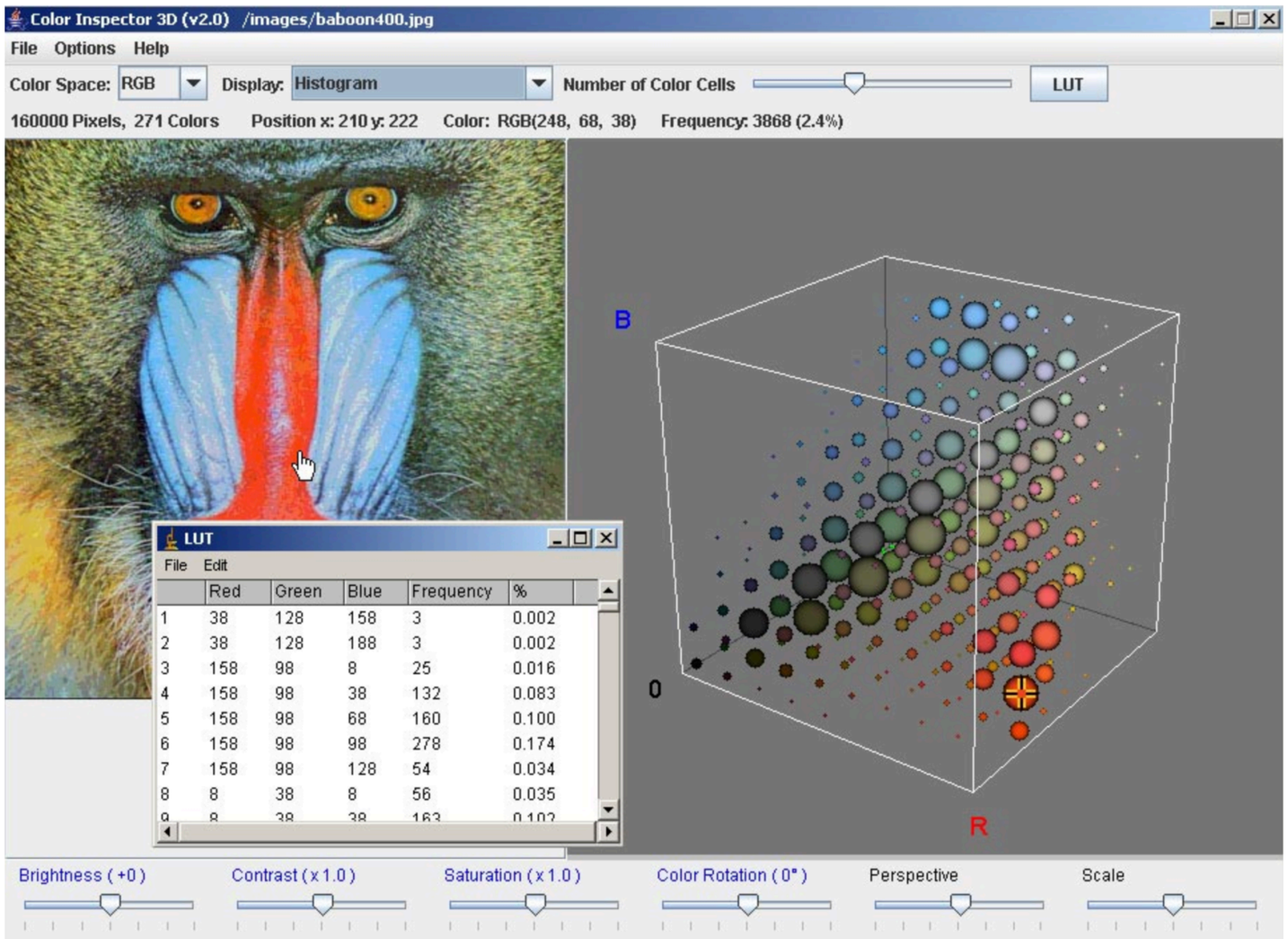
1. How do we represent a texton?

2. How do we represent a distribution over them?

Version 0

Build probabilistic model of RGB pixels where R,G,B are quantized into 8 bins

Build a color histogram (*Color Indexing*, IJCV 99)



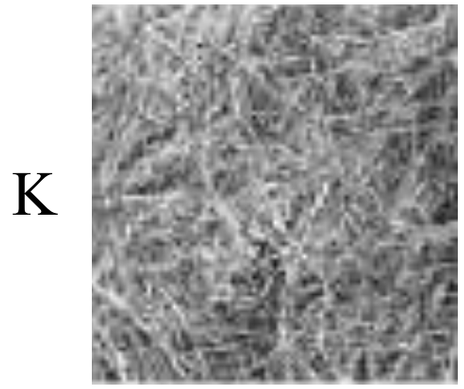
Generate pixels by rolling a $8^3 = 512$ sided die

What's wrong with this texture model?

Version 0.5

Instead of modeling *pixels*, lets model *patches* … that are grayscale and nonoverlapping for now

Let's build a discrete probability distribution (or histogram) over small KxK patches (for say, K≈50), where each pixel can take on one of N = 256 values.



What would be the size of this histogram?

Hopeless! (256x256x256….) = N^{KK} = 256^{2500}

Long digression: it turns out one can build such *conditional* histograms *on-the-fly* to generate image texture….

- P(pixel|neighboorhhood of pixels)
- Inspired by *markov* models for generating text p(next_word|previous_word)

"A dog is a man's best friend. It's a dog eat dog world out there."

prev_word

	a	dog	is	man's	best	friend	it's	eat	world	out	there	.
a		2/3		1/3								
dog			1/3					1/3	1/3			
is	1											
man's					1							
best						1						
friend												1
it's	1											
eat		1										
world										1		
out											1	
there												1
.							1					

next_word

Synthesized examples

"I Spent an Interesting Evening Recently with a Grain of Salt"

- Mark V. Shaney

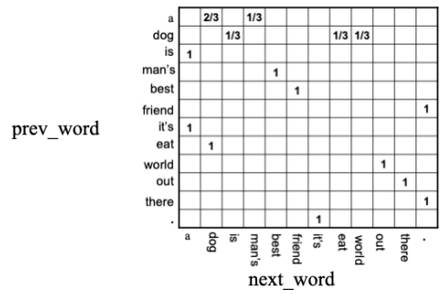
(computer-generated contributor to UseNet News group called net.singles)

Output of 2nd order word-level Markov Chain after training on 90,000 word philosophical essay:

"Perhaps only the allegory of simulation is unendurable--more cruel than Artaud's Theatre of Cruelty, which was the first to practice deterrence, abstraction, disconnection, deterritorialisation, etc.; and if it were our own past. We are witnessing the end of the negative form. But nothing separates one pole from the very swing of voting "rights" to electoral..."

Key idea for generation with high-order markov models: compute single rows of this table on-the-fly

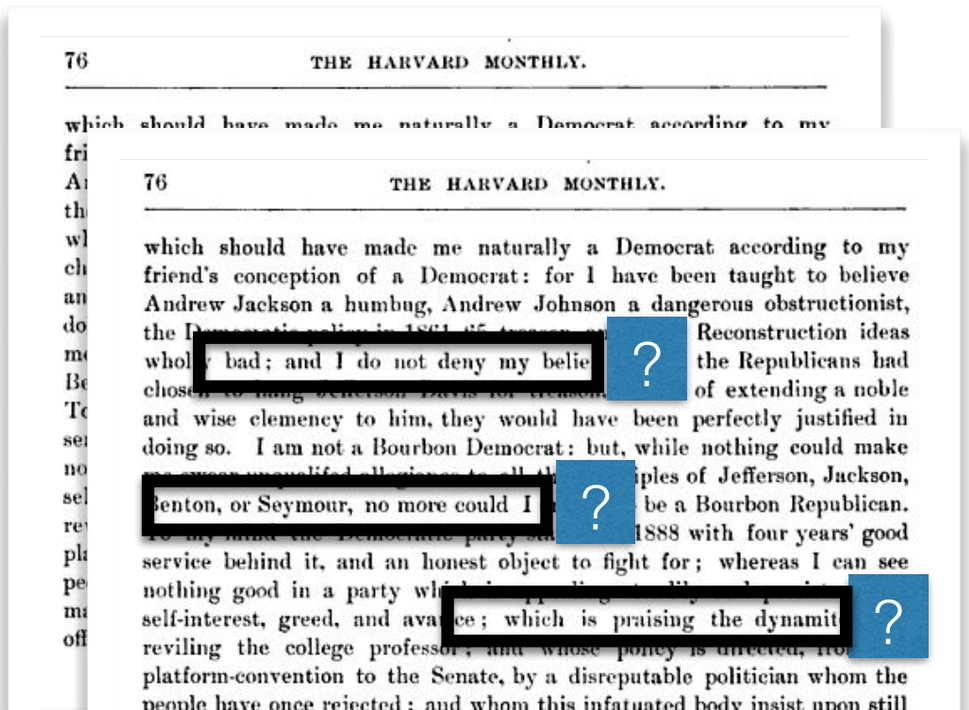
by searching for matches to markov context in training data



Generated text

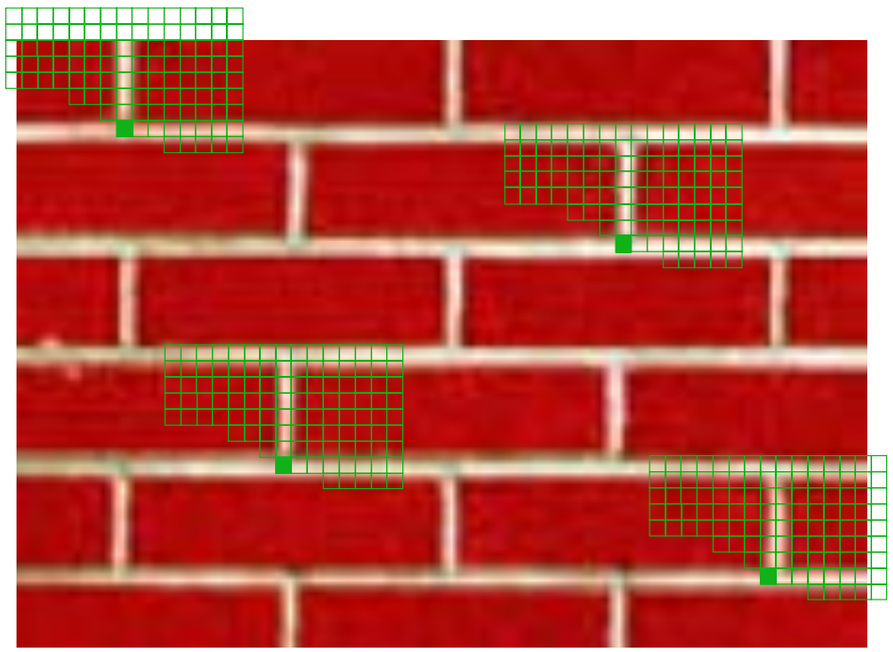
"Perhaps only the allegory of simulation is unendurable--more cruel than Artaud's Theatre of Cruelty, which was the first to practice deterrence, abstraction, disconnection, deterritorialisation, etc.; and if it were our own past. We are witnessing the end of the negative form. But nothing separates one pole from the very swing of voting "rights" to ?

