Sharif University Of Technology

Vision Neuroscience

A Feedforward Architecture Accounts for Rapid Categorization

Instructor: Dr. Ebrahimpour

Zahra Kavian - 98102121

Contents

Section 1: Introduction Hmax Model

HMAX model is a bio-inspired feedforward architecture for object recognition, which is derived from the simple and complex cells model in cortex proposed by Hubel and Wiesel. As a hierarchical bio-based recognition model, HMAX captures the properties of primate cortex with alternated S and C layers, corresponding to simple cells and complex cells respectively. It should be noted that unlike other object recognition models or feature extraction models proposed and optimized for specific tasks, HMAX model differs from them in its creativeness to incorporate biological fmdings into computer algorithms and systems.

1.1 ORIGINAL HMAX MODEL

In its simplest version, the standard model consists of four layers of computational units where simple S units, which combine their inputs with Gaussian-like tuning to increase object selectivity, alternate with complex C units, which pool their inputs through a maximum operation, thereby introducing gradual invariance to scale and translation. The model has been able to quantitatively duplicate the generalization properties exhibited by neurons in inferotemporal monkey cortex (the so-called view-tuned units) that remain highly selective for particular objects (face, hand, toilet brush) while being invariant to ranges of scales and positions.

A sketch of this original HMAX model:

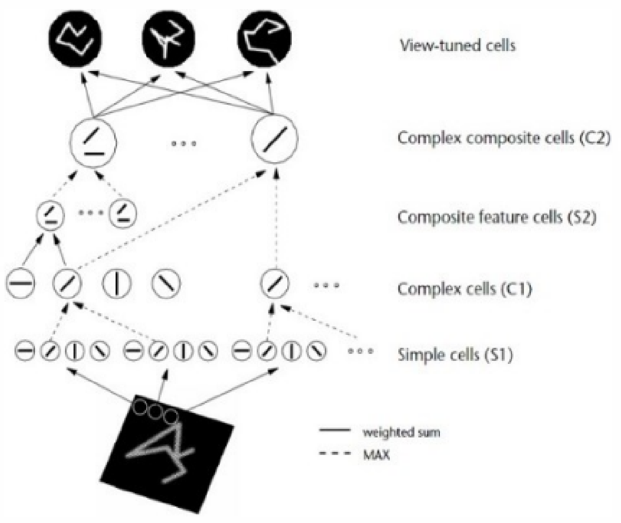

Fig.1. Origin HMAX model structure, originally presented in [4]

- *S1 Layer*:

  The $S1$ responses are obtained by applying to the input image a battery of Gabor filters, which can be described by the following equation:

  $$G(x, y) = exp(\frac{X^2 + \gamma^2 Y^2}{2\sigma^2}) \cos(\frac{2\pi}{\lambda} X) \quad (1)$$

  where $X = x \cos\theta + y \sin\theta$ and $Y = x \sin\theta + y \cos\theta$.

  We adjusted the filter parameters, i.e., orientation θ, effective width σ, and wavelength λ, so that the tuning profiles of $S1$ units match those of $V1$ parafoveal simple cells. This was done by first sampling the space of parameters and then generating a large number of filters. We applied those to stimuli commonly used to probe V1 neurons (i.e., gratings, bars and edges). After removing filters that were incompatible with biological cells, we were left with a final set of 16 filters at 4 orientations.

- *C1 Layer*:

  $C1$ corresponds to complex cells which show some tolerance to shift and size: complex cells tend to have larger receptive fields (twice as large as simple cells), respond to oriented bars or edges anywhere within their receptive field (shift invariance) and are in general more broadly tuned to spatial frequency than simple cells (scale invariance). Modifying the original Hubel and Wiesel proposal for building complex cells from simple cells through pooling, Riesenhuber and Poggio proposed a max-like pooling operation for building position- and scale tolerant $C1$ units. In the meantime, experimental evidence in favor of the max operation has appeared. Again pooling parameters were set so that $C1$ units match the tuning properties of complex cells as measured experimentally.

