Scaling Biologically Inspired Computer Vision Algorithms for Video Content Analysis

Tom Dean

Google

Jim Mutch

Center for Biological & Computational Learning

MIT Brain & Cognitive Sciences

Principles of Biological Vision

Principles of Computer Vision

Principles of Computer Vision I

- Localized, oriented, band-pass filters
  - *e.g.,* Gabor functions, Haar wavelets
- Adaptive extrema-seeking attention maps
  - *e.g.,* Harris corners, Laplacian operator
- Neighborhood preserving topographic maps
  - *e.g.,* retinotopy and subspace pooling

Case Study: Bag of Words

- Detect local feature coordinates:
  - random, regular grid, find interest points
- Compute feature descriptors:
  - histograms of gradients of local patches
- Vector quantize the descriptors:
  - k-means to map descriptors to clusters
- Summarize as term-frequency vector:
  - relationships among descriptors are lost

SIFT, SURF, GLOH, *etc.*

- Resize the image if necessary ‡
- Generate scale-space pyramid ‡
- Laplacian differential operator ‡
- Find local extrema in scale space ‡
- Orient the interest-point frame
- Gabor-wavelet decomposition ‡
- Compress resulting descriptors

‡ Operations that can be accelerated by using either CuBLAS or CuFFT.

Principles of Computer Vision II

- Local generalized-contrast normalization
  - *e.g.,* luminance gain control in the retina
- Saturating non-linear transfer functions
  - *e.g.,* thresholding, half-wave rectification
- Efficient-distributed representations
  - *e.g.,* sparse coding, vector quantization

Case Study: Sparse Coding

Reconstruct X as a linear combination of B

$$B^* = \arg\min_B \langle \min_A \|X - AB\|_2^2 + \lambda S(A) \rangle$$

where

- X is a matrix whose columns are flattened patches,
- B is a matrix of basis vectors with same dimension,
- A is a matrix of reconstruction coefficients, and
- S is a penalty function that encourages *sparsity.*

Analysis-Synthesis Iteration

Analysis step: solve for B holding A constant

$$\text{minimize}_B \; J(B|A) = \|X - AB\|_2^2$$

subject to $\sum_{i=1}^{L} B_{i,j}^2 < c$

Synthesis step: solve for A holding B constant

$$\text{minimize}_A \; J(A|B) = \|X - AB\|_2^2 + \lambda \|A\|_1$$

David Mumford. Neuronal architectures for pattern-theoretic problems. In *Large Scale Neuronal Theories of the Brain*, pages 125–152. MIT Press, 1994.

Coordinate Descent in Jacket ‡

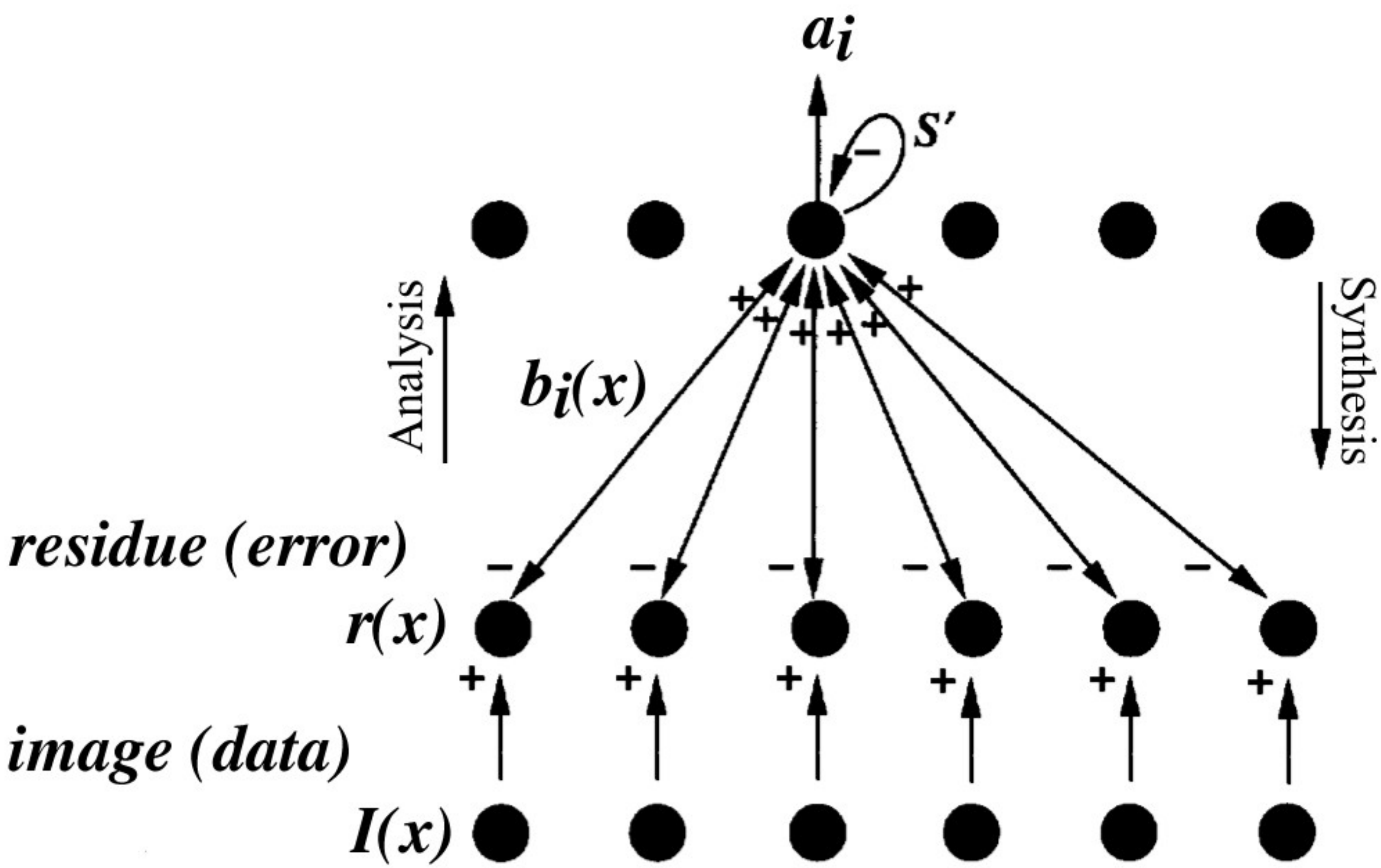
```
function X = coord_descent(A, Y)
% Determine required dimensions:
[~, num_bases] = size(A);          [~, num_cases] = size(Y);
% Initalize the coefficients:
X = zeros(num_cases,num_bases);
% Specify default parameters:
max_iter = 128;        gamma = 0.95000;     tolerance = 0.000001;
% Precompute static components:
AtA = A'* A;           YtA = Y' * A;        Pj = diag(AtA)';
% Specify gradient step sizes:
alphas = [1,3e-1,1e-1,3e-2,1e-2]; num_alphas = length(alphas);
% Append no-progress step-size:
alphas_plus_zero = [alphas 0.0];  no_progress = num_alphas + 1;
% Apply coordinate descent:
for iter = 1:max_iter
  % Compute the gradient vector:
  Y_minus_Ax_t_A = YtA - X * AtA;
  Qj = Y_minus_Ax_t_A + repmat(Pj, [num_cases,1]) .* X;
  Xstar = (Qj + sign(-Qj) * gamma) ./ repmat(Pj, [num_cases,1]);
  % Zero out small coefficients:
  Xstar( abs(-Qj) < gamma ) = 0;
  % Prepare for the line search:
  D = Xstar - X;      DtAtA = D * AtA';
  av = sum(0.5 * DtAtA .* D,2);
  bv = sum(- Y_minus_Ax_t_A .* D,2);
  % Find the minimizing step size:
  minHx = gamma * norms(X,2);
  % Solve the line-search equations:
  Hx = ones(num_alphas,num_cases);
  for k = 1:num_alphas
    Hx(k,:) = av * alphas(k) * alphas(k) + ...
      bv * alphas(k) + gamma * norms(X + alphas(k) * D, 2);
  end
  [Hx, I] = min(Hx, [], 1);
  I( ~( Hx' < minHx * ( 1 - tolerance ) ) ) = no_progress;
  % Terminate loop if no progress:
  if all( I == no_progress ); break; end
  % Apply the gradient step update:
  X = X + repmat(alphas_plus_zero(I)',[1,num_bases]) .* D;
end
```

Exploits data parallelism via matrix multiplication.

Performs parallel coordinate descent via the use of element-wise operators.