Training corpus

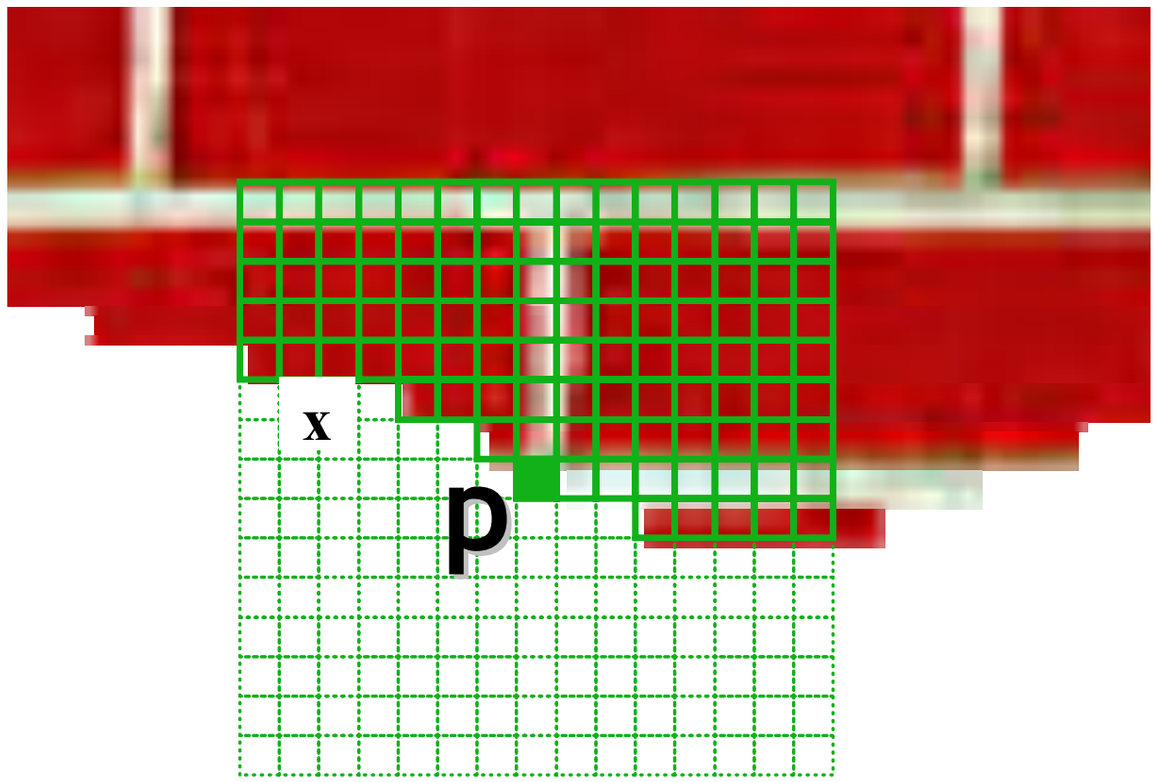


pretend these are matches:)

Translating to an image: implicitly build histograms *on-the-fly*



input image

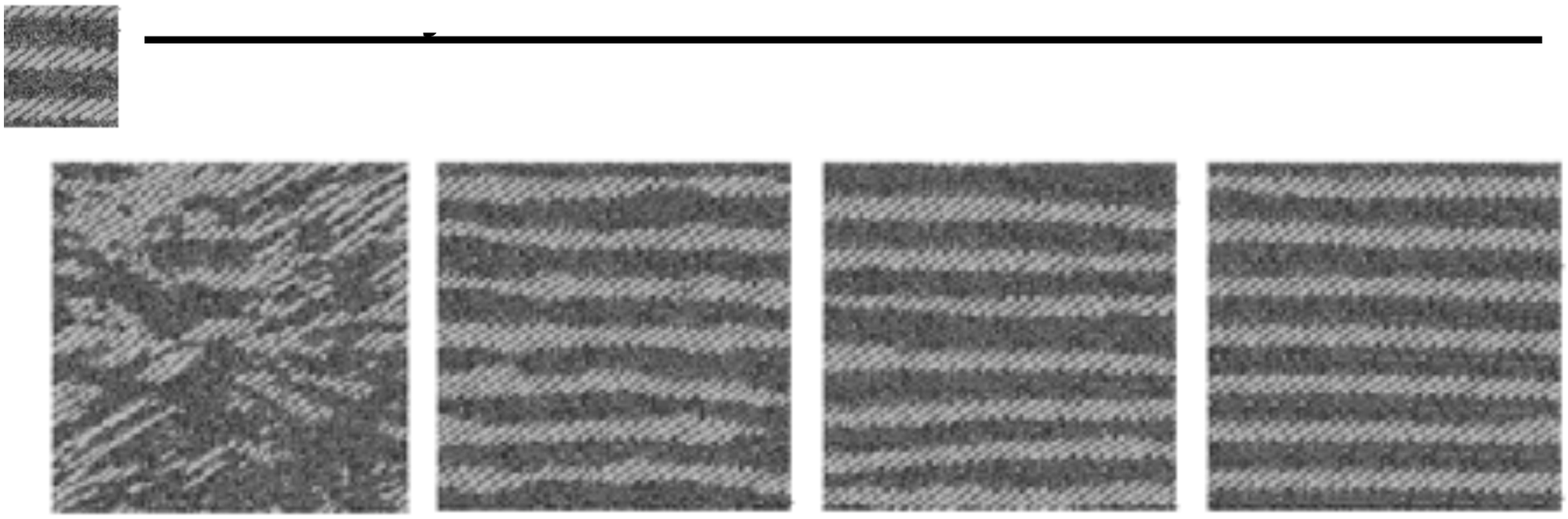


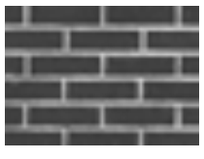
synthesized image

- Assume we've synthesized a partial image (somehow), and would like to synthesize the *next* pixel.
- Ideally, we'd like to sample from P(**x**| neighborhood of already synthesized pixels around **x**)
- Rather than explicitly constructing such a table…
  - find all windows from input image that match the neighborhood
  - pick one window at random
  - assign **x** to the center of that window!

Texture Synthesis by Non-parametric Sampling

Alexei A. Efros and Thomas K. Leung
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776, U.S.A.
{efros,leungt}@cs.berkeley.edu





Increasing window size →

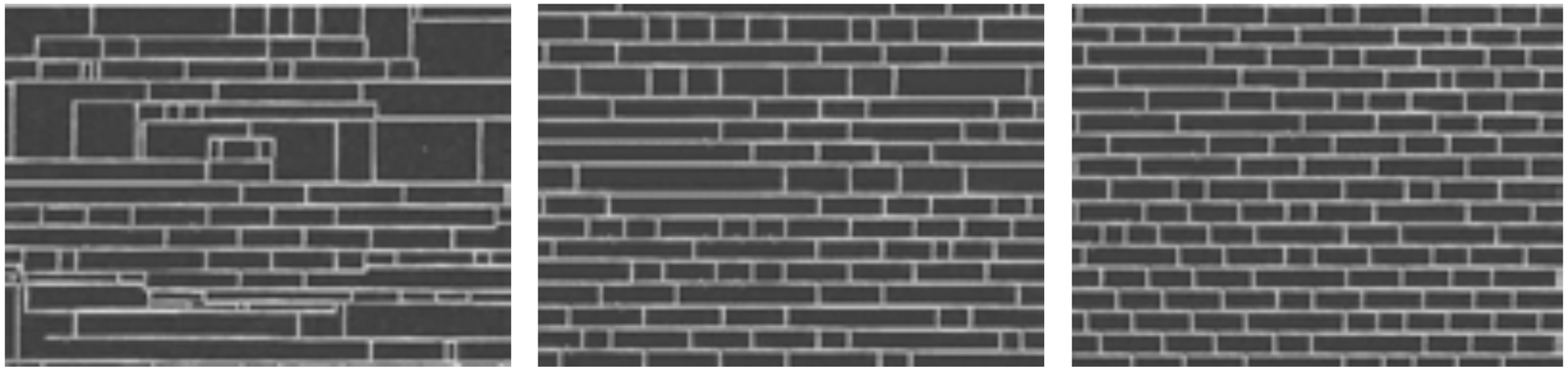
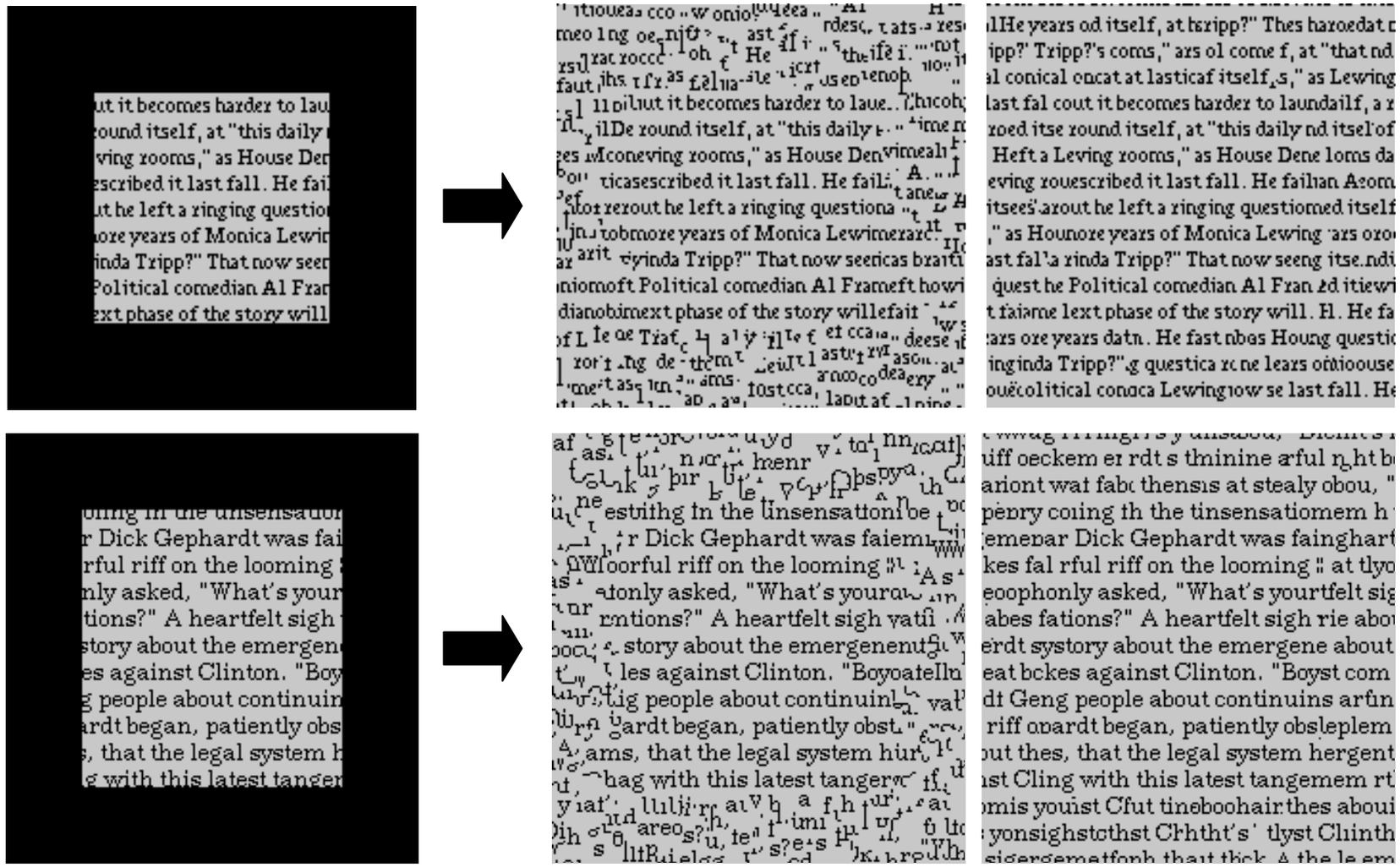


Image generation of text!

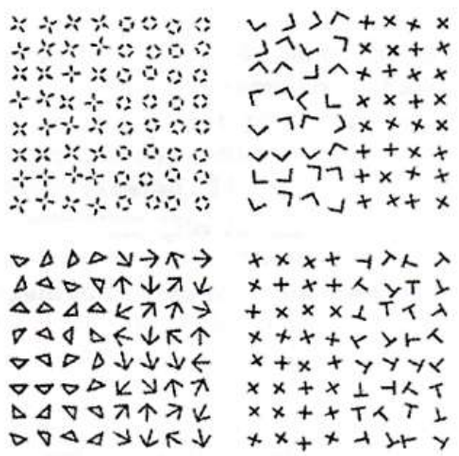


Back to texture *analysis* (vs synthesis)

Let's build a discrete probability distribution (or histogram) over small KxK (≈ 50x50) patches, where each pixel can take on one of N = 256 values.