  Fig. 2 illustrates how pooling from $S1$ to $C1$ is done. $S1$ units come in 16 scales arranged in 8 bands Σ. For instance, consider the first band $\Sigma = 1$. For each orientation, it contains two $S1$ maps: one obtained using a filter of size 7, and one obtained using a filter of size 9. Note that both of these $S1$ maps have the same dimensions. In order to obtain the $C1$ responses, these maps are sub-sampled using a grid cell of size $N^{\Sigma} * N^{\Sigma} = 8 * 8$. From each grid cell we obtain one measurement by taking the maximum of all 64 elements. As a last stage we take a max over the two scales, by considering for each cell the maximum value from the two maps. This process is repeated independently for each of the four orientations and each scale band.

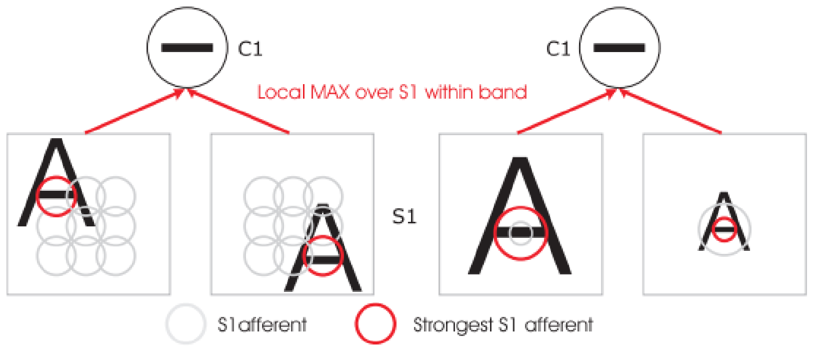

Figure 2. Scale- and position-tolerance at the complex cells (C1) level: Each C1 unit receives inputs from S1 units at the same preferred orientation arranged in bands Σ, i.e., S1 units in two different sizes and neighboring positions (grid cell of size $N^{\Sigma} \times N^{\Sigma}$). From each grid cell (left) we obtain one measurement by taking the max over all positions allowing the C1 unit to respond to an horizontal edge anywhere within the grid (tolerance to shift). Similarly, by taking a max over the two sizes (right) the C1 unit becomes tolerant to slight changes in scale.

- *S2 Layer*:

  A large pool of K patches of various sizes at random positions are extracted from a target set of images at the $C1$ level for all orientations, i.e., a patch P_i of size $n_i * n_i$ contains $n_i * n_i * 4$ elements, where the 4 factor corresponds to the four possible $S1$ and $C1$ orientations. The training process ends by setting each of those patches as prototypes or centers of the $S2$ units which behave as radial basis function (RBF) units during recognition, i.e., each $S2$ unit response depends in a Gaussian like way on the Euclidean distance between a new input patch (at a particular location and scale) and the stored prototype. This is consistent with well-known neuron response properties in primate inferotemporal cortex and seems to be the key property for learning to generalize in the visual and motor systems. When a new input is presented, each stored $S2$ unit is convolved with the new $(C1)^{\Sigma}$ input image at all scales (this leads to $K * 8(S2)_i^{\Sigma}$ images, where the K factor corresponds to the K patches extracted during learning and the 8 factor, to the 8 scale bands).

- *C2 Layer*:

  Compute the max over all positions and scales for each $S2$ map type $(S2)_i$ (i.e., corresponding to a particular patch P_i) and obtain shift- and scale-invariant $C2$ features $(C2)_i$, for $i = 1 \cdots K$.