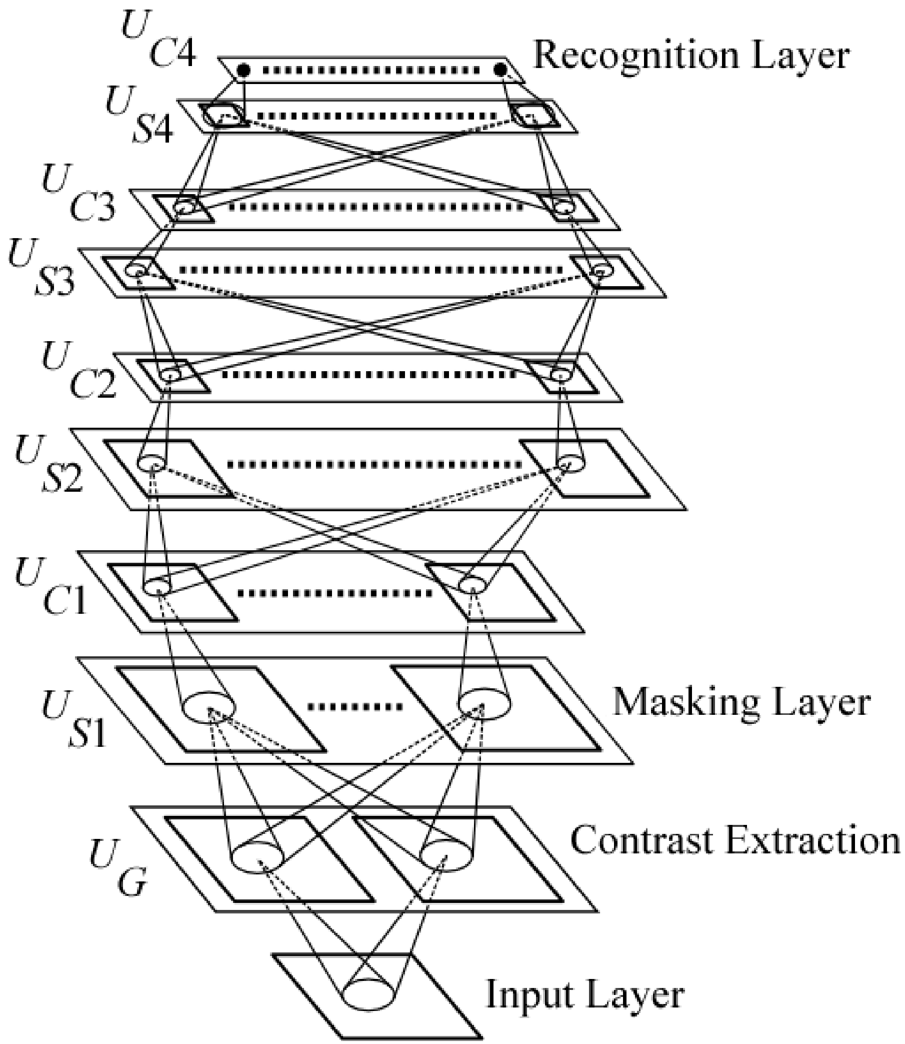
‡ Jacket is a product of AccelerEyes: www.accelereyes.com

Analysis Synthesis Network



David Mumford. Neuronal architectures for pattern-theoretic problems. In *Large Scale Neuronal Theories of the Brain*, 125−152. MIT Press, 1994.

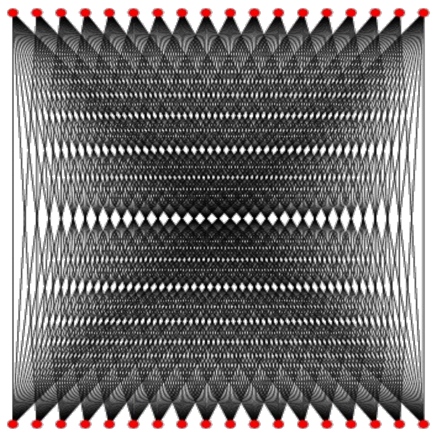
Multilayer Perceptron Models



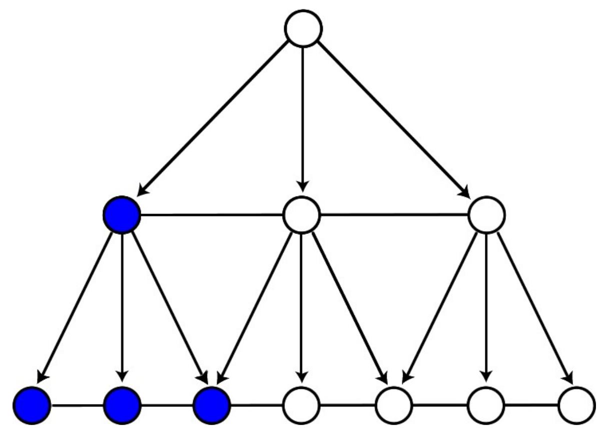
K. Fukushima. Neocognitron: A self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4):93–202, 1980.