Explicitly representing joint statistics over KxK patches is too hard since we'd need histograms / tables of size (256x256x256….) = $N^{KK} = 256^{2500}$

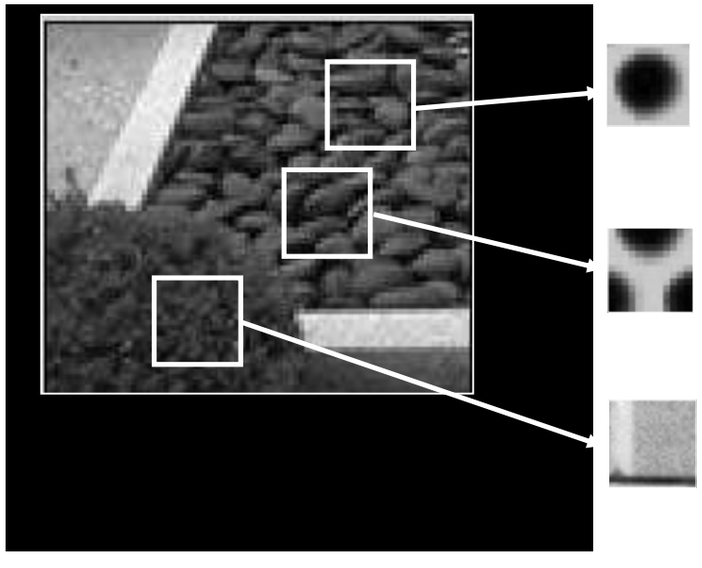
Revisit inspiration from human vision



(Julesz,1981)

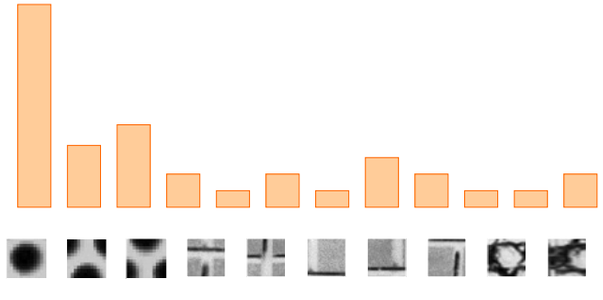
“textons”

Let’s build of *vocabulary* of K = 1000 textons or “visual words” or *typical* 50x50 patches

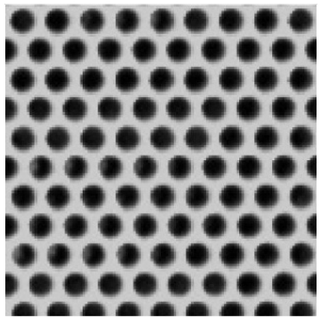


We can then represent an image as a bag-of-visual-words by…

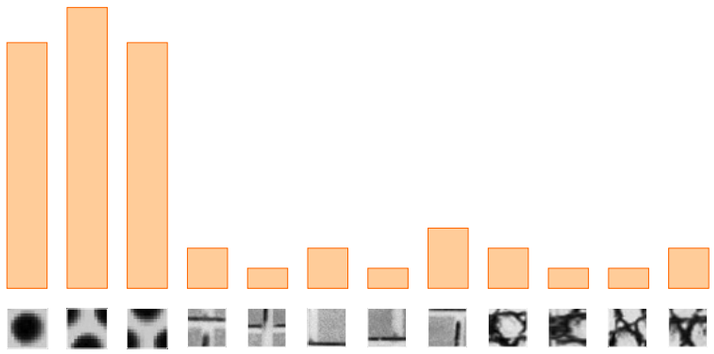
1. Quantize each overlapping 50x50 patch into 1 of K textons
2. Represent entire image as a histogram with K bins

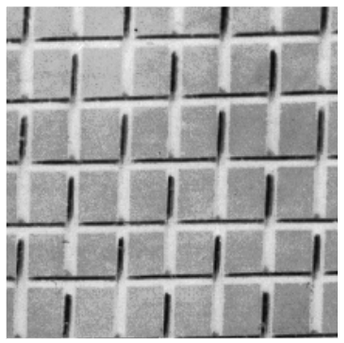


Texture recognition

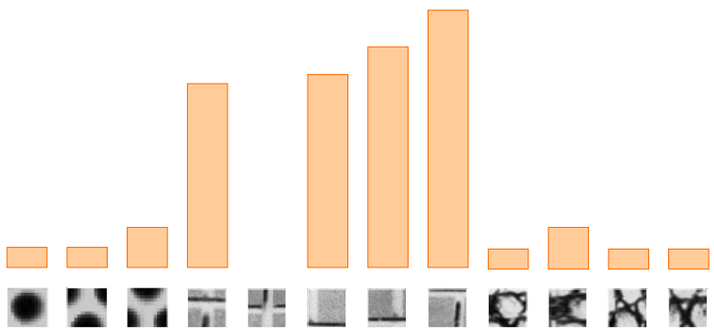


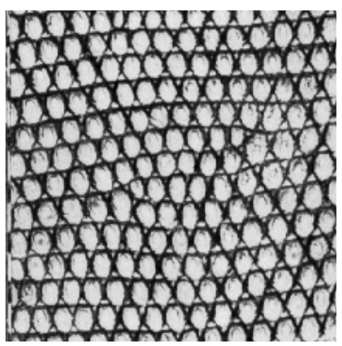
polka-dot



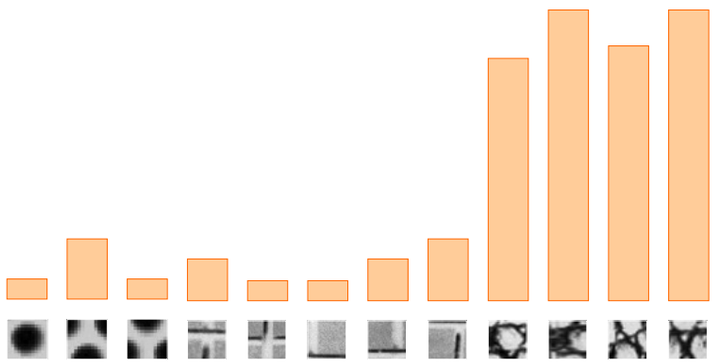


brick

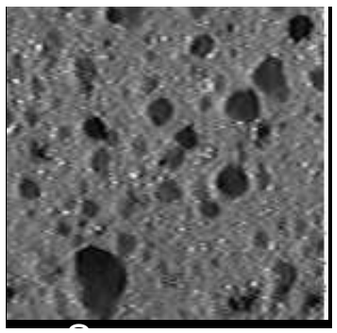


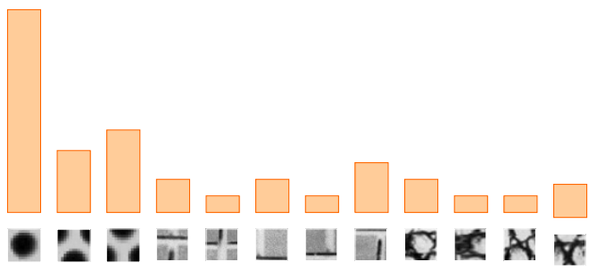


mesh



match





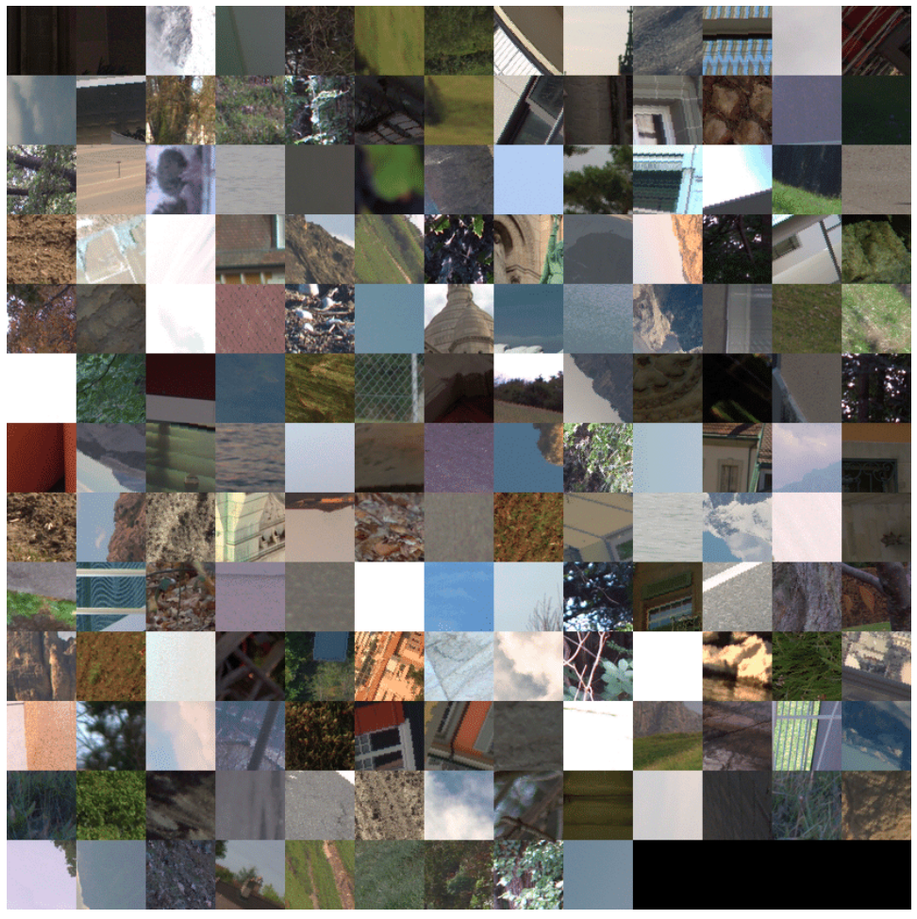
Crucial questions:

(1) How to build effective vocabulary of visual words?

(2) How to quantize each patch to a visual word?

(3) How to compare histograms?

Some approaches



1. Pick a random set of K patches

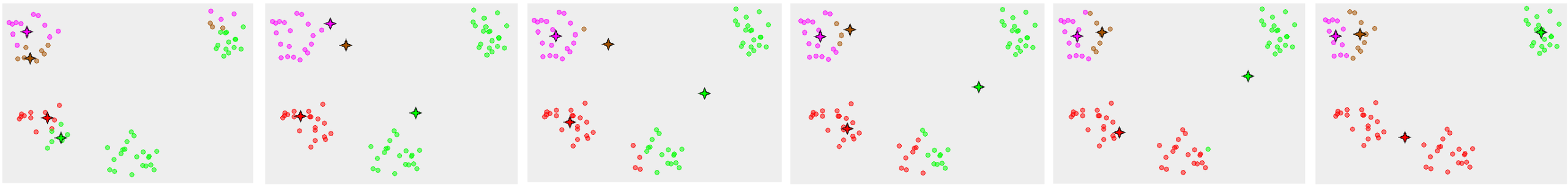
(works suprisingly well for large K… why not randomly sampling pixel values?)

2. Given a large dataset of random patches, pick K representitive ones

with clustering algorithm (e.g., K-means)!

K-means

visual intuition



0. Start with random assignment of centroids {purple, red,…}
1. Color each data point according to its closest centroid
2. Recompute the {purple,red,…} centroid by averaging

Cost function

$\min_{Z,D} C(Z, D, X)$ where $C(Z, D, X) =$

$x_i \in R^N$: i$^{\text{th}}$ input vector (to be clustered)

$z_i \in \{1 \ldots K\}$: i$^{\text{th}}$ label

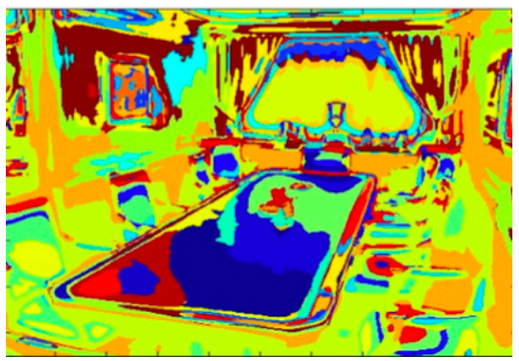
d_k: k$^{\text{th}}$ dictionairy element (or mean)

Coordinate descent optimization

$$\min_{Z,D} C(Z, D, X) \quad \text{where} \quad C(Z, D, X) = \sum_i ||x_i - d_{z_i}||^2$$

1. $\min_Z C(Z, D, X)$

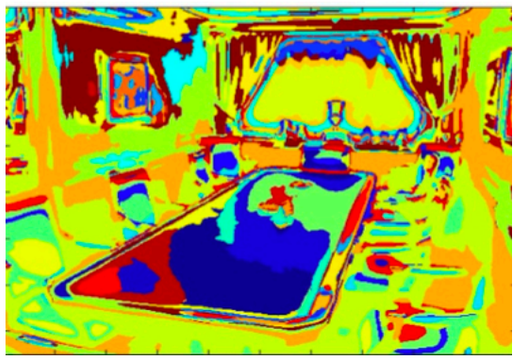
2. $\min_D C(Z, D, X)$

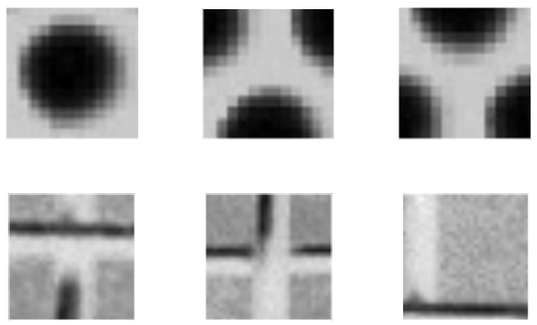


Training vs testing

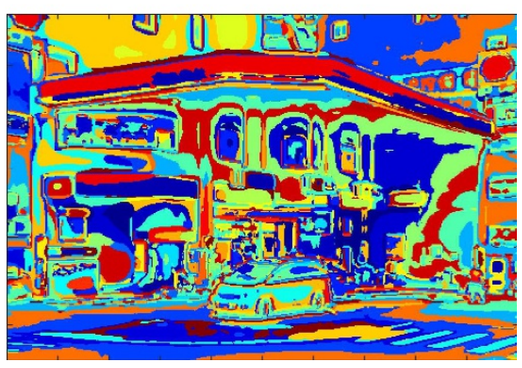
$$C(Z, D, X) = \sum_i ||x_i - d_{z_i}||^2$$