Section 2: Model Functions

2.1 demoRelease

It is the main function. At first, it initialize some parameters such as patch size, train and test images direction path. Then it calls the function **'readAllImages'**. We explain this function later, but the output of this function is a $4 * 1$ cells contains all training and test images.

```
        useSVM = 1;
        READPATCHESFROMFILE = 0;
        patchSizes = [4 8 12 16];
        numPatchSizes = length(patchSizes);

        %specify directories for training and testing images
        train_set.pos   = '';
        train_set.neg   = '';
        test_set.pos    = '';
        test_set.neg    = '';

        cI = readAllImages(train_set,test_set);

        if isempty(cI{1}) | isempty(cI{2})
        error(['No training images were loaded -- did you remember to' ...
        ' change the path names?']);
        end
```

Then it extracts the $C1$ prototype, or in other word, $S1$ and $C1$ layers of the standard model for positive train images. It calls the function **'extractRandC1Patches'** which give the positive train images, number of patch sizes ($inourmodel = 4$), number of patches per size ($inthemodel = 250$), patch sizes $(4, 8, 12, 16)$. I analysis this function later. It return random prototypes as part of the training C2 classification.

```
        if ~READPATCHESFROMFILE
           tic
           numPatchesPerSize = 250;
           cPatches = extractRandC1Patches(cI{1}, numPatchSizes, ...
           numPatchesPerSize, patchSizes); %fix: extracting from positive only

           totaltimespectextractingPatches = toc;
        else
           fprintf('reading patches');
           cPatches = load('PatchesFromNaturalImages250per4sizes','cPatches');
           cPatches = cPatches.cPatches;
        end
```

After that, it set the parameters, which are used in 'gabor' filter and extract patches. 'rot' is a filter direction, 'c1ScaleSS' is scale vector, 'c1SpaceSS' is pooling range, 'RF_size' is a filter size. The **'init_gabor'** is called and return the all filters with different direction and size (in 'filters' parameter). We use them in $S1$ layer.

```matlab
        %----Settings for Testing --------%
        rot = [90 -45 0 45];
        c1ScaleSS = [1:2:18];
        RF_siz    = [7:2:39];
        c1SpaceSS = [8:2:22];
        Div = [4:-.05:3.2];
        %--- END Settings for Testing --------%

        [fSiz,filters,c1OL,numSimpleFilters] = init_gabor(rot, RF_siz, Div);
```

Then the **'extractC2forcell'** function is called to extract $C2$ feature for each training and testing images.

```matlab
        for i = 1:4,
        C2res{i} = extractC2forcell(filters,fSiz,c1SpaceSS,c1ScaleSS,c1OL,
           cPatches,cI{i},numPatchSizes);
        end
```

At the end, it use 'SVM' (**'CLSosusvm'** function) or 'Nearest Neighbor classifier' (**'CLSnnC'** function) to label the training images and measure the model score.

```matlab
        %Simple classification code
        XTrain = [C2res{1} C2res{2}];
        XTest =  [C2res{3},C2res{4}];
        ytrain = [ones(size(C2res{1},2),1);-ones(size(C2res{2},2),1)];
        ytest = [ones(size(C2res{3},2),1);-ones(size(C3res{4},2),1)];

        if useSVM
           Model = CLSosusvm(XTrain,ytrain);
           [ry,rw] = CLSosusvmC(XTest,Model);
        else
           Model = CLSnn(XTrain, ytrain);
           [ry,rw] = CLSnnC(XTest,Model);
        end
        successrate = mean(ytest==ry) % a simple classification score
```

2.2 readAllImages

- input: train_set (train images direction), test_set (test images direction), maximages-perdir (in our model set it ’inf)
- output: cI ($4 * 1$ cell, all training and testing images pathway and information)

It read and return the images. At first save train and test image directory (line $7 - 12$), then load and convert to gray-scale level (line $14 - 17$). It return *’CI’*, a cell contains the images of each four group.

```
 1  function cI = readAllImages(train_set,test_set,maximagesperdir);
 2    dnames = {train_set.pos,train_set.neg,test_set.pos,test_set.neg};
 3
 4    fprintf('Reading images...');
 5    cI = cell(4,1);
 6    for i = 1:4,
 7      c{i} = dir(fullfile(dnames{i}, '*.jpg'));
 8      if length(c{i})>0,
 9         if c{i}(1).name == '.',
10            c{i} = c{i}(3:end);
11         end
12      end
13  cI{i} = cell(length(c{i}),1);
14      for j = 1:length(c{i}),
15          cI{i}{j} = double(imread([dnames{i} '/' c{i}(j).name]))./255;
16          cI{i}{j} = rgb2gray(cI{i}{j});
17      end
18    end
```