Case Study: Deep Networks

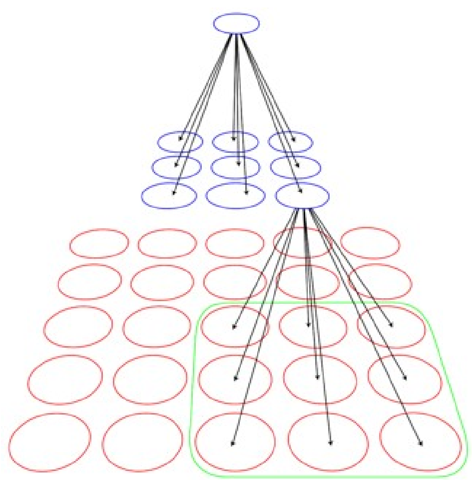
Complete Bipartite



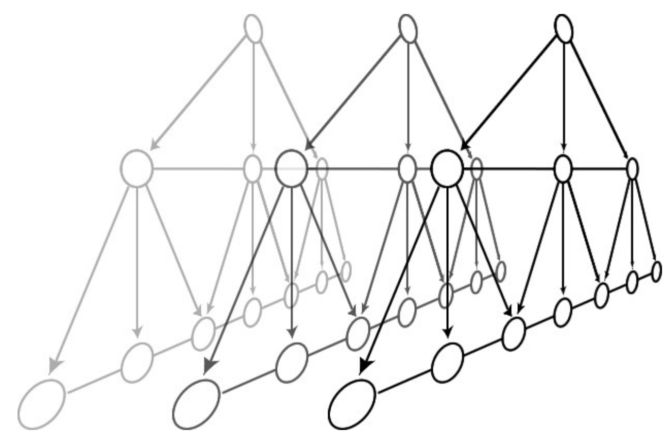
Hierarchical Structure



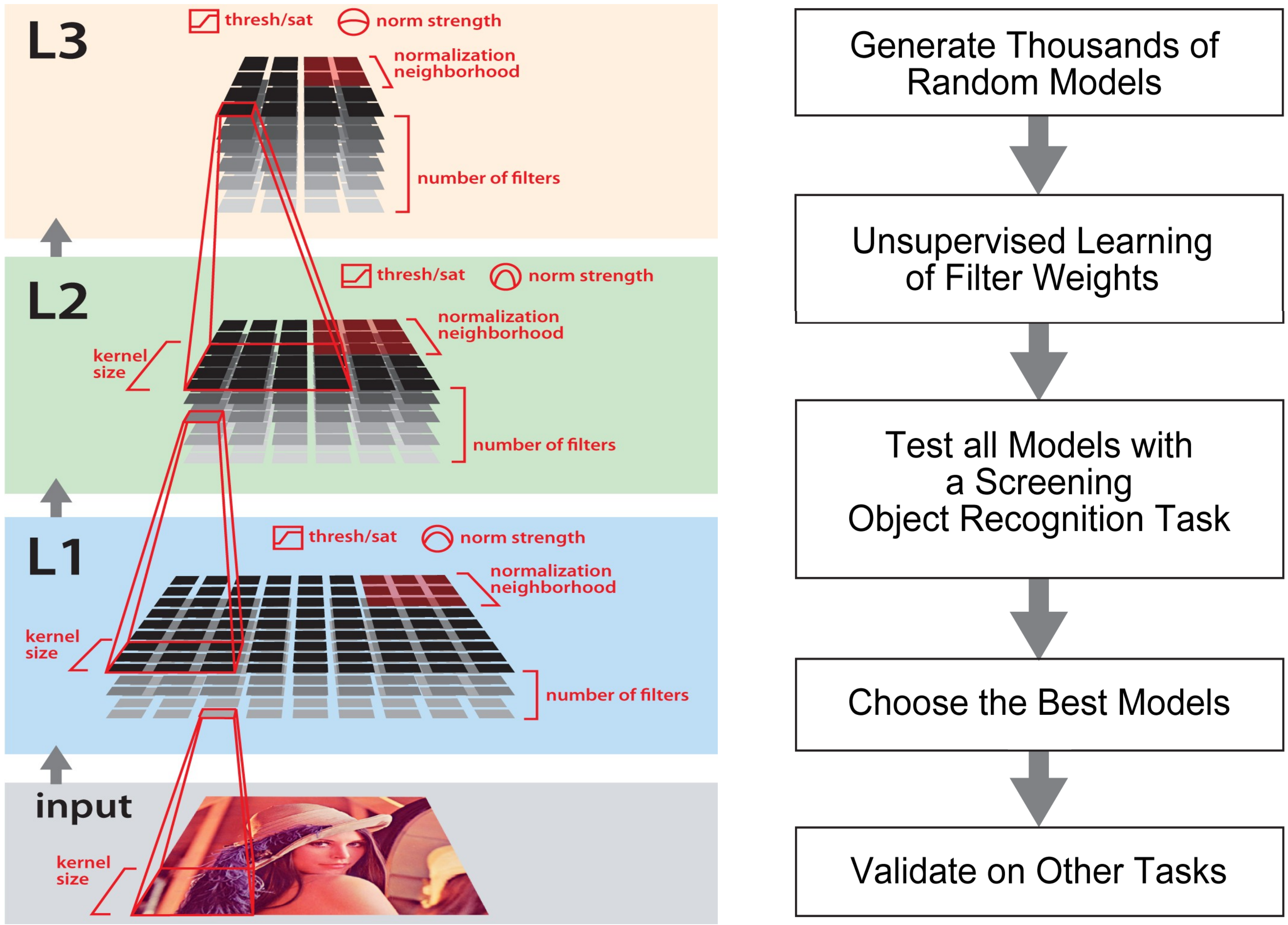
Spatial Structure



Temporal Structure

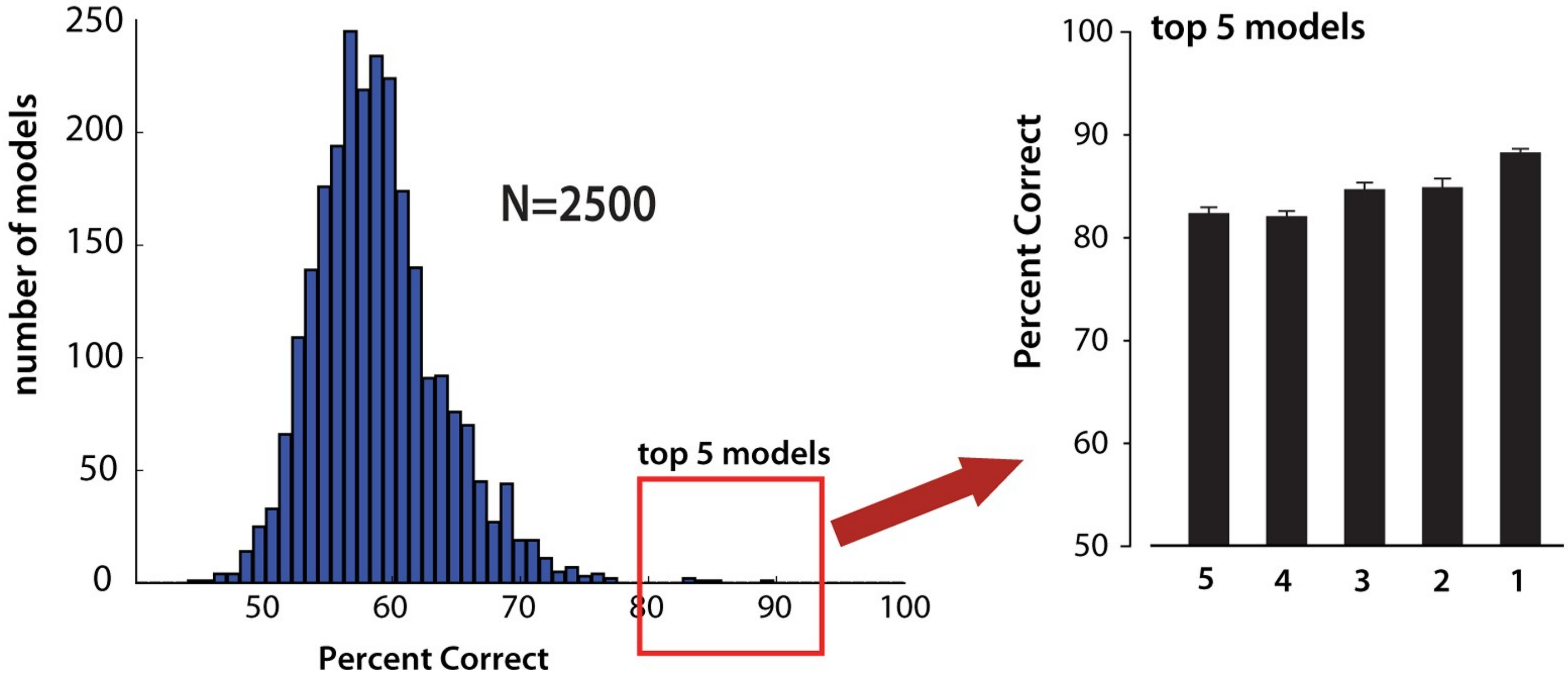


G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. Science, 313(5786):504–507, 2006.



Nicolas Pinto, David Doukhan, James DiCarlo, and David Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Computational Biology*, 5(11):e1000579, November 2009.

Searching for Top Performing Models in the Long Tail



Nicolas Pinto, David Doukhan, James DiCarlo, and David Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Computational Biology*, 5(11):e1000579, November 2009.

Fast Prototyping Frameworks

[1] O. Breuleux, J. Bergstra, J. Turian, F. Bastien, P. Lamblin, G. Desjardins, R. Pascanu, O. Delalleau, and Y. Bengio. Theano: A package for efficient computation in python. *Journal of Machine Learning Research*, under review, 2010.

[2] A. Klöeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[3] Jim Mutch, Ulf Knoblich, and Tomaso Poggio. CNS: A GPU-based framework for simulating cortically-organized networks. Technical Report MIT-CSAIL-TR-2010-013 / CBCL-286, Massachusetts Institute of Technology, Cambridge, MA, February 2010.

Fast Prototyping Frameworks

[1] O. Breuleux, J. Bergstra, J. Turian, F. Bastien, P. Lamblin, G. Desjardins, R. Pascanu, O. Delalleau, and Y. Bengio. Theano: A package for efficient computation in python. *Journal of Machine Learning Research*, under review, 2010.

[2] A. Klöeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

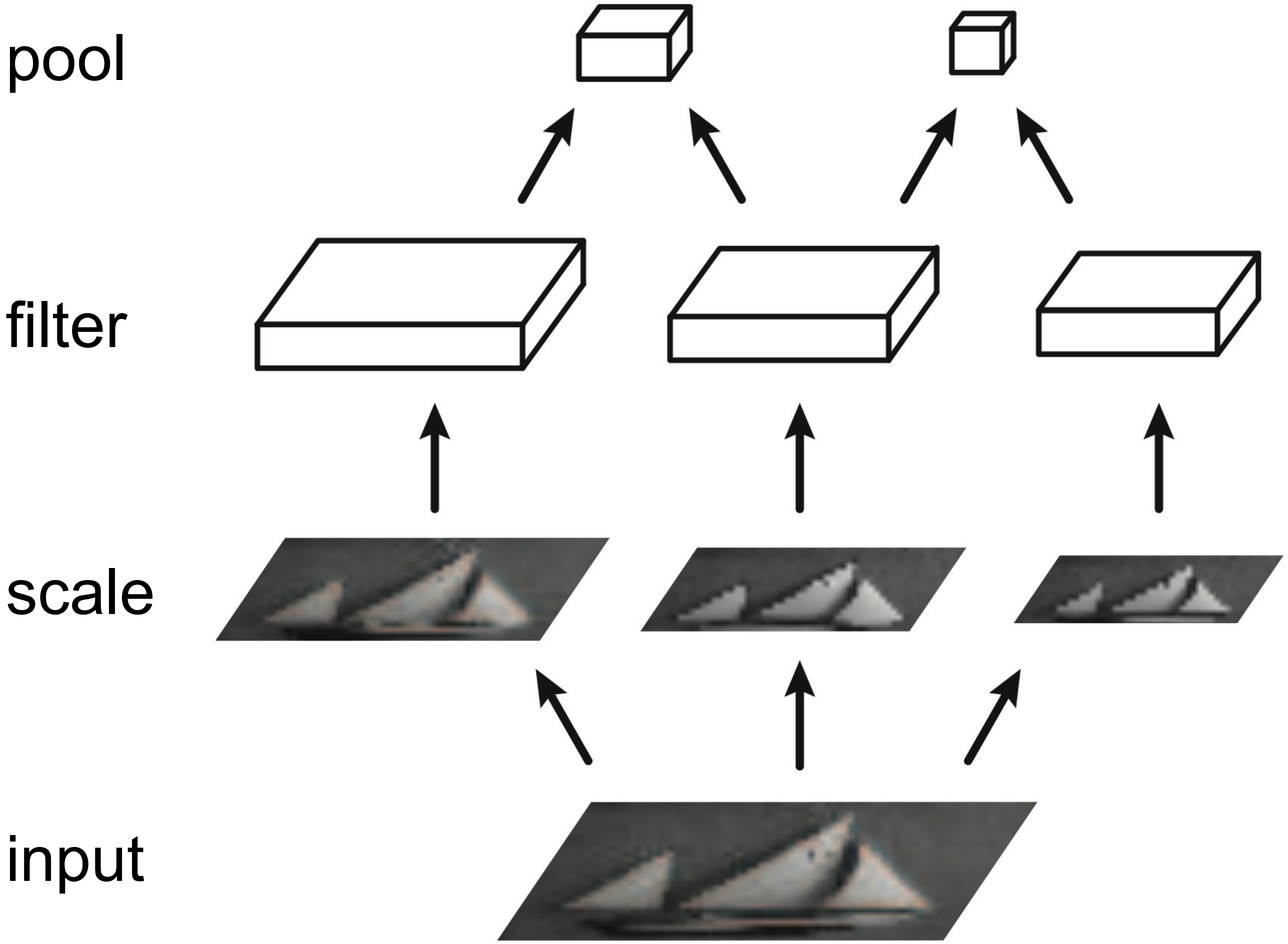
[3] Jim Mutch, Ulf Knoblich, and Tomaso Poggio. CNS: A GPU-based framework for simulating cortically-organized networks. Technical Report MIT-CSAIL-TR-2010-013 / CBCL-286, Massachusetts Institute of Technology, Cambridge, MA, February 2010.