Training: $\min_{Z,D} C(Z, D, X_{\text{train}})$



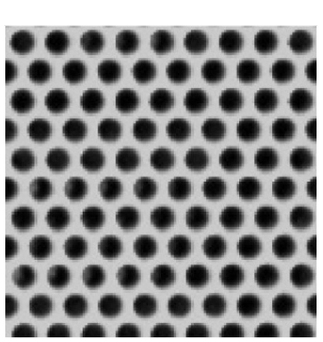


Testing: $\min_{Z} C(Z, D, X_{\text{test}})$

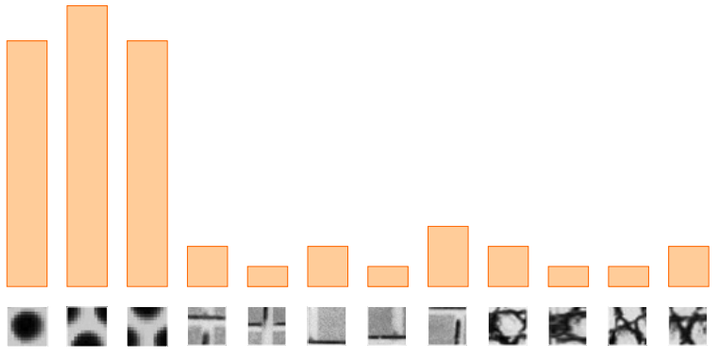


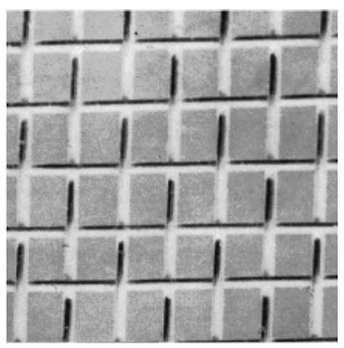
Question: why shouldn't you optimize over D at test?

Texture recognition

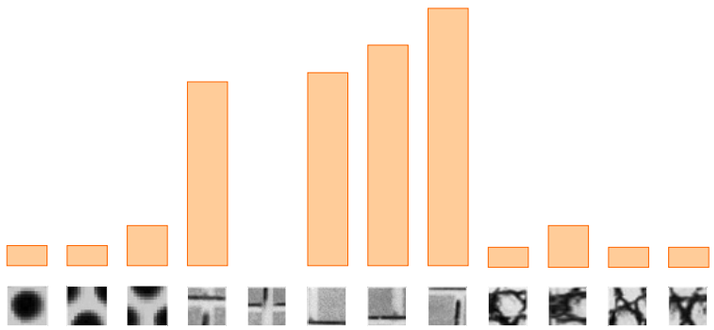


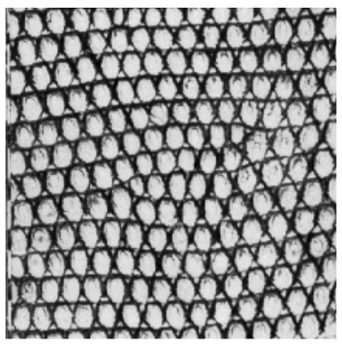
polka-dot



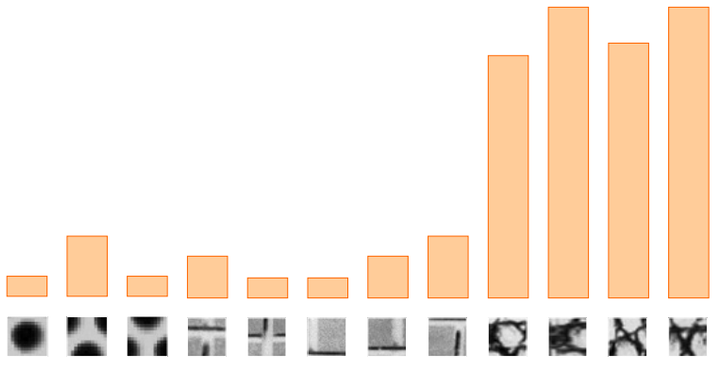


brick

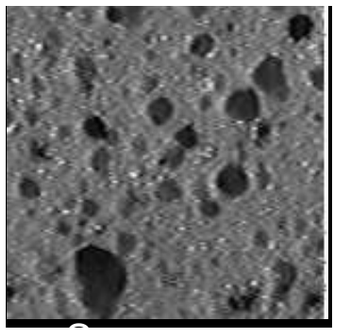


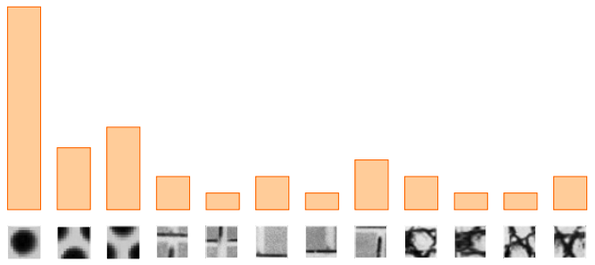


mesh



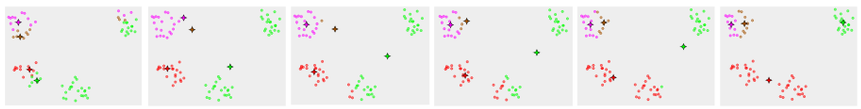
match



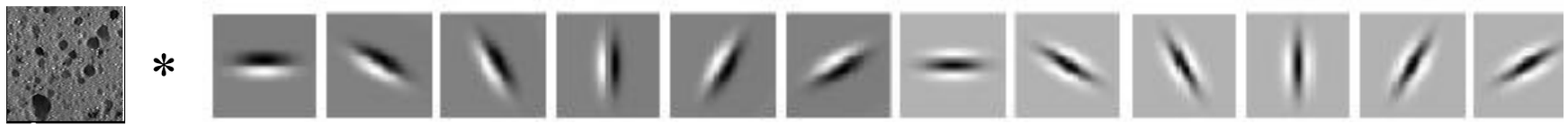


Major difficulty: working in high dimensions is slow and inaccurate

$$C(Z, D, X) = \sum_i ||x_i - d_{z_i}||^2$$



Rather than representing each patch with 2500 (50x50) numbers, let's *linearly project* patches down into 48 dimensional space with a filter bank



[Here * is element multiplication]

$$x \in R^{2500} \Rightarrow x \in R^{48}$$

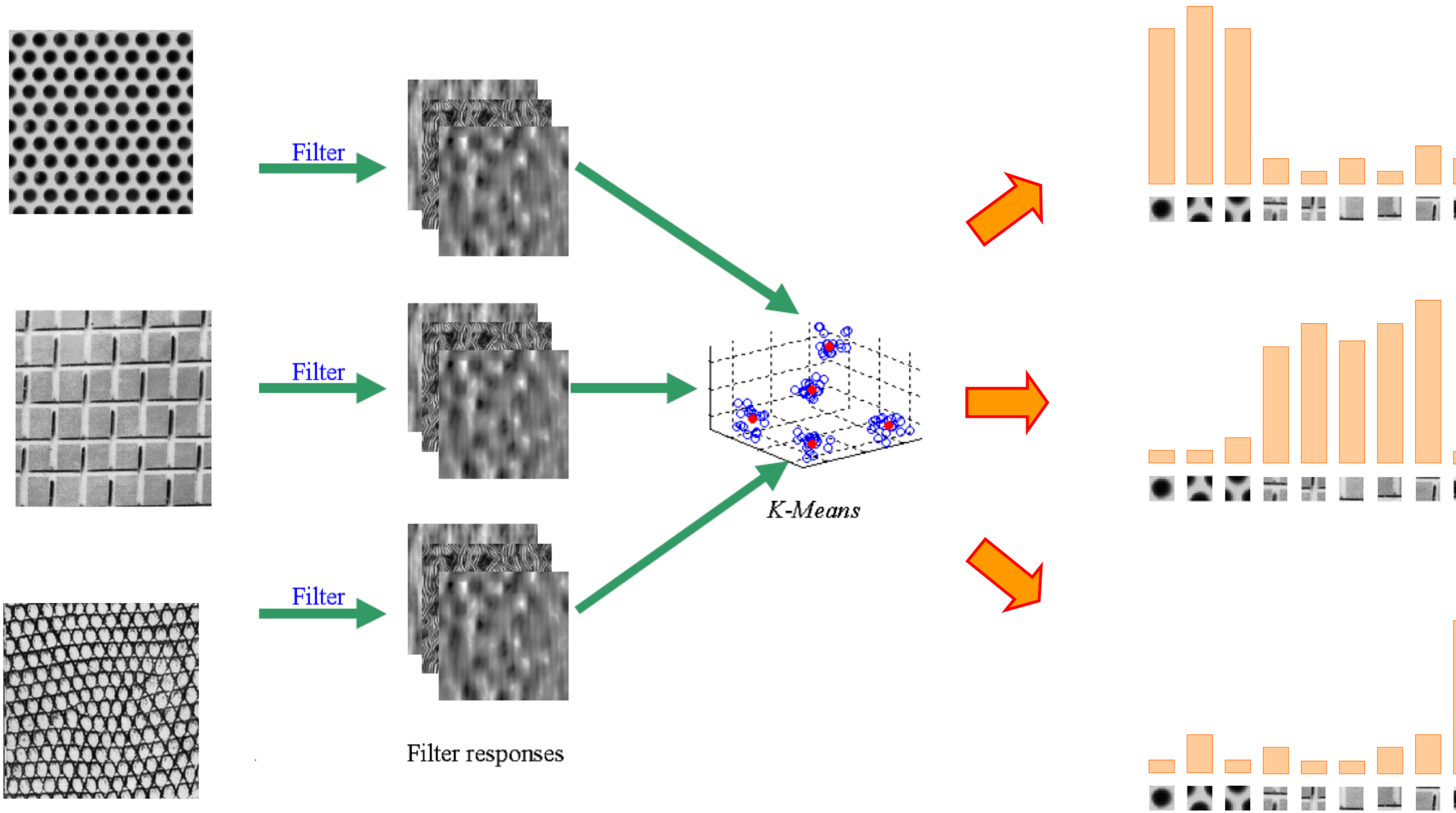
Intuitively, these 48 numers capture "verical barness", "diagonal stripeness", etc.. of original image patch

If I represent filter bank as 2500 x 48 matrix, how do we perform this conversion?

vector-matrix multipication

Overall pipeline

Handy implementation trick: efficiently linear project all overlapping patches in an image with *filtering*



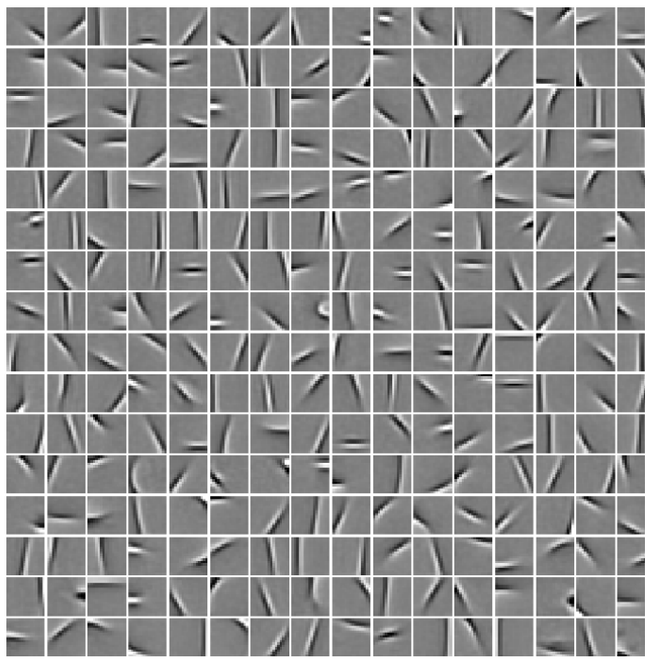
Learning Feature Representations with K-means