2.3 init_gabor

- input: rot, RF_siz, Div *(I explain about these parameters in section 2.1)*
- output: fSiz (size of all filters), filters, c1OL, numSimpleFilters(number of filter, in this model equal four)

At first, we initial parameters:

```
        c1OL             = 2;
        numFilterSizes   = length(RF_siz);
        numSimpleFilters = length(rot);
        numFilters       = numFilterSizes*numSimpleFilters;
        fSiz             = zeros(numFilters,1); % vector with filter sizes
        filters          = zeros(max(RF_siz)^2,numFilters);

        lambda = RF_siz*2./Div;
        sigma  = lambda.*0.8;
        G      = 0.3;   % spatial aspect ratio: 0.23 < gamma < 0.92

        lambda = RF_siz*2./Div;
        sigma  = lambda.*0.8;
        G      = 0.3;   % spatial aspect ratio: 0.23 < gamma < 0.92
```

It creates two filter in size $11 * 11$ and $13 * 13$ for each direction. Store these filters in a matrix with size $13^2 * 8$. I describe each line with command in the below code.

```
        for k = 1:numFilterSizes
          for r = 1:numSimpleFilters
            theta     = rot(r)*pi/180; %%% convert degree to radian
            filtSize  = RF_siz(k);  %%% filter size (in our model: [7:2:39])
            center    = ceil(filtSize/2); %%% filter center
            filtSizeL = center-1;
            filtSizeR = filtSize-filtSizeL-1; %%% filtSizeL & filtSizeR set the
                filter region
            sigmaq    = sigma(k)^2;  %%% variance filter function

            %%% For each point in filter range, we measure gabor function with
                equation (1).

            for i = -filtSizeL:filtSizeR
                for j = -filtSizeL:filtSizeR
                   if ( sqrt(i^2+j^2)>filtSize/2 )
                       E = 0;
                   else
                       x = i*cos(theta) - j*sin(theta);
                       y = i*sin(theta) + j*cos(theta);
                       E = exp(-(x^2+G^2*y^2)/(2*sigmaq))*cos(2*pi*x/lambda(k));
                   end
                   f(j+center,i+center) = E;
                end
            end

          %%% Normalization

            f = f - mean(mean(f));
            f = f ./ sqrt(sum(sum(f.^2)));

          %%% p= 8, four direction in two scale filters. Reshape filter and
              store them in size 13*13. (we use two size of filter; 11 and 13. I
              think all of them store in a same matrix size)

            p = numSimpleFilters*(k-1) + r;
            filters(1:filtSize^2,p)=reshape(f,filtSize^2,1);
            fSiz(p)=filtSize;
          end
        end
```

2.4 unpadimage

- input: i (input image, after using gabor filter), amnt (rows and columns want to delete them)
- output: o (rebuild image after frame deletion)

It gets the images and delete the frame from each dimension.

```
%if length(amnt == 1), unpad equal on each side
%if length(amnt == 2), first amnt is left right, second up down
%if length(amnt == 4), then [left top right bottom];

switch(length(amnt))
  case 1
   sx = size(i,2) - 2 * amnt; sy = size(i,1) - 2 * amnt; %%% new image size
   l = amnt + 1; r = size(i,2) - amnt; t = amnt + 1; b = size(i,1) - amnt;  %%% which column and rows should keep in new image
  case 2
   sx = size(i,2) - 2 * amnt(1); sy = size(i,1) - 2 * amnt(2);
   l = amnt(1) + 1; r = size(i,2) - amnt(1); t = amnt(2) + 1; b = size(i,1) - amnt(2);
  case 4
   sx = size(i,2) - (amnt(1) + amnt(3)); sy = size(i,1) - (amnt(2) + amnt(4));
   l = amnt(1) + 1; r = size(i,2) - amnt(3); t = amnt(2) + 1; b = size(i,1) - amnt(4);
  otherwise
         error('illegal unpad amount\n');
  end

  o = i(t:b,l:r,:); %%% new image after unpadding
```