- Define a network as a MATLAB struct:
    - the number and type of layers,
    - the dimensionality and size of layers.
    - the connectivity of layers and cells, and
    - the initial value of layer-specific variables.
- The only procedural code you write (in C++) is that executed by a single cell.
- Cell code calls macros to read/write the cell's variables, find other cells, and read other cell's variables.
    - This makes it possible to compile a network for a CPU or a GPU.
- Details of what cells are connected to other cells, how memory is organized, *etc.* are all handled by the framework.

```
m = struct;
m.layers{1}.type = ‘ndp’;
m.layers{1}.size = {100 100 50};
…
m.layers{2}.type = ‘max’;
m.layers{2}.size = {30 30 50};
…
```

```
// Code to compute one cell's response.

// Retrieve the filter size.
int ySize = WEIGHT_Y_SIZE(WZ);
int xSize = WEIGHT_X_SIZE(WZ);

// Find cell's RF in the previous layer.
GET_LAYER_Y_RF_NEAR(PZ, ySize, y1, y2);
GET_LAYER_X_RF_NEAR(PZ, xSize, x1, x2);

// Compute RF's response to the filter.
float v = 0.0f;
for (int j = xSize - 1, x = x1; j >= 0
for (int i = ySize - 1, y = y1; i >= 0
  float p = READ_LAYER_VAL(PZ, y, x);
  float w = READ_WEIGHT_VAL(WZ, i, j,
...
```

Case Study: Simple Network



Define and Run a CNS Model

```
%****************************************************

m.layers{1}.type    = 'input';
m.layers{1}.pz      = 0;
m.layers{1}.size{1} = 1;
m = cns_mapdim(m, 1, 'y', 'pixels', 256);
m = cns_mapdim(m, 1, 'x', 'pixels', 256);

m.layers{2}.type    = 'scale';
m.layers{2}.pz      = 1;
m.layers{2}.size{1} = 1;
m = cns_mapdim(m, 2, 'y', 'scaledpixels', 256, 2);
m = cns_mapdim(m, 2, 'x', 'scaledpixels', 256, 2);

m.layers{3}.type    = 'filter';
m.layers{3}.pz      = 2;
m.layers{3}.rfCount = 11;
m.layers{3}.fParams = {'gabor', 0.3, 5.6410, 4.5128};
m.layers{3}.size{1} = 4;
m = cns_mapdim(m, 3, 'y', 'int', 2, 11, 1);
m = cns_mapdim(m, 3, 'x', 'int', 2, 11, 1);

%****************************************************

% Instantiate the above model in GPU memory:
cns('init', m);

% Read test image and load it into the model:
input = imread('ketch_0010.jpg');

% This allows you to SET the value of a layer:
cns('set', 1, 'val', input);

% For this model, RUN implies feedforward pass:
cns('run');

% This allows you to GET the value of a layer:
output = cns('get', 3, 'val');

% Relinquish hold and free up device memory:
cns('done');

%****************************************************
```

A CNS Cell Type: Definition

```
%***********************************************************************

function varargout = demopkg_cns_type_filter(method, varargin)

[varargout{1 : nargout}] = feval(['method_' method], varargin{:});

%***********************************************************************

function p = method_props

p.methods = {'initlayer'};

p.blockYSize = 16;
p.blockXSize = 8;

%***********************************************************************

function d = method_fields

d.fVals = {'ga', 'private', 'cache', ...
    'dims'   , {1   2   1  }, ...
    'dparts', {1   1   2  }, ...
    'dnames', {'y' 'x' 'f'}};

%***********************************************************************

function m = method_initlayer(m, z)

c = m.layers{z};

switch c.fParams{1}
case 'gabor'
    c.fVals = GenerateGabor(c.rfCount, c.size{1}, c.fParams{2 : end});
otherwise
    error('invalid filter type');
end

for f = 1 : c.size{1}
    a = c.fVals(:, :, f);
    a = a - mean(a(:));
    a = a / sqrt(sum(a(:) .* a(:)));
    c.fVals(:, :, f) = a;
end

m.layers{z} = c;

%***********************************************************************
```

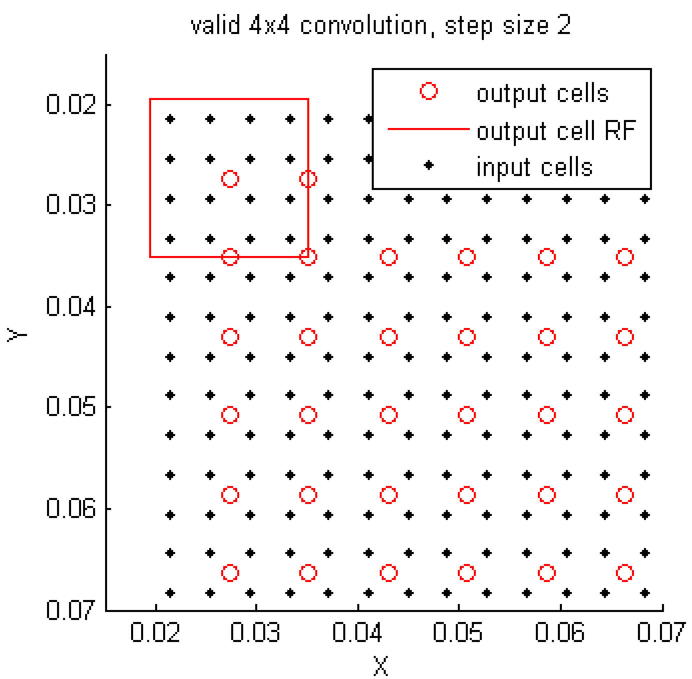
A CNS Cell Type: Kernel

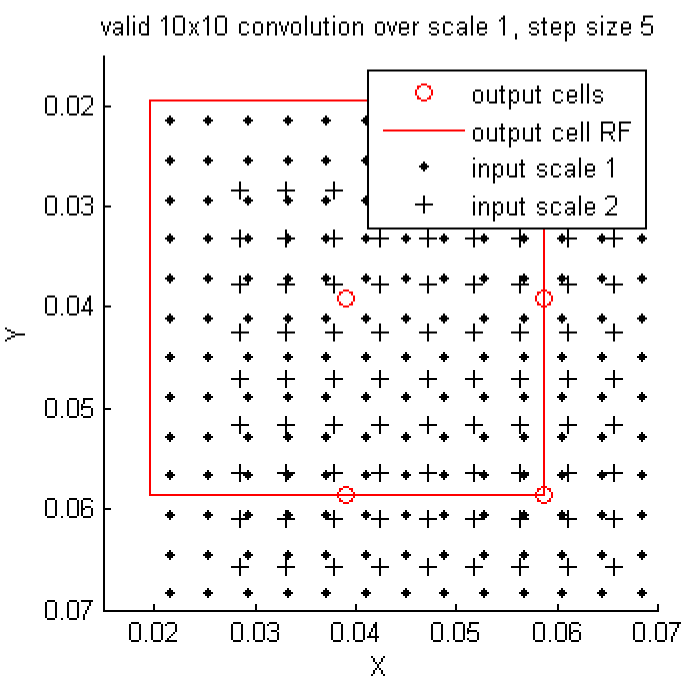
```
int y1, y2, x1, x2;
GET_LAYER_Y_RF_NEAR(PZ, FVALS_Y_SIZE, y1, y2);
GET_LAYER_X_RF_NEAR(PZ, FVALS_X_SIZE, x1, x2);

int f = THIS_F;

float res = 0.0f;
float len = 0.0f;

for (int j = 0, x = x1; x <= x2; j++, x++) {
for (int i = 0, y = y1; y <= y2; i++, y++) {


    // Read the value of the input cell:
    float v = READ_LAYER_VAL(PZ, 0, y, x);

    // Read corresponding filter value:
    float w = READ_FVALS(i, j, f);

    res += w * v;
    len += v * v;

}
}


res = fabsf(res);
if (len > 0.0f) res /= sqrtf(len);

// Write out value of this cell.
WRITE_VAL(res);
```

```
int y1, y2, x1, x2;
GET_LAYER_Y_RF_NEAR(PZ, FVALS_Y_SIZE, y1, y2);
GET_LAYER_X_RF_NEAR(PZ, FVALS_X_SIZE, x1, x2);

int f = THIS_F;

float res = 0.0f;
float len = 0.0f;

int j = 0;
#UNROLL_START 4 %x x1 <= x2
int i = 0;
#UNROLL_START 4 %y y1 <= y2

    // Read the value of the input cell:
    float v = READ_LAYER_VAL(PZ, 0, %y, %x);

    // Read corresponding filter value:
    float w = READ_FVALS(i, j, f);

    res += w * v;
    len += v * v;

i++;
#UNROLL_END
j++;
#UNROLL_END

res = fabsf(res);
if (len > 0.0f) res /= sqrtf(len);

// Write out value of this cell.
WRITE_VAL(res);
```