Adam Coates and Andrew Y. Ng

Stanford University, Stanford CA 94306, USA
{acoates,ang}@cs.stanford.edu

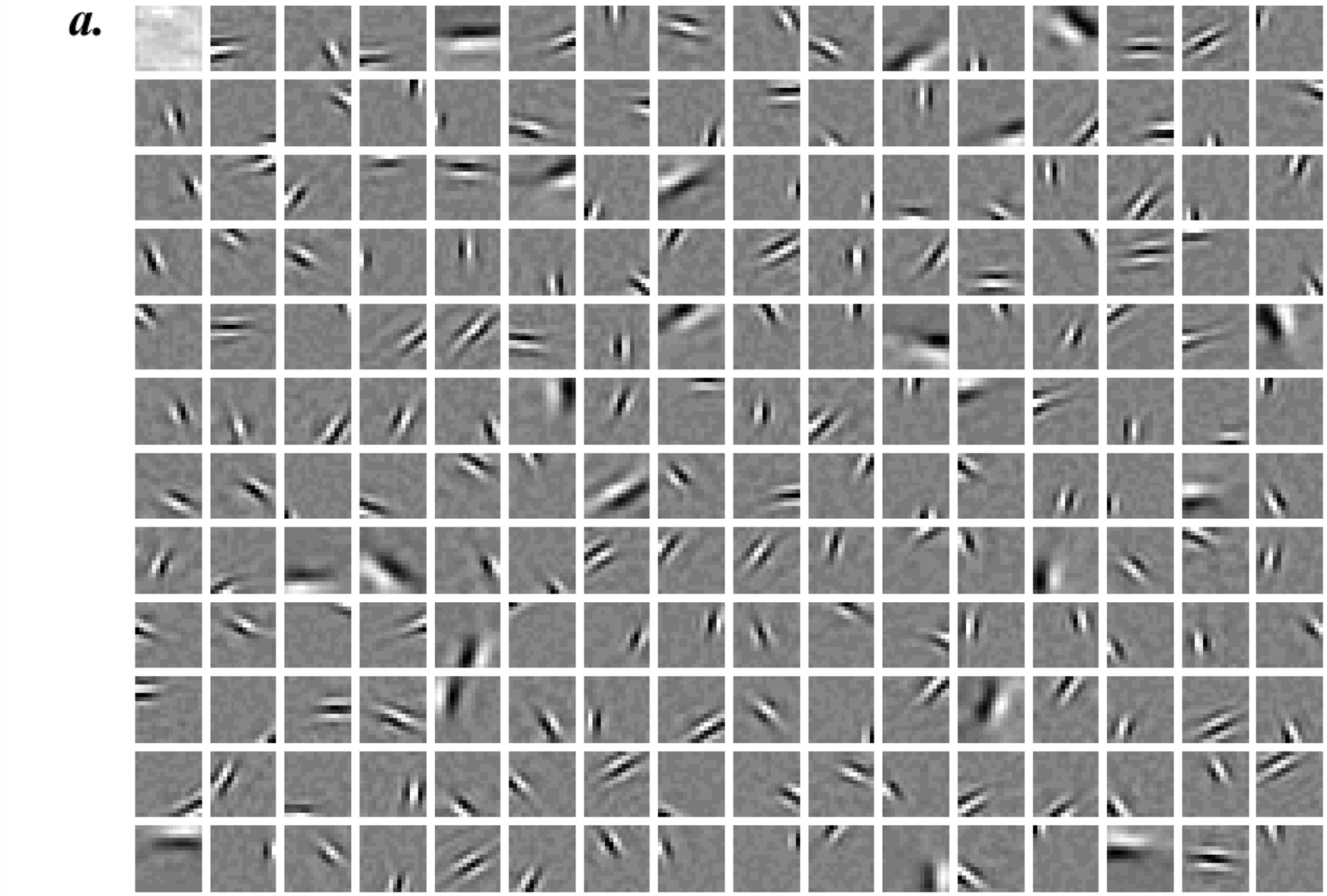
Originally published in: G. Montavon, G. B. Orr, K.-R. Müller (Eds.), Neural Networks: Tricks of the Trade, 2nd edn, Springer LNCS 7700, 2012.

Early precursor to learning representations with deep networks



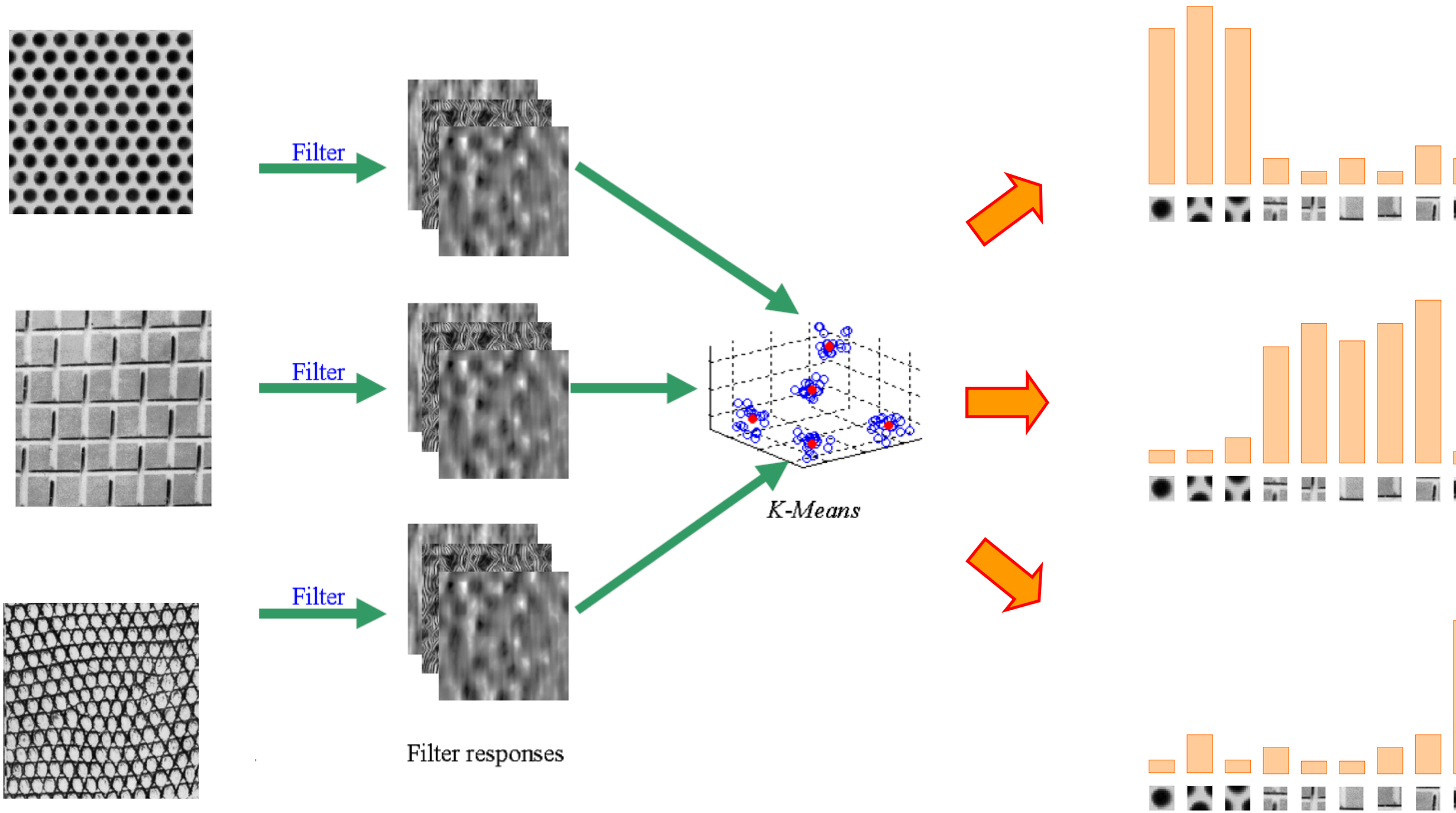
[Aside: use statistical tricks like principle component analysis or whitening]

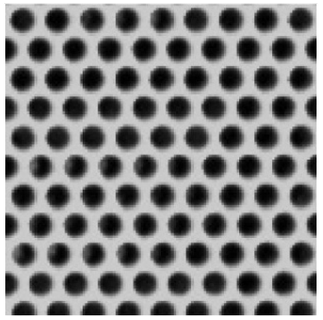
Learning dictionaries seems on natural images seems to produce gabor-like patches!



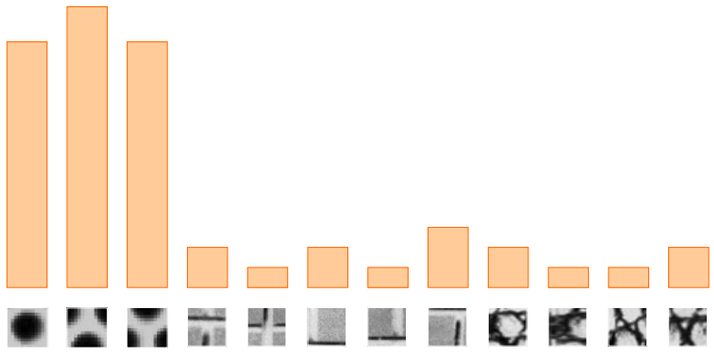
- **Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images.**
  Olshausen BA, Field DJ (1996). *Nature*, 381: 607-609

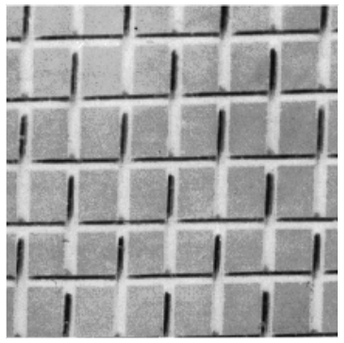
[Aside: use statistical tricks like sparse regularization[


Filter
Filter
Filter
K-Means
Filter responses

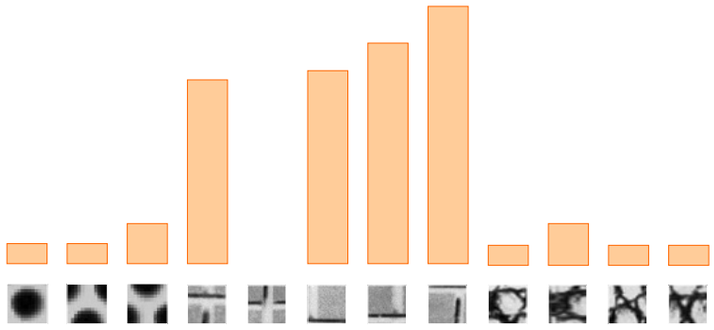


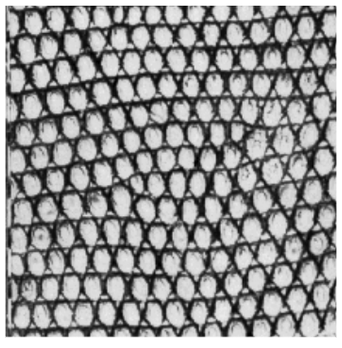
polka-dot



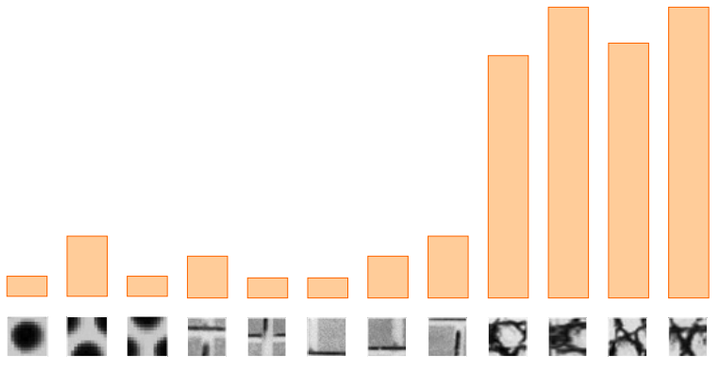


brick

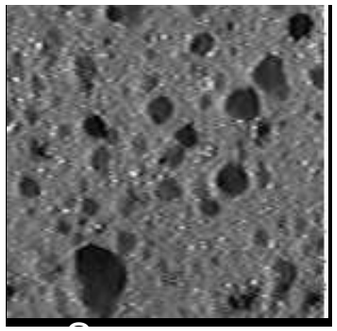


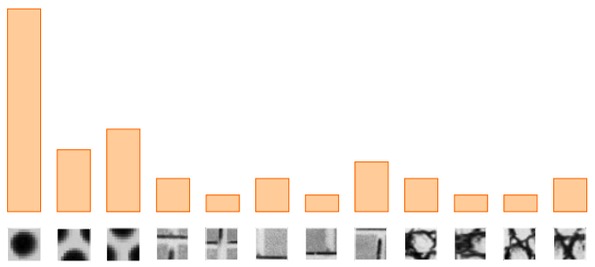


mesh

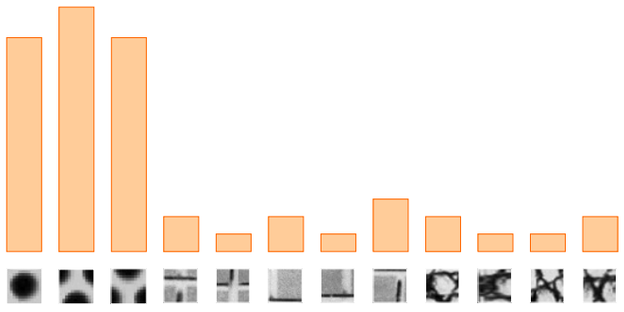


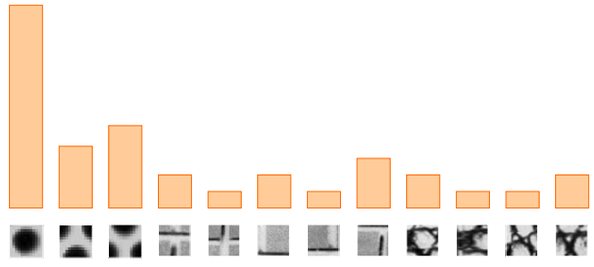
match





What's the "right" way to compare histograms?