2.5 padimage

- input: i (input image, after using gabor filter), amnt (rows and columns want to add them), method (String values for pad method, 'circular': Pads with circular repetion of elements,'replicate': Repeats border elements of A, 'symmetric' Pads array with mirror reflections of itself.)
- output: o (rebuild image after padding)

Padarray which operates on only the first 2 dimensions of a 3 dimensional image. (of arbitrary number of layers)

```
    o = zeros(size(i,1) + 2 * amnt, size(i,2) + 2* amnt, size(i,3));
    for n = 1:size(i,3)
        o(:,:,n) = padarray(i(:,:,n),[amnt,amnt],method,'both');

        %%% 'both': Pads before the first element and after the last array
            element along each dimension.
        %%% In our model, methode=0, it means each pad element contains the
            value 0.
        %%% 'padarray(A,padsize,padval)': pads array A where padval specifies a
            constant value to use for padded elements or a method to replicate
            array elements.
    end
```

2.6 maxfilter

- input: I (input image), radius (the vector contain the max operation pooling radius. In other word, how many pixel in each dimension we choose for local max operation. If the vector have four nonzero value, we perfume local maximization in rectangle window. In this model radius = [0 0 9 9])
- output: I (new image after max pooling)

Performs morphological dilation on a multilayer image. In each row and column, only save the maximum number in the window, which length is same as radius (input parameter).

```
        switch length(radius)
           case 1,
             I = padimage(I,radius); %%% add extra row and columns to buffer
                 image in max operation
             [n,m,thirdd] = size(I); %%% input image size
             B = I;
             for i = radius+1:m-radius,
                 B(:,i,:) = max(I(:,i-radius:i+radius,:),[],2); %%% local
                     maximization in second dimension
             end

             for i = radius+1:n-radius,
                 I(i,:,:) = max(B(i-radius:i+radius,:,:),[],1); %%% local
                     maximization in 3th dimension
             end

             I = unpadimage(I,radius);
           case 4, %%% same as first case, but we have rectangle window for
               pooling
             [n,m,thirdd] = size(I);
             B = I;
             for i=1:radius(1)
                 B(:,i,:) = max(I(:,max(1,i-radius(1)):min(end,i+radius(3)),:)
                     ,[],2);
             end

             for i = radius(1)+1:m-radius(3),
                 B(:,i,:) = max(I(:,i-radius(1):i+radius(3),:),[],2);
             end

             for i=m-radius(3)+1:m
                 B(:,i,:) = max(I(:,i-radius(1):min(end,i+radius(3)),:),[],2);
             end

             for i = 1:radius(2),
                 I(i,:,:) = max(B(max(1,i-radius(2)):i+radius(4),:,:),[],1);
             end

             for i = radius(2)+1:n-radius(4),
                 I(i,:,:) = max(B(max(1,i-radius(2)):min(end,i+radius(4)),:,:)
                     ,[],1);
             end

             for i = n-radius(4)+1:n,
                 I(i,:,:) = max(B(i-radius(2):min(end,i+radius(4)),:,:),[],1);
             end
           otherwise,
             error('maxfilter: poorly defined radius\n');
        end
```

2.7 C1

- input: stim (the input image must be grayscale (single channel)), filters (Matrix of Gabor filters of size max_fSiz x num_filters, where max_fSiz is the length of the largest filter and num_filters the total number of filters. Column j of filters matrix contains a n_jxn_j filter (reshaped as a column vector and padded with zeros).In our model: $39^2 * 68$), fSiz (Vector of size num_filters containing the various filter sizes. In our model: $1 * 68$), c1SpaceSS (Vector defining the scale bands, i.e. a group of filter sizes over which a local max is taken to get the $C1$ unit responses, e.g. c1ScaleSS = $[1knum\_filters + 1]$ means 2 scale bands, the first one contains filters$(:, 1 : k - 1)$ and the second one contains filters$(:, k : num_filters)$. If N pooling bands, c1ScaleSS should be of length $N + 1$. In our model: $[8 : 2 : 22]$ (7 scale band)), c1ScaleSS (Vector defining the spatial pooling range for each scale band, i.e. c1SpaceSS(i) = m_i means that each $C1$ unit response in band i is obtained by taking a max over a local neighborhood of $m_i * m_i S1$ units. If N bands then c1SpaceSS should be of size NIn the model: $[1 : 2 : 18]$.), c1OL (Scalar value defining the overlap between $C1$ units. In scale band i, the $C1$ unit responses are computed every $c1Space(i)/c1OL.$), INCLUDEBORDERS (the type of treatment for the image border)
- output: S1 (Filtered images), C1 (The output of $C1$ layer, after max polling over the images)