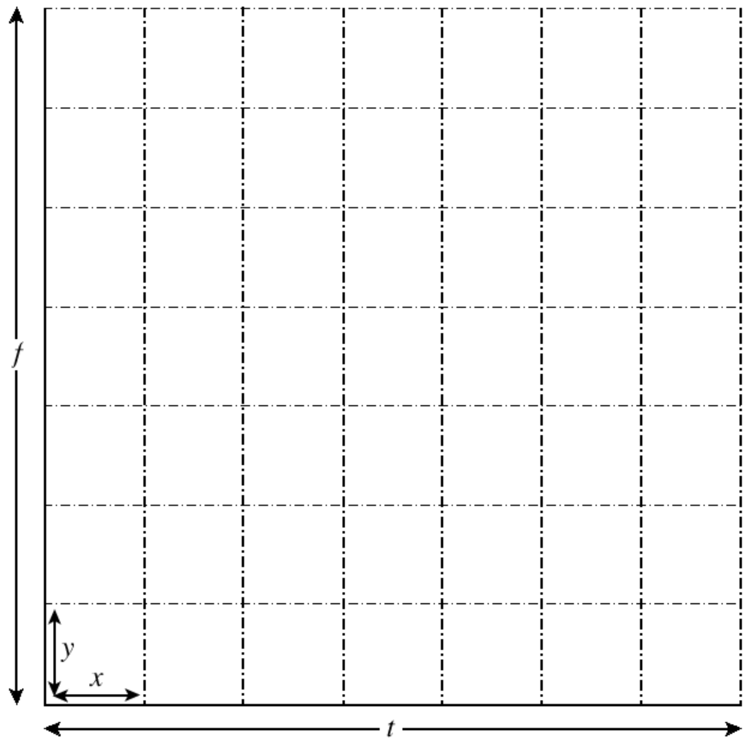
Common Coordinate Space

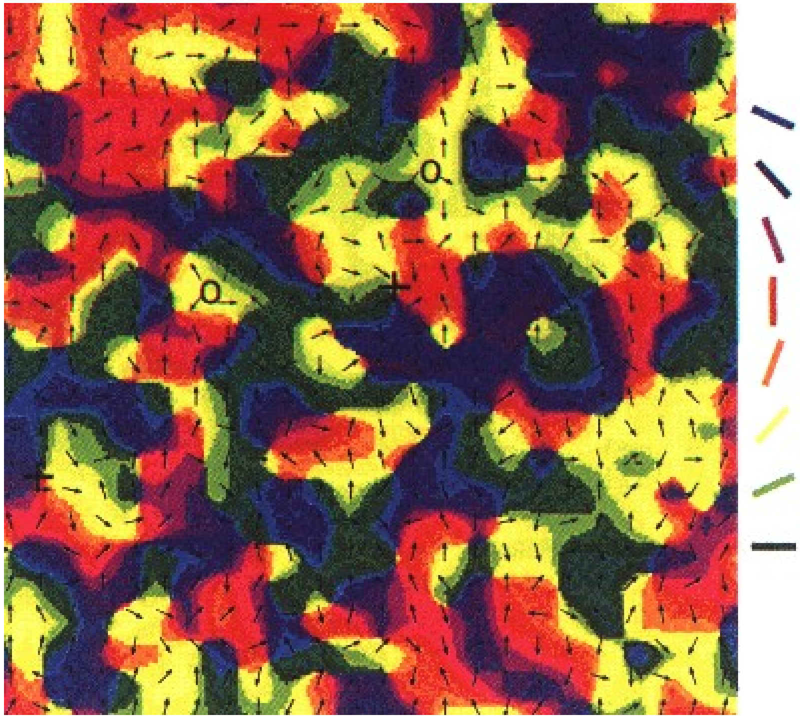
- Integer indices of cells within layers are not meaningful across layers.
- Under CNS, for topological dimensions, each cell knows its position in a real-valued coordinate space that is meaningful, *e.g.,* retinal position.
- When a cell executes, it can call CNS macros to find its *input* cells:
  - *e.g.,* find the 4 x 4 cells nearest me in layer 1,
  - *e.g.,* find all of the cells within 0.03 units of me.