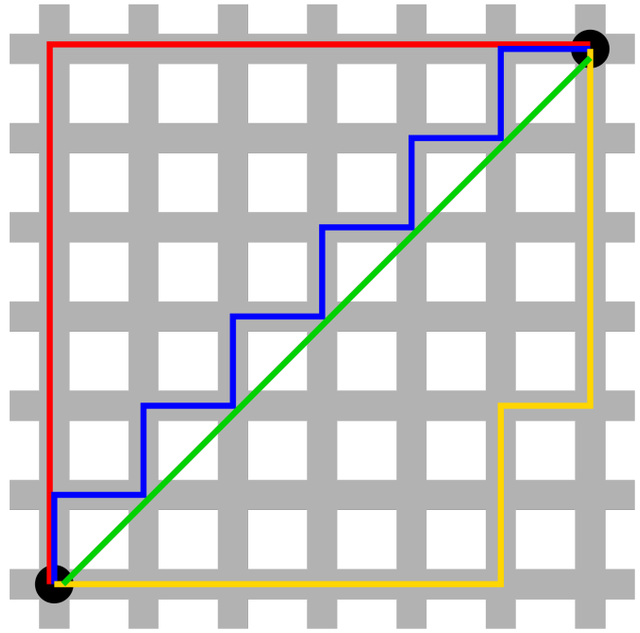




L2 (euclidean distance)?
L1 distance?

$$Euc(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$$

$$Manhattan(x,y) = \sum_i |x_i - y_i|$$



Notational switch:
x = histogram for image1, x_i = count for bin i.
y = histogram for image2, y_i = count for bin i

Turns out there is a lot of histogram “distances”!

Euclidean (L2 distance)

$$Euc(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$$

L1 distance

$$Manhattan(x,y) = \sum_i |x_i - y_i|$$

chi-squared distance
[derived from chi-squared test in statistics]

$$Chi(x,y) = \sum_i \frac{(x_i - y_i)^2}{x_i + y_i}$$

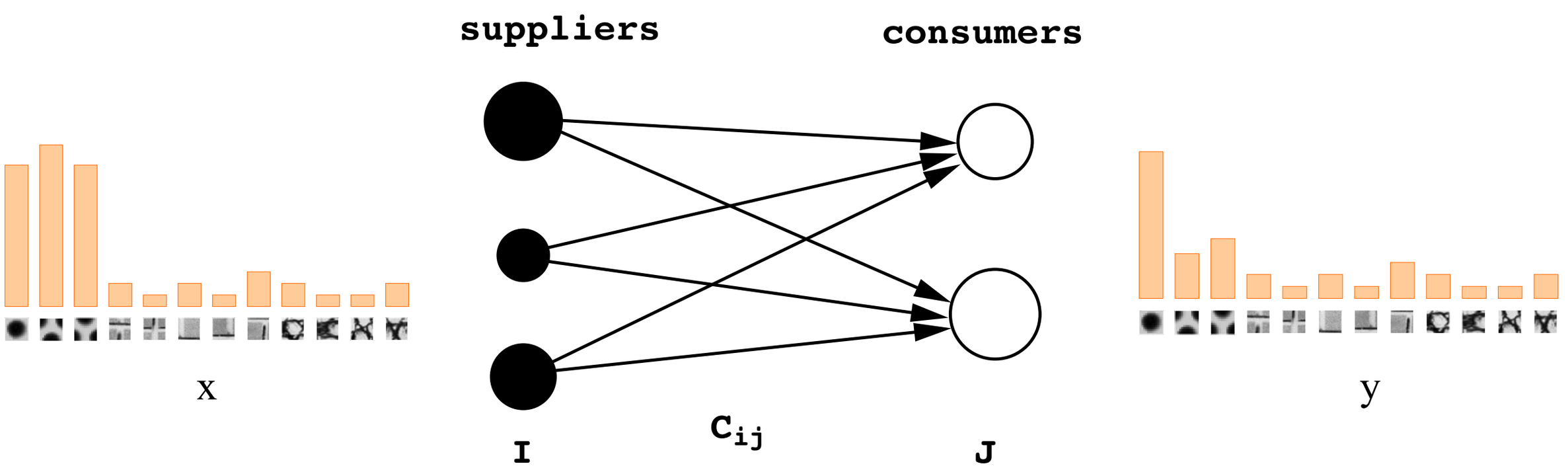
K-L divergence
[log probability of seeing x under model y]

$$D_{KL}(x,y) = \sum_i x_i \log \frac{x_i}{y_i}$$

chi-squared intuition: (100vs99) is less different than (1vs0)

A “gold standard”: Earth mover’s distance

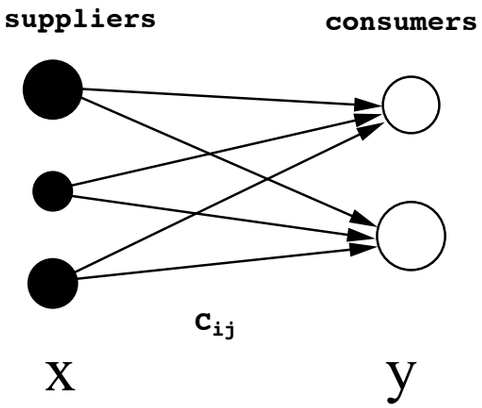
- All existing measures fail to account for similarity between bin i and bin j (written as say, c_{ij})
- Cast as transport problem that seeks to find low-cost soln for moving “dirt” from xbins to closeby ybins



Earth mover's distance

Bipartite network flow

$$\min_{f_{ij}} \sum_{ij} c_{ij} f_{ij} \quad s.t.$$



$$f_{ij} \geq 0$$

$$\sum_i f_{ij} = y_j \qquad \text{“total dirt entering target bin j = } y_j\text{”}$$

$$\sum_j f_{ij} = x_i \qquad \text{“total dirt leaving source bin i = } x_i\text{”}$$

$$EMD(\mathbf{x}, \mathbf{y}) = \sum_{ij} c_{ij} f_{ij}$$

https://www.cs.duke.edu/~tomasi/papers/rubner/rubnerTr98.pdf

Recent work: sinkhorn distances

SINKHORN DISTANCES: LIGHTSPEED COMPUTATION OF OPTIMAL TRANSPORTATION DISTANCES

MARCO CUTURI

Abstract. Optimal transportation distances are a fundamental family of parameterized distances for histograms. Despite their appealing theoretical properties, excellent performance in retrieval tasks and intuitive formulation, their computation involves the resolution of a linear program whose cost is prohibitive whenever the histograms' dimension exceeds a few hundreds. We propose in this work a new family of optimal transportation distances that look at transportation problems from a maximum-entropy perspective. We smooth the classical optimal transportation problem with an entropic regularization term, and show that the resulting optimum is also a distance which can be computed through Sinkhorn-Knopp's matrix scaling algorithm at a speed that is several orders of magnitude faster than that of transportation solvers. We also report improved performance over classical optimal transportation distances on the MNIST benchmark problem.

Neurips 2013, 2K+ citations

Modify the problem to make optimization really fast on GPUs

Similarity Kernals

[sometimes more intuitive to define similarity rather than distance]

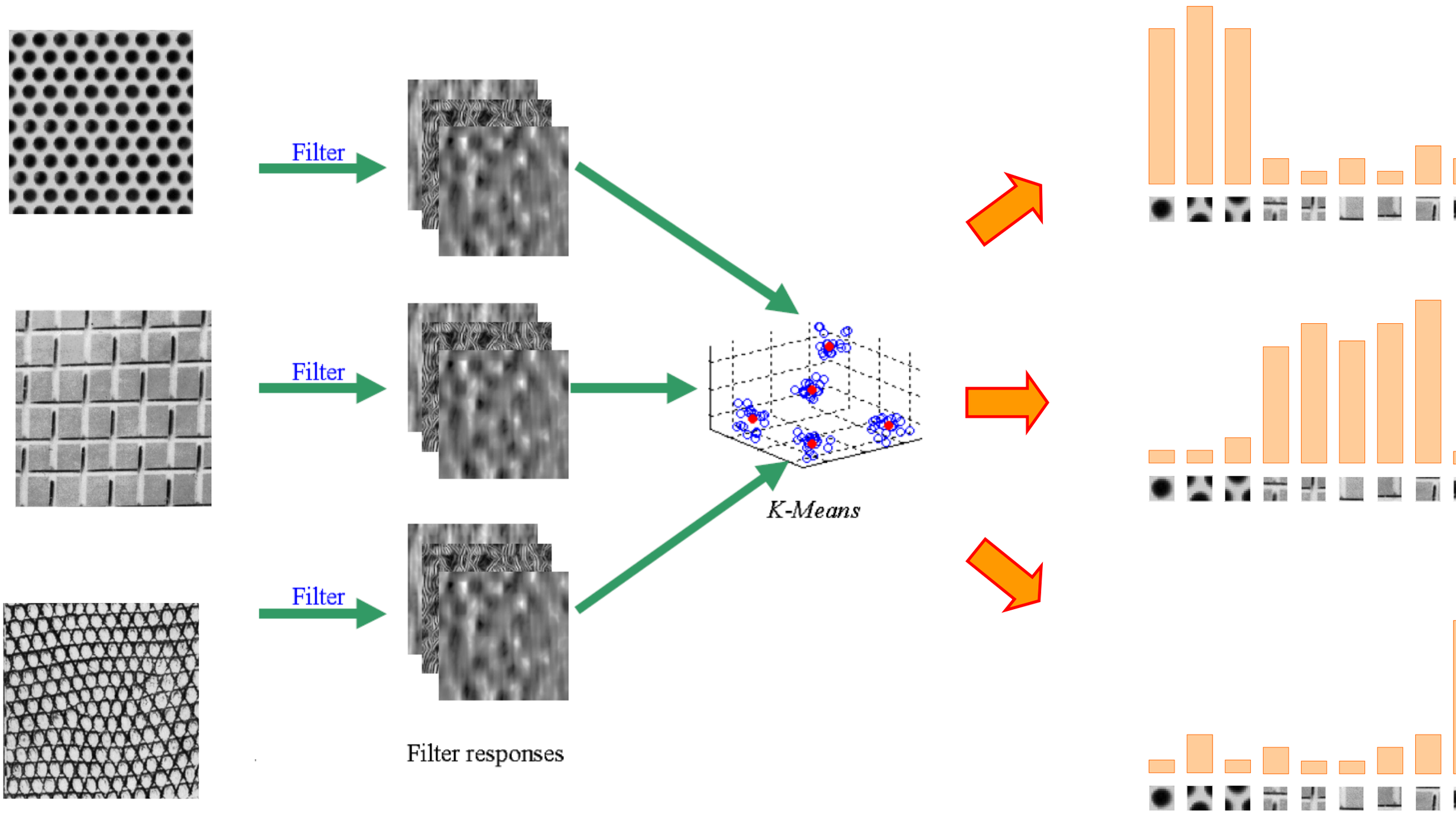
$$||x - y||^2 = x^T x + y^T y - 2x^T y$$

$$D(x, y) = K(x, x) + K(y, y) - 2K(x, y)$$

$$K_{lin}(x, y) = \sum_i x_i y_i$$

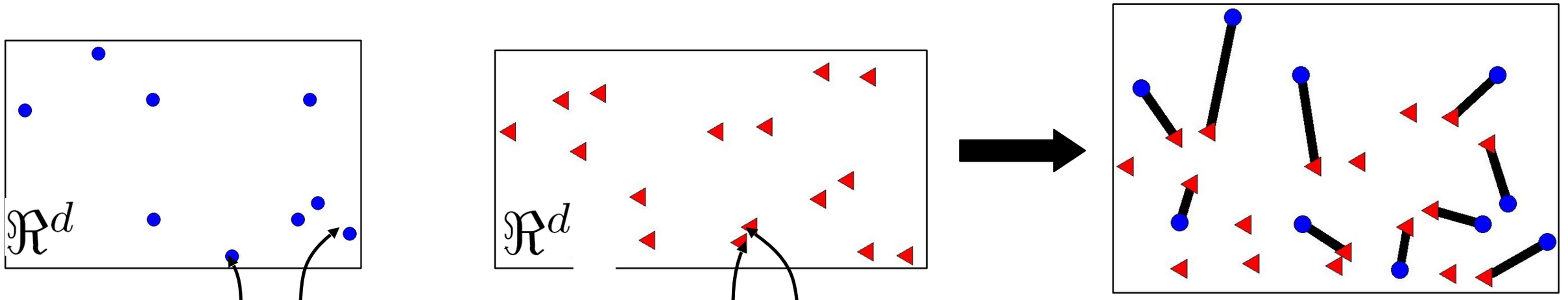
[what's corresponding distance function?]