Returns the $C1$ and $S1$ units' activation given the input image.

Computation for S layer:

At first, it rebuild the gabor filter to be suitable for 'conv2' function. All filter save in 'sqfilter' cell.

Before use gabor filter, images is convolve with a unique vector to normalize it (in function 'sumfilter'). Each eight gabor filter move across pixels and extract image features (use 'imread' matlab function). After using each filter, 'removeborders' function is called. This function replace the last frame of images with zero.

Computation for C layer:

(1) pool over scales within band: It maximize between two filtered images (after use two filter with same direction and different scale). We do this work for each four direction.

(2) pool over local neighborhood: In 'maxfilter' function, we locally maximize the image's pixel.

(3) subsample: The image is down-sample and save in lower dimension.

```
        numScaleBands=length(c1ScaleSS)-1;  % convention: last element in
            c1ScaleSS is max index + 1
        numScales=c1ScaleSS(end)-1;
        %   last index in scaleSS contains scale index where next band would
            start, i.e., 1 after highest scale!!
        numSimpleFilters=floor(length(fSiz)/numScales);

        for iBand = 1:numScaleBands
            ScalesInThisBand{iBand} = c1ScaleSS(iBand):(c1ScaleSS(iBand+1) -1);
        end

        % Rebuild all filters (of all sizes)
        %%%%%%%%
        nFilts = length(fSiz);
        for i = 1:nFilts
            sqfilter{i} = reshape(filters(1:(fSiz(i)^2),i),fSiz(i),fSiz(i));
            if USECONV2
               sqfilter{i} = sqfilter{i}(end:-1:1,end:-1:1); %flip in order to
                   use conv2 instead of imfilter (%bug_fix 6/28/2007);
            end
        end
```

```
        % Calculate all filter responses (s1)
        %%%%%%%%
        sqim = stim.^2;
        iUFilterIndex = 0;
        % precalculate the normalizations for the usable filter sizes
        uFiltSizes = unique(fSiz);
        for i = 1:length(uFiltSizes)
            s1Norm{uFiltSizes(i)} = (sumfilter(sqim,(uFiltSizes(i)-1)/2)).^0.5;
            %avoid divide by zero
            s1Norm{uFiltSizes(i)} = s1Norm{uFiltSizes(i)} + ~s1Norm{uFiltSizes(i
               )};
        end

        for iBand = 1:numScaleBands
            for iScale = 1:length(ScalesInThisBand{iBand})
                for iFilt = 1:numSimpleFilters
                    iUFilterIndex = iUFilterIndex+1;
                    if ~USECONV2
                    %%% imfilter(A,h,'symmetric','same','corr'): filters the
                        multidimensional array A with the multidimensional filter
                         h, 'symmetric': Input array values outside the bounds of
                         the array are computed by mirror-reflecting the array
                        across the array border, 'corr': imfilter performs
                        multidimensional filtering using correlation, 'same': The
                         output array is the same size as the input array.
                       s1{iBand}{iScale}{iFilt} = abs(imfilter(stim,sqfilter{
                          iUFilterIndex},'symmetric','same','corr'));

                    if(~INCLUDEBORDERS)
                      s1{iBand}{iScale}{iFilt} = removeborders(s1{iBand}{
                          iScale}{iFilt},fSiz(iUFilterIndex));
                    end
                    s1{iBand}{iScale}{iFilt} = im2double(s1{iBand}{iScale}{
                        iFilt}) ./ s1Norm{fSiz(iUFilterIndex)};
                  else %not 100% compatible but 20% faster at least
                    s1{iBand}{iScale}{iFilt} = abs(conv2(stim,sqfilter{
                        iUFilterIndex},'same'));
                    if(~INCLUDEBORDERS)
                      s1{iBand}{iScale}{iFilt} = removeborders(s1{iBand}{
                          iScale}{iFilt},fSiz(iUFilterIndex));
                    end
                    s1{iBand}{iScale}{iFilt} = im2double(s1{iBand}{iScale}{
                        iFilt}) ./ s1Norm{fSiz(iUFilterIndex)};
                  end
              end
          end
      end
```

```

      % Calculate local pooling (c1)
      %%%%%%%%

      %   (1) pool over scales within band
      for iBand = 1:numScaleBands
          for iFilt = 1:numSimpleFilters
              c1{iBand}(:,:,iFilt) = zeros(size(s1{iBand}{1}{iFilt}));
              for iScale = 1:length(ScalesInThisBand{iBand});
                  c1{iBand}(:,:,iFilt) = max(c1{iBand}(:,:,iFilt),s1{iBand}{
                      iScale}{iFilt});
              end
          end
      end

      %   (2) pool over local neighborhood
      for iBand = 1:numScaleBands
          poolRange = (c1SpaceSS(iBand));
          for iFilt = 1:numSimpleFilters
              c1{iBand}(:,:,iFilt) = maxfilter(c1{iBand}(:,:,iFilt),[0 0
                  poolRange-1 poolRange-1]);
          end
      end


      %   (3) subsample
      for iBand = 1:numScaleBands
          sSS=ceil(c1SpaceSS(iBand)/c1OL);
          clear T;
          for iFilt = 1:numSimpleFilters
              T(:,:,iFilt) = c1{iBand}(1:sSS:end,1:sSS:end,iFilt);
```

```
            end
            c1{iBand} = T;
        end
```

```

        function sout = removeborders(sin,siz)
        sin = unpadimage(sin, [(siz+1)/2,(siz+1)/2,(siz-1)/2,(siz-1)/2]);
        sin = padarray(sin, [(siz+1)/2,(siz+1)/2],0,'pre');
        sout = padarray(sin, [(siz-1)/2,(siz-1)/2],0,'post');
```