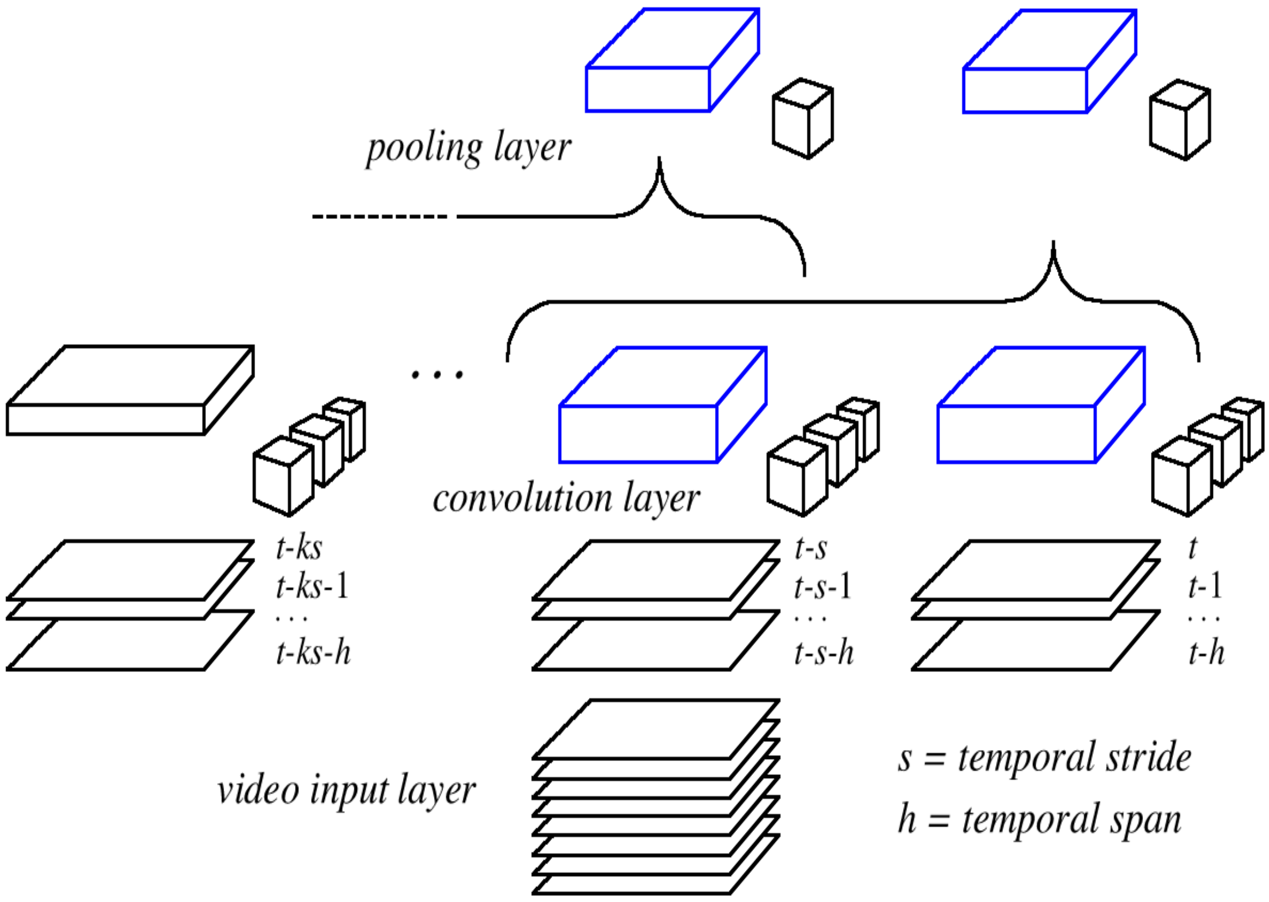
Mapping N-D to 2-D in Cortex



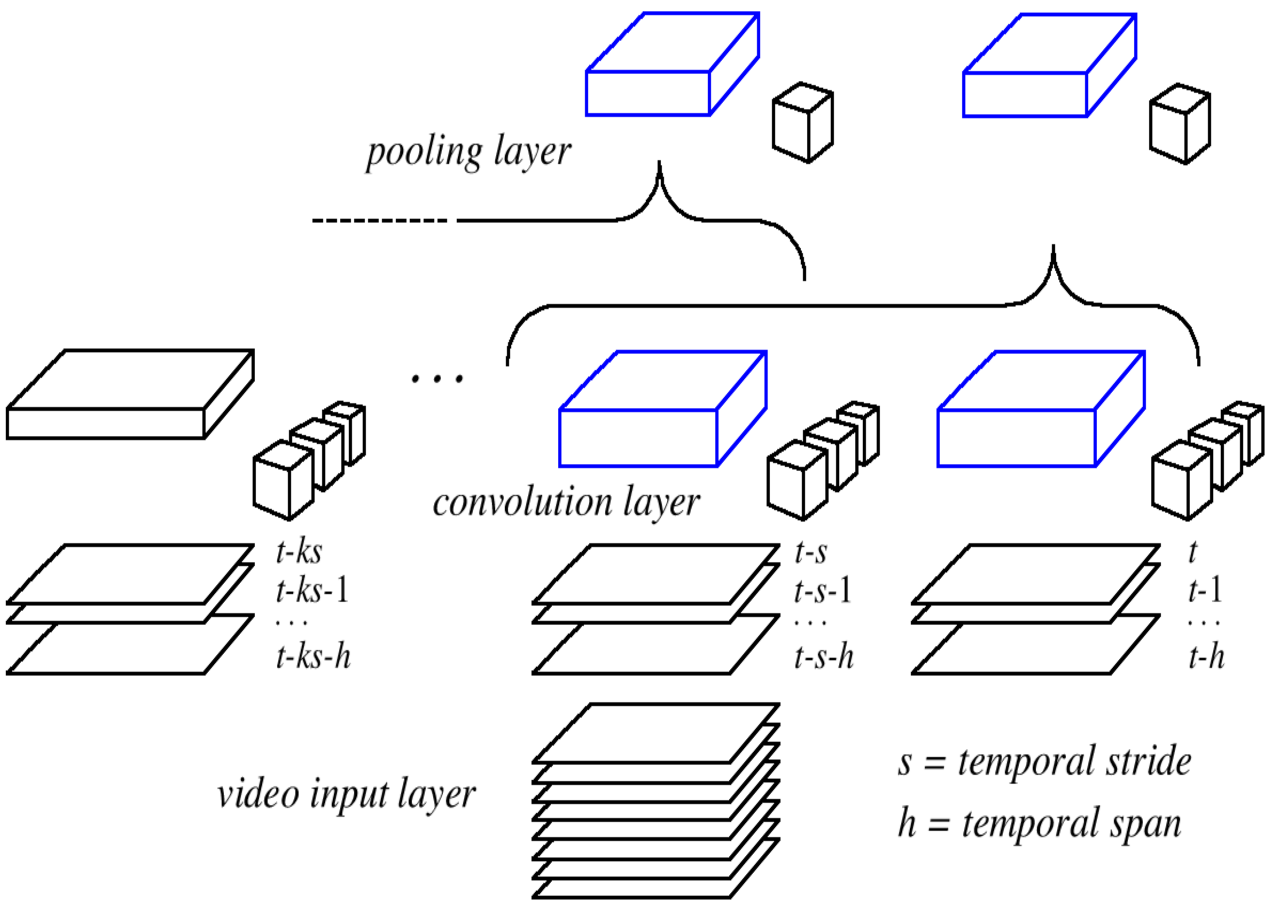


Jim Mutch, Ulf Knoblich, and Tomaso Poggio. CNS: A GPU-based Framework for Simulating Cortically-organized Networks. Technical Report MIT-CSAIL-TR-2010-013 / CBCL-286, Massachusetts Institute of Technology, Cambridge, MA, February 2010.

Case Study: Video Analysis



Case Study: Video Analysis



Define a Temporal CNS Model

```
%**************************************************

m.layers{1}.type    = 'input';
m.layers{1}.pz      = 0;
m.layers{1}.size{1} = 3;
m = cns_mapdim(m, 1, 't', 'temp1'  ,  10);
m = cns_mapdim(m, 1, 'y', 'pixels', 256);
m = cns_mapdim(m, 1, 'x', 'pixels', 256);

m.layers{2}.type    = 'norm';
m.layers{2}.pz      = 1;
m.layers{2}.tCount  = 3;
m.layers{2}.xyCount = 7;
m.layers{2}.gain    = 1;
m.layers{2}.zero    = 1;
m.layers{2}.thres   = 0.15;
m.layers{2}.size{1} = m.layers{1}.size{1};
m = cns_mapdim(m, 2, 't', 'temp2', 1, 3, 1, 8);
m = cns_mapdim(m, 2, 'y', 'int'  , 1, 7, 1);
m = cns_mapdim(m, 2, 'x', 'int'  , 1, 7, 1);

%**************************************************
```

```
%**************************************************

m.layers{3}.type    = 'conv';
m.layers{3}.pz      = 2;
m.layers{3}.fCount  = 4;
m.layers{3}.tCount  = 5;
m.layers{3}.xyCount = 11;
m.layers{3}.abs     = 1;
m.layers{3}.size{1} = 4;
m = cns_mapdim(m, 3, 't', 'temp2', 2,  5, 1, 10);
m = cns_mapdim(m, 3, 'y', 'int'  , 2, 11, 1);
m = cns_mapdim(m, 3, 'x', 'int'  , 2, 11, 1);

m.layers{4}.type    = 'max';
m.layers{4}.pz      = 3;
m.layers{4}.tCount  = 2;
m.layers{4}.xyCount = 10;
m.layers{4}.size{1} = m.layers{3}.size{1};
m = cns_mapdim(m, 4, 't', 'temp2', 3,  2, 2, 10);
m = cns_mapdim(m, 4, 'y', 'int'  , 3, 10, 5);
m = cns_mapdim(m, 4, 'x', 'int'  , 3, 10, 5);

%**************************************************
```

Sparse Spatiotemporal Coding

- A good sparse coding basis for video spans frequencies, orientations, velocities and typically involves hundreds of basis vectors each of which spans both space and time.
- Running an iterative solver such as coordinate descent on each 3-D video patch corresponding to the receptive field of an individual cell is not practical even on modern GPUs.
- Instead of solving for the sparse coefficients, we learn to predict good approximations of these coefficients using a method called *predictive sparse decomposition* (PSD).

Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. Fast inference in sparse coding algorithms with applications to object recognition. Technical Report CBLL-TR-2008-12-01, Computational and Biological Learning Lab, Courant Institute, NYU, 2008.

Sparse Spatiotemporal Coding

Sparse Coding Objective Function:

$$J = \|X - AB\|_2^2 + \lambda \|A\|_1$$

Predictive Sparse Decomposition Function:

$$F(X;W) = F(X;G,M,B) = G * \tanh(MX + B)$$

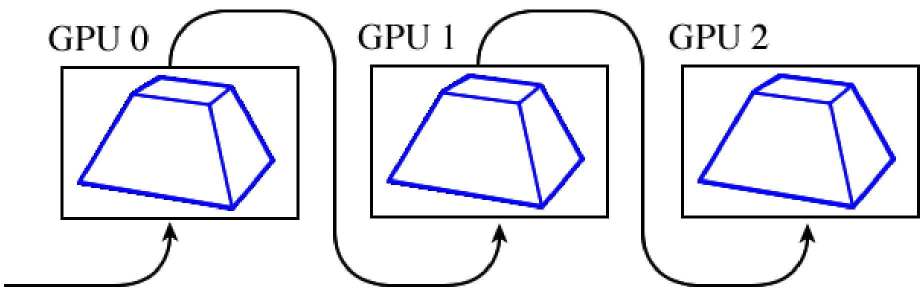
Amended Sparse Coding Objective Function:

$$J = \|X - AB\|_2^2 + \lambda \|A\|_1 + \beta \, \|A - F(X;W)\|_2^2$$

Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. Fast inference in sparse coding algorithms with applications to object recognition. Technical Report CBLL-TR-2008-12-01, Computational and Biological Learning Lab, Courant Institute, NYU, 2008.

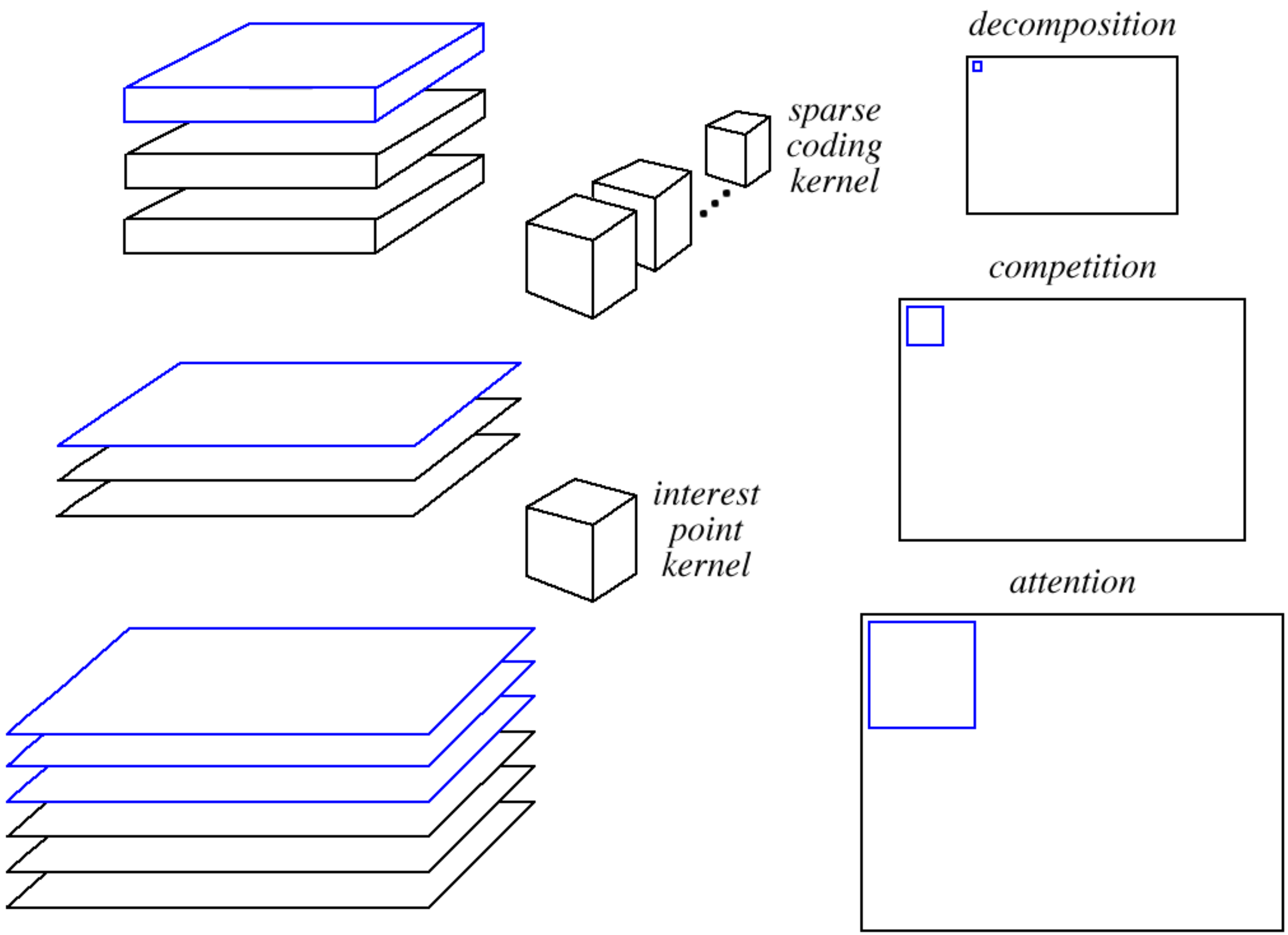
Sparse Spatiotemporal Coding

- Predictive sparse coding approximates the sparse codes produced by coordinate descent by substituting simple convolutions for the more time consuming iterative solver.
- Unfortunately, running hundreds of convolutions involving large convolution kernels is not practical even on GPUs.
- We could distribute the work over multiple GPUs, *e.g.,*