Recall texture pipeline



Alternative to quantization:
when computing similarity between a query and training image pair, explicitly find matching image patches

Grauman & Darrell



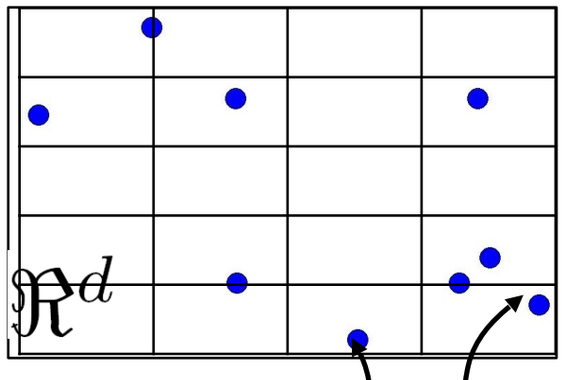
$$\max_{\pi:\mathbf{X}\to\mathbf{Y}} \sum_{\mathbf{x}_i\in\mathbf{X}} \mathcal{S}(\mathbf{x}_i, \pi(\mathbf{x}_i))$$

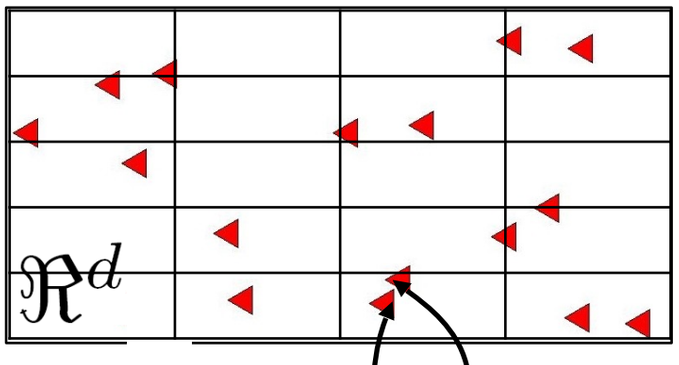
$$\mathbf{X} = \{\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_m\}$$
$$\vec{\mathbf{x}}_i \in \Re^d$$

$$\mathbf{Y} = \{\vec{\mathbf{y}}_1, \ldots, \vec{\mathbf{y}}_n\}$$
$$\vec{\mathbf{y}}_i \in \Re^d$$

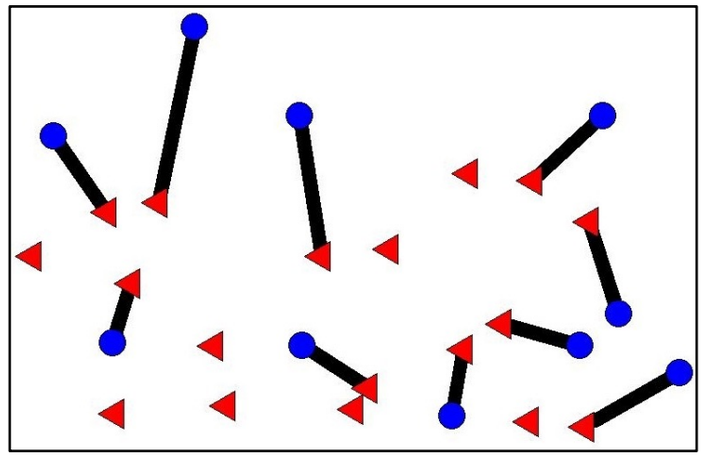
Let's approximate matching by binning

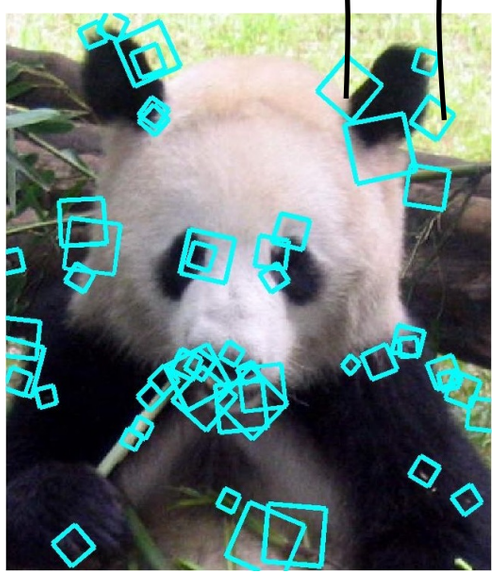
But wait a sec…. didn't we want to move *away* from quantization?
[We'll get there!]





$\approx$



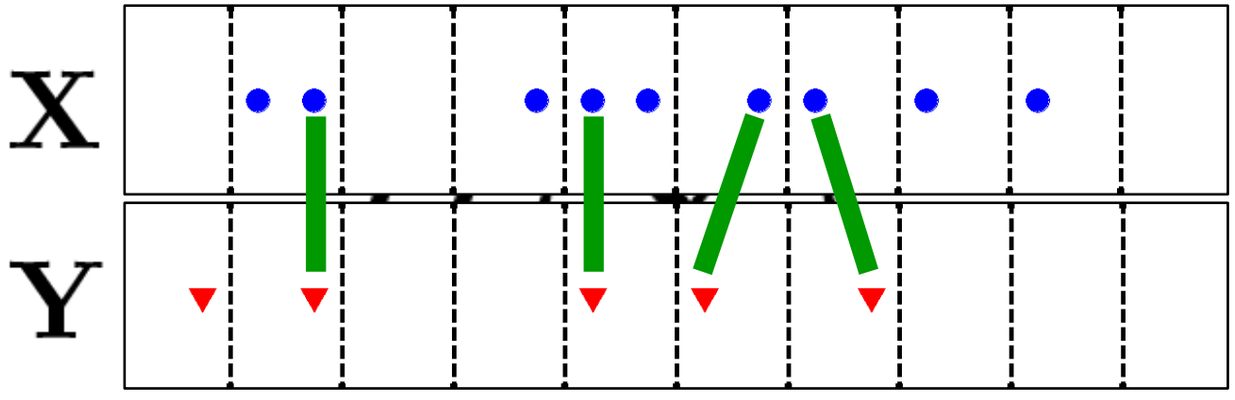


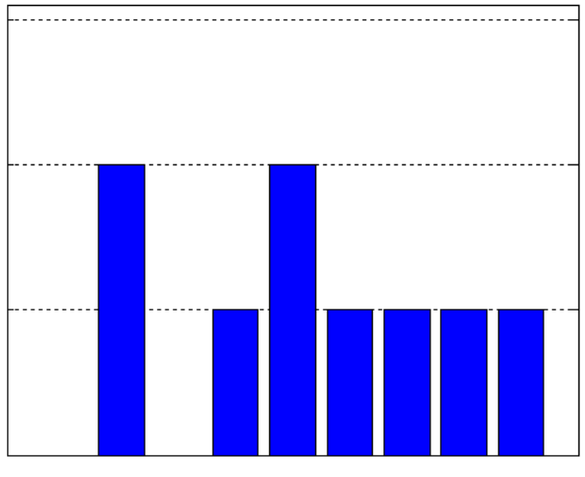
$$\mathbf{X} = \{\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_m\}$$
$$\vec{\mathbf{x}}_i \in \Re^d$$

$$\mathbf{Y} = \{\vec{\mathbf{y}}_1, \ldots, \vec{\mathbf{y}}_n\}$$
$$\vec{\mathbf{y}}_i \in \Re^d$$

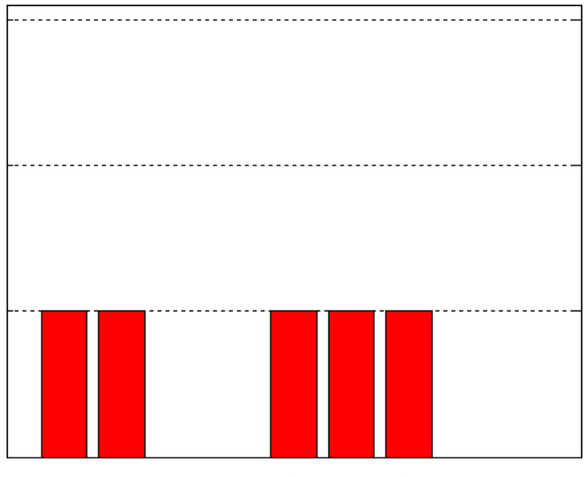
Histogram intersection kernal

$$\mathcal{I}(H(\mathbf{X}), X(\mathbf{Y})) = \sum_k \min(H(\mathbf{X}_k), X(\mathbf{Y}_k))$$