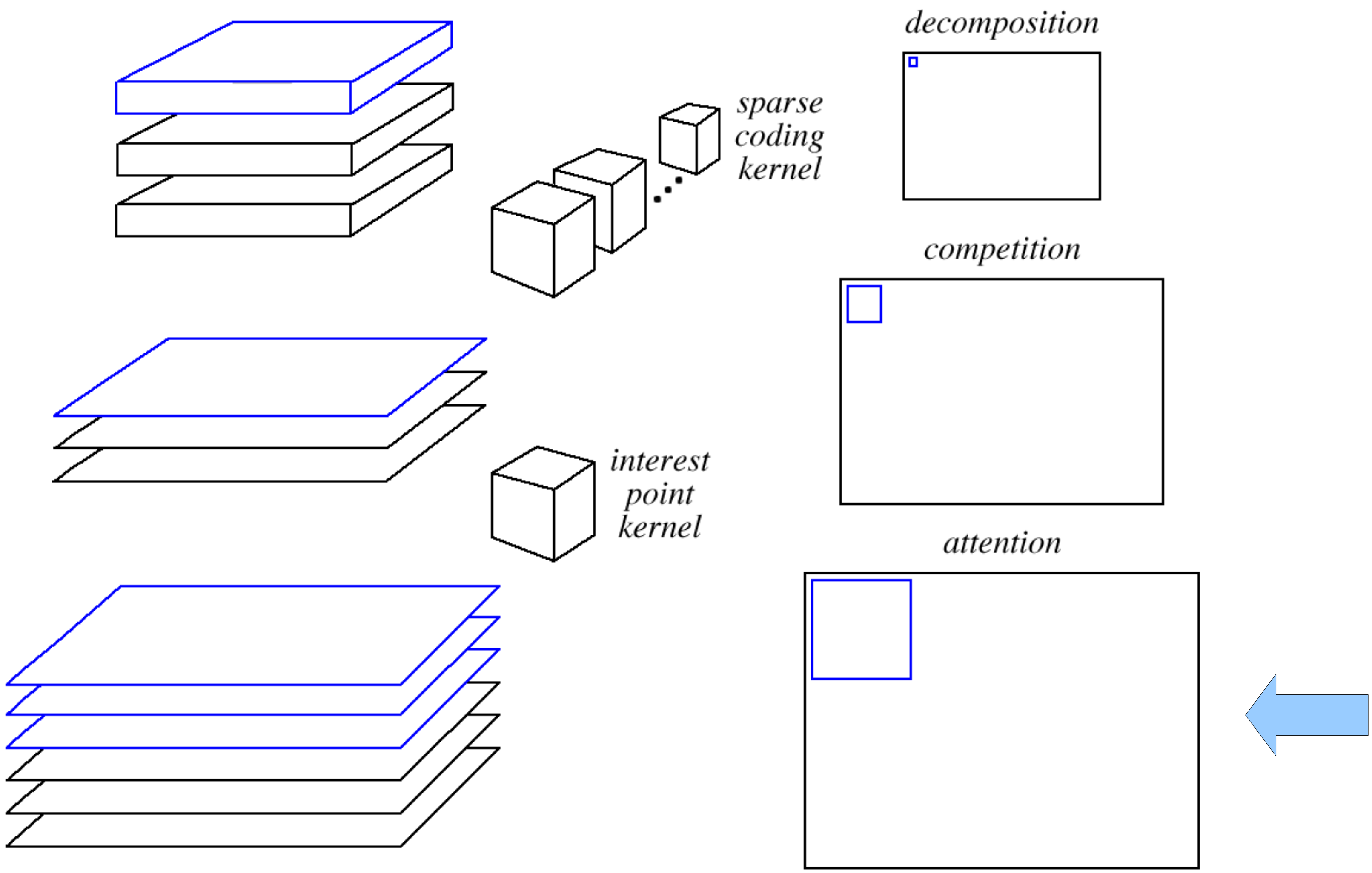


- Alternatively we could be more selective where we code.

Attention-Gated Sparse Coding



Attention-Gated Sparse Coding



Space-Time Interest Points

```
function response = filter(input, sigma, tau, radius)
% INPUTS
%   input    - 3-D input data
%   sigma    - spatial scale
%   tau      - temporal scale
%   radius   - filter radius
% OUTPUTS
%   response - detector response

% Generate 2-D Gaussian smoothing filter:
gauss = filterGauss2D(2 * radius + 1, [sigma, sigma]);

% Apply the smoothing filter spatially:
smooth = convn(input, gauss, 'valid');

% Generate Gabor filter quadrature pair:
[even, odd] = filterGabor1D(2 * tau, 2 * tau, 0.5 / tau);

% Apply the Gabor filters temporally:
quad_even = convn(smooth, permute(even,[3 1 2]),'valid');
quad_odd  = convn(smooth, permute(odd, [3 1 2]),'valid');

% Sum responses for quadrature energy:
response  = quad_even.^2 + quad_odd.^2;
```

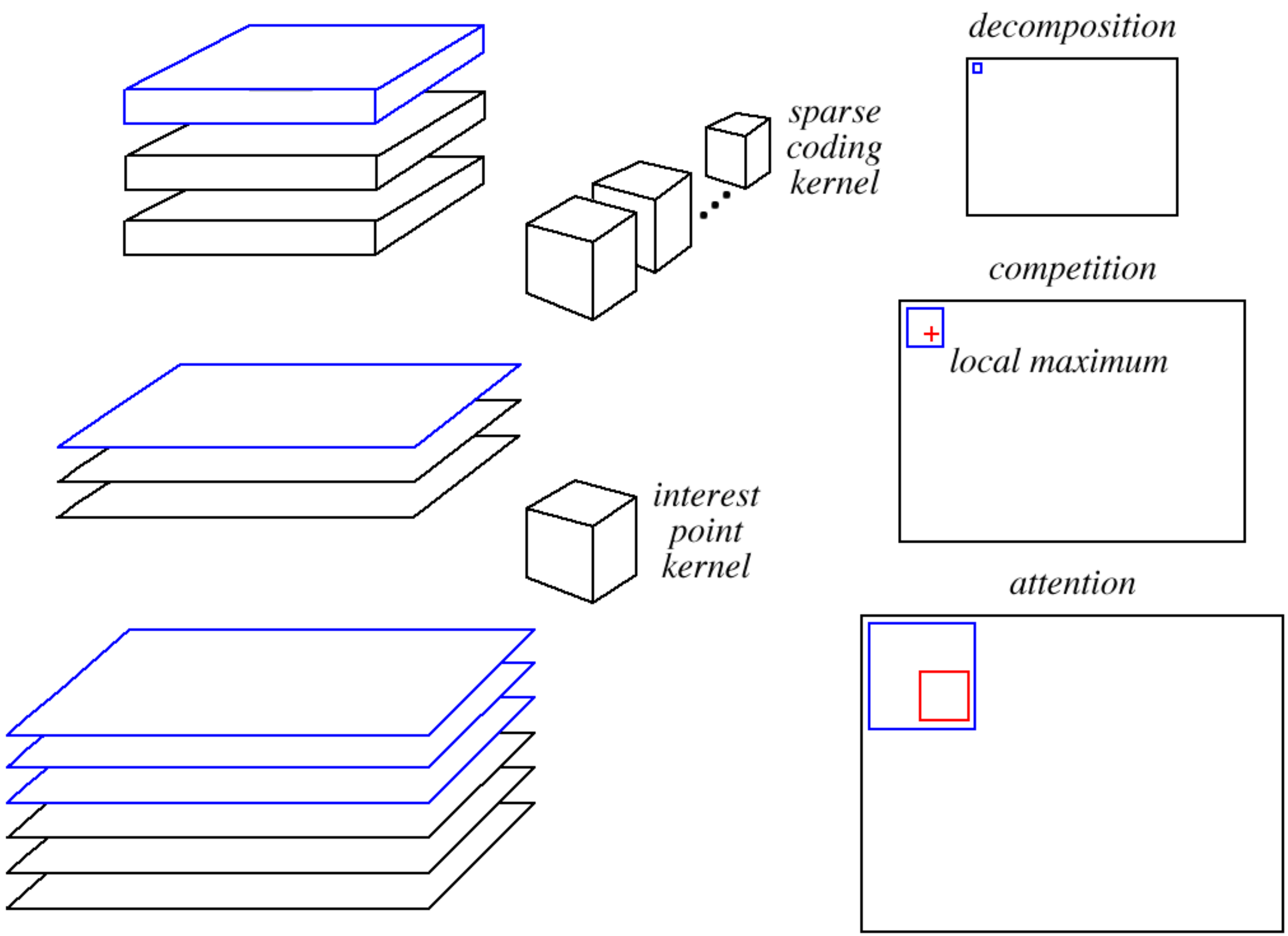
Piotr Dollár, Vincent Rabaud, Garrison Cottrell, and Serge Belongie. Behavior recognition via sparse spatio-temporal features. In Second Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005.

Interest Point Operator Kernel

```
int x1, x2, y1, y2;
GET_LAYER_X_RF_NEAR(PZ, GAUSS_X_SIZE, x1, x2);
GET_LAYER_Y_RF_NEAR(PZ, GAUSS_Y_SIZE, y1, y2);

float quad_even = 0.0f;
float quad_odd  = 0.0f;

// Dollar et al [2005] interest-point operator:
for (int t = 0; t < GABOR_T_SIZE; t++) {

    // Spatial smoothing with a Gaussian filter:
    float smooth = 0.0f;
    for (int j = 0, x = x1; x <= x2; j++, x++) {
    for (int i = 0, y = y1; y <= y2; i++, y++) {
        float v = READ_LAYER_VAL(PZ, 0, t, y, x);
        float w = READ_GAUSS(j, i);
        // Smooth the kth frame of the 3-D stack:
        smooth += v * w;
    }
    }

    // Temporal filter 1-D Gabor quadrature pair:
    quad_even += smooth * READ_GABOR(0, t);
    quad_odd  += smooth * READ_GABOR(1, t);

}

// Write quadrature-energy value for this cell:
WRITE_VAL(pow(quad_even, 2) + pow(quad_odd, 2));
```

Piotr Dollár, Vincent Rabaud, Garrison Cottrell, and Serge Belongie. Behavior recognition via sparse spatio-temporal features. In Second Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005.

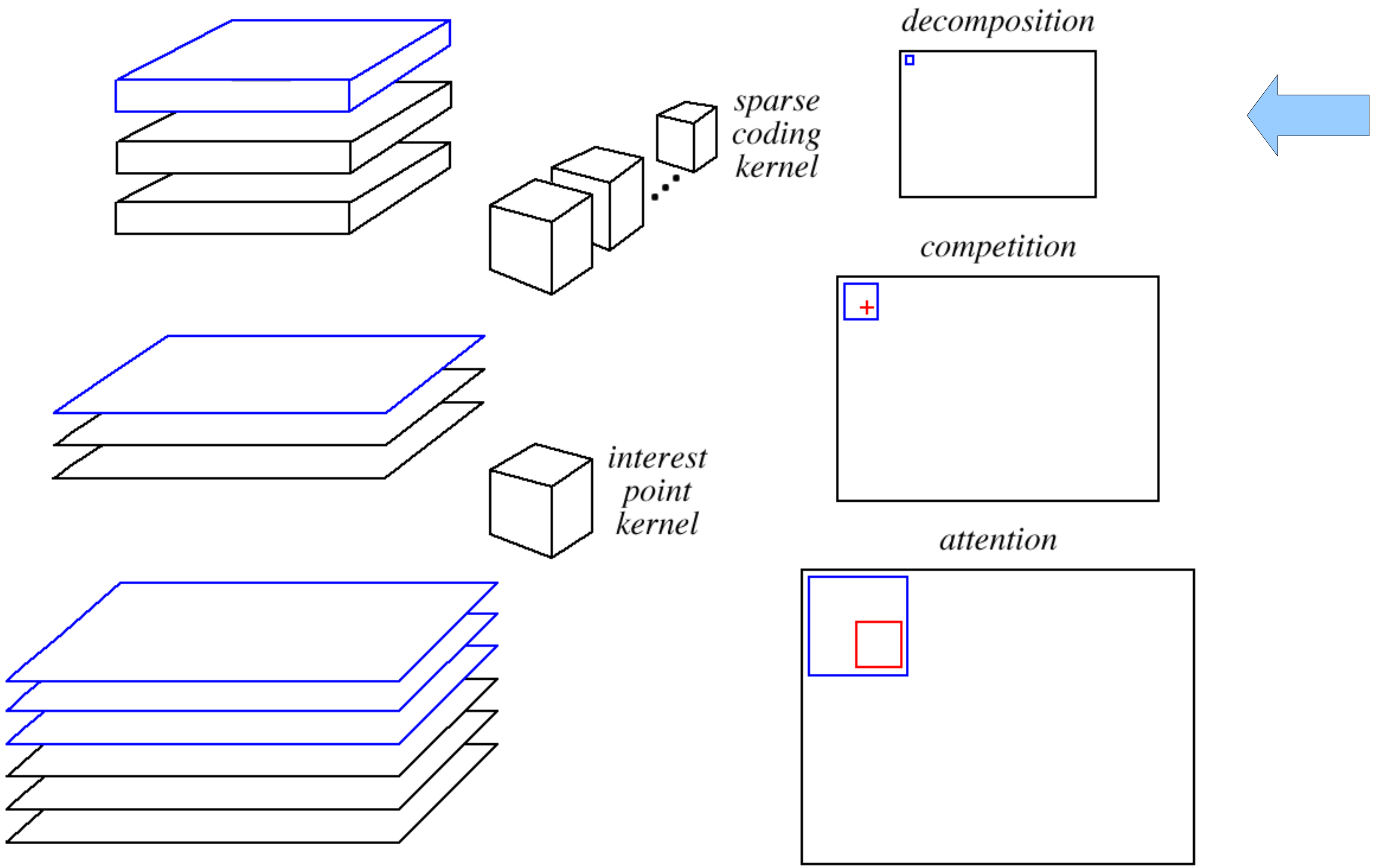
Attention-Gated Sparse Coding



Winner Take All Kernel

```
int y1, y2, x1, x2;
GET_LAYER_Y_RF_NEAR(PZ, WTASRCHWIN, y1, y2);
GET_LAYER_X_RF_NEAR(PZ, WTASRCHWIN, x1, x2);

int max_row, max_col;
int radius = MAXSUPRWIN / 2;
float max_resp = CNS_FLTMIN;

for (int x = x1, col = 0; x <= x2; col++, x++) {
for (int y = y1, row = 0; y <= y2; row++, y++) {
    // Read response from the center of 3 x 3 window:
    float ctr_resp = READ_LAYER_VAL(PZ, 0, 0, y, x);
    // Only interested if response exceeds threshold:
    if (ctr_resp < WTATHRSH)
        continue;
    // Determine if the response is a local maximum:
    bool max_flag = 1;
    for (int i = -radius; i <= radius; i++) {
    for (int j = -radius; j <= radius; j++) {
        if (i != 0 || j != 0) {
          float nbr_resp = READ_LAYER_VAL(PZ, 0, 0, y+j, x+i);
          max_flag = max_flag && (nbr_resp < ctr_resp);
        }
    }
    }
    // Save if local maximum and greater than current:
    if (max_flag && (ctr_resp > max_resp)) {
        max_row = row; max_col = col; max_resp = ctr_resp;
    }
}
}

if (max_resp == CNS_FLTMIN) {
    WRITE_ROW(-1); // no maxima were found
} else {
    WRITE_ROW(max_row);
    WRITE_COL(max_col);
}
```

Attention-Gated Sparse Coding



Localized Sparse Coding Kernel

```
int t1, t2, x1, x2, y1, y2;
GET_LAYER_T_RF_NEAR(PZ, FVALS_T_SIZE, t1, t2);
GET_LAYER_X_RF_NEAR(PZ, FVALS_X_SIZE, x1, x2);
GET_LAYER_Y_RF_NEAR(PZ, FVALS_X_SIZE, y1, y2);

// Read in the offsets for the local maximum:
int row_offset = READ_LAYER_ROW(CZ, 0, 0, THIS_Y, THIS_X);
int col_offset = READ_LAYER_COL(CZ, 0, 0, THIS_Y, THIS_X);

// If no local maximum found, write zero code:
if (row_offset < 0) { WRITE_VAL(0.0f); return; }

// Shift X and Y subscript indices by offset:
x1 += col_offset; x2 += col_offset;
y1 += row_offset; y2 += row_offset;

// Dot product of basis filter and coding region:
float result = 0.0f;
int f = THIS_F;
for (int k = 0, t = t1; t <= t2; k++, t++) {
for (int j = 0, y = y1; y <= y2; j++, y++) {
for (int i = 0, x = x1; x <= x2; i++, x++) {
    float v = READ_LAYER_VAL(PZ, 0, t, y, x);
    float w = READ_FVALS(0, k, j, i, f);
    result += w * v;
}
}
}

// Compute predictive sparse decomposition function:
WRITE_VAL(READ_GAIN(f) * tanh(result + READ_BIAS(f)));
```

Summary and Conclusions

- GPU programming can significantly shorten run time but it also invariably lengthens development time.
- However, CNS models can run automatically on GPUs without modification. How is this accomplished?
  - Everything in CNS is defined parametrically except the code a single kernel thread executes.
  - Even that code only communicates with its environment via macros.
  - When compiling for a CPU, kernel macros expand into code that accesses data structures in host memory.
  - When compiling for a GPU, those same macros expand into code that accesses GPU memory.

Summary and Conclusions

- CNS shields the programmer from the details of the GPU programming API, takes care of thread management, and handles most details of memory management including:
  - selecting the class of memory — global, texture, constant, shared,
  - explicitly initiating host-GPU memory transfers,
  - memory alignment and addressing, as well as
  - dimension mapping — N-D to 2-D, texture packing.
- CNS supports a powerful model of computing particularly well suited to biologically inspired computer vision:
  - Multiple layers encoding neighborhood preserving feature maps.
  - Layers consisting of cells defined by the same kernel computations.
  - Abstractions that cleanly generalize to handle space, scale and time.