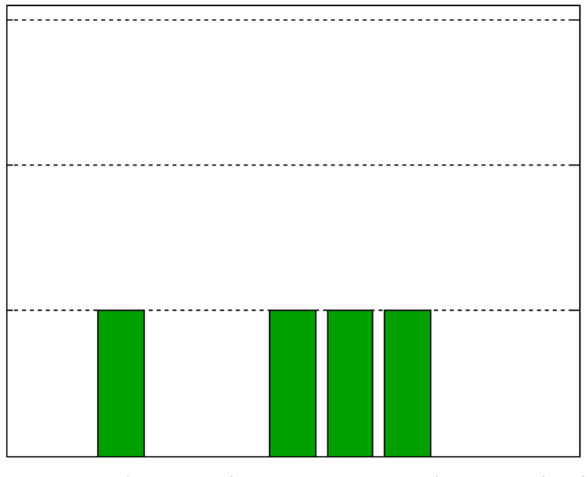




$H(\mathbf{X})$

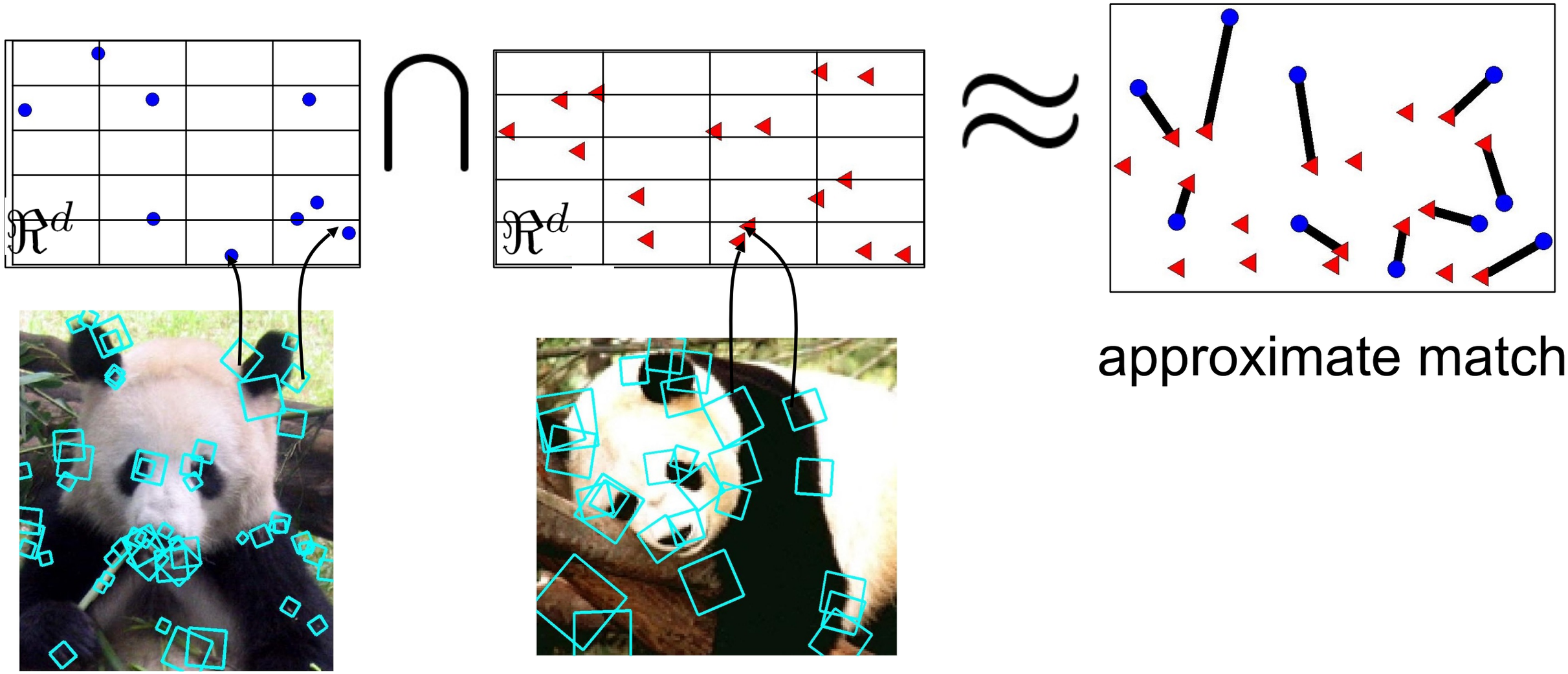


$H(\mathbf{Y})$



$\mathcal{I}\left(H(\mathbf{X}), H(\mathbf{Y})\right) = 4$

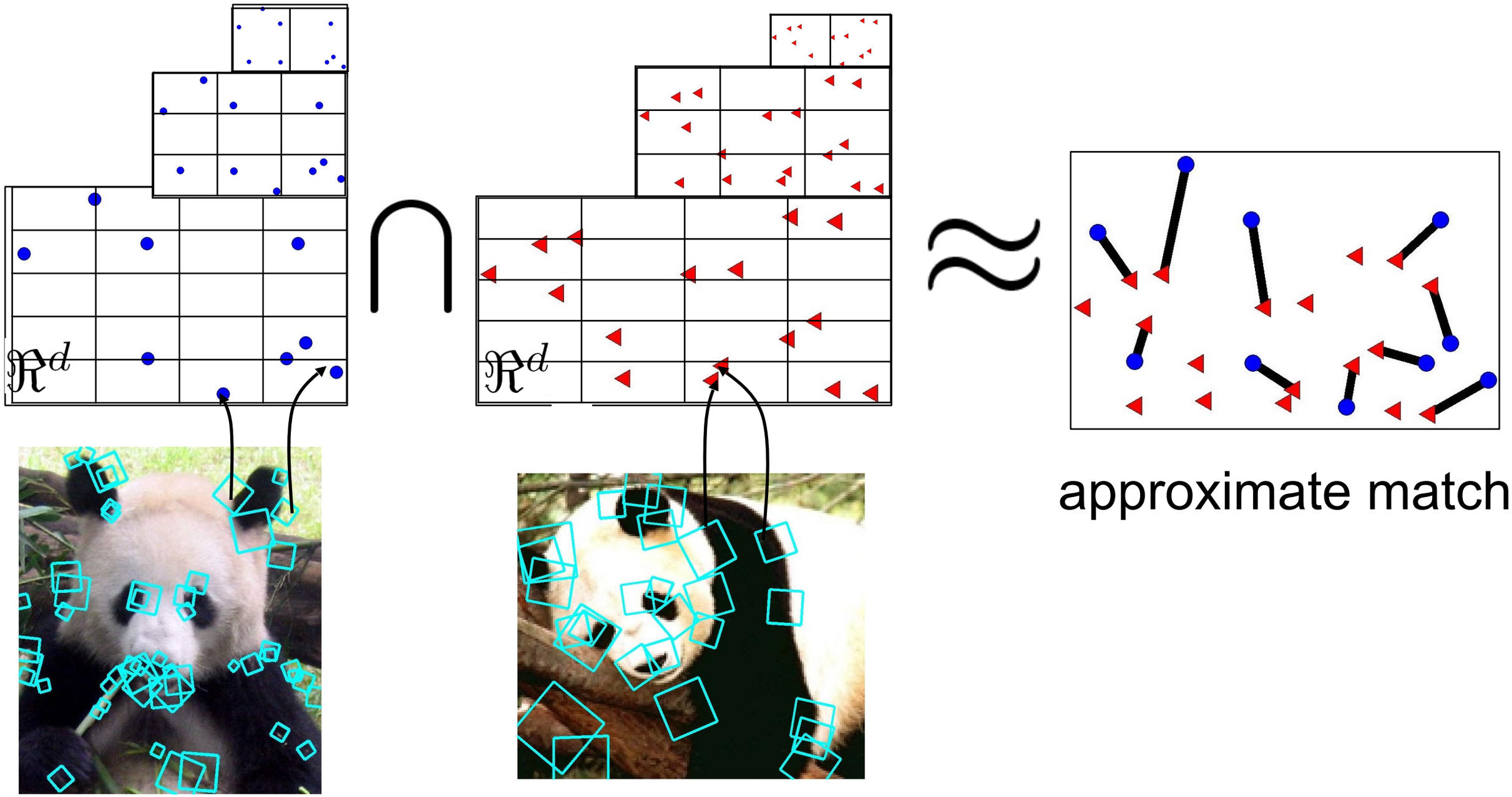
Back to correspondence matching



approximate match

But what about bin effects (partial credit for near matches)?

Back to correspondence matching



Count matches obtained from larger bins

Counting new matches

Histogram intersection

$$\mathcal{I}\left(H(\mathbf{X}), H(\mathbf{Y})\right) = \sum_{j=1}^{r} \min\left(H(\mathbf{X})_j, H(\mathbf{Y})_j\right)$$

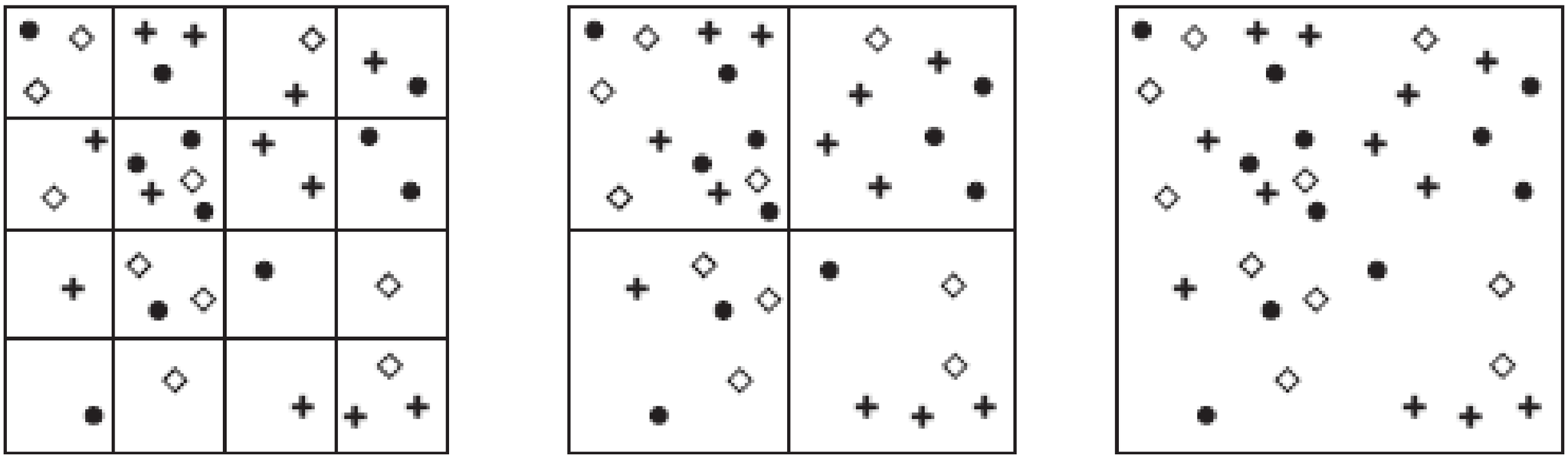
matches at this level — matches at previous level

$$N_i = \mathcal{I}\left(H_i(\mathbf{X}), H_i(\mathbf{Y})\right) - \mathcal{I}\left(H_{i-1}(\mathbf{X}), H_{i-1}(\mathbf{Y})\right)$$

Difference in histogram intersections across levels counts *number of new pairs* matched

Giving partial credit for new matches

Problem: we'll always get many more matches ith larger bins!



Weight new matches inversely porportional to bin width

$$\frac{1}{2^i}$$

Pyramid match kernel

histogram pyramids

$$K_\Delta\left(\overbrace{\Psi(\mathbf{X}), \Psi(\mathbf{Y})}\right) =$$

$$\sum_{i=0}^{L} \frac{1}{2^i} \Big(\underbrace{\mathcal{I}\left(H_i(\mathbf{X}), H_i(\mathbf{Y})\right) - \mathcal{I}\left(H_{i-1}(\mathbf{X}), H_{i-1}(\mathbf{Y})\right)}_{\text{number of newly matched pairs at level } i} \Big)$$

measure of difficulty of a match at level *i*

- Weights inversely proportional to bin size
- Normalize kernel values to avoid favoring large sets

Spatial Pyramid Matching

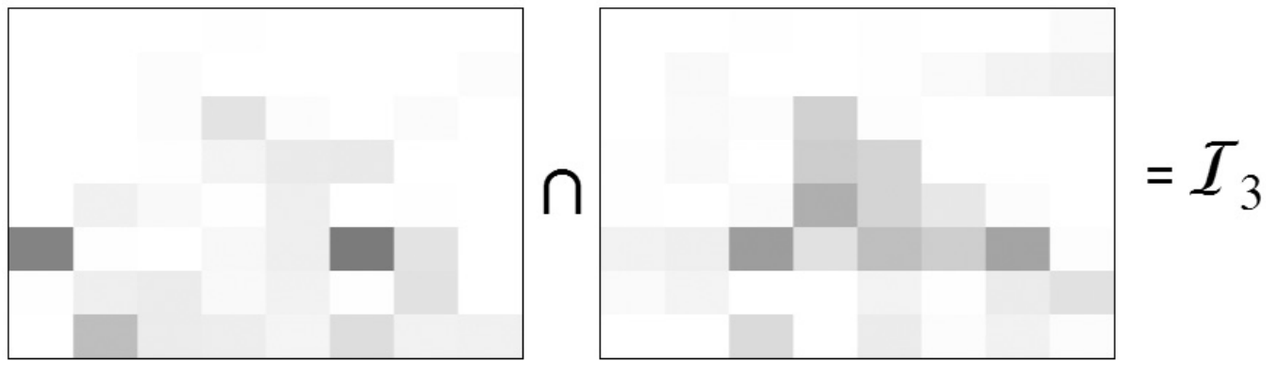
Quantize features into words, but build pyramid in space

Nifty way to encode constraints like “eye” words lie near top of image

Original images

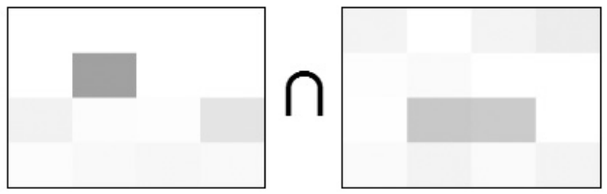
Feature histograms:

Level 3



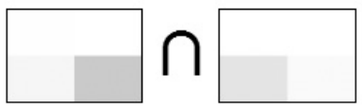
$= \mathcal{I}_3$

Level 2



$= \mathcal{I}_2$

Level 1



$= \mathcal{I}_1$

Level 0

$= \mathcal{I}_0$

Total weight (value of *pyramid match kernel*): $\mathcal{I}_3 + \frac{1}{2}(\mathcal{I}_2 - \mathcal{I}_3) + \frac{1}{4}(\mathcal{I}_1 - \mathcal{I}_2) + \frac{1}{8}(\mathcal{I}_0 - \mathcal{I}_1)$

Texture

- Texture
    - Markov models
    - Textons Dictionaries
    - Sparse Coding
- Bag-of-words
    - Earth Mover’s Distance
    - Similarity Kernals
- (Spatial) pyramid matching
    - Approximate Correspondence
    - Spatial Encodings