2.8 extractRandC1Patches

- input: cItrainingOnly (positive train images), numPatchSizes (the number of sizes in which the prototypes come. In our model = 4, since our patch size is [481216]), numPatchesPerSize (the number of prototypes extracted for each size. In our model = 250), patchSizes (vector of the patche sizes, [481216])
- output: cPatches ($4 * 1$ cell, return prototypes for each patch size)

It extracts random prototypes as part of the training of the $C2$ classification. At first, set the gabor filter parameters and call it.

```
        nImages = length(cItrainingOnly); %%% the number of train images

        %----Settings for Training the random patches--------%
        rot = [90 -45 0 45]; %%% gabor filter direction
        c1ScaleSS = [1 3];   %%% scale vector
        RF_siz    = [11 13]; %%% gabor filter size
        c1SpaceSS = [10]; %%% pooling range
        div = [4:-.05:3.2];
        Div       = div(3:4);
        %--- END Settings for Training the random patches--------%

        [fSiz,filters,c1OL,numSimpleFilters] = init_gabor(rot, RF_siz, Div);
```

It create the $4 * 1$ cells, for each patch size. In each cell, we have the zero matrix which has 250 columns (as the number of patch for specific patch size) and the rows for each direction and patch size (line $3 - 6$). Fore example, for patch size equal 8, we have $4 * 8 * 8$ rows ($4 * 4$ for reshape patch and 4 relate to 4 direction).

```
1        cPatches = cell(numPatchSizes,1); %%% to save our 4 type patches
2        bsize = [0 0];
3        pind = zeros(numPatchSizes,1);
4        for j = 1:numPatchSizes
5            cPatches{j} = zeros(patchSizes(j)^2*4,numPatchesPerSize);
6        end
```

At last, the $'C1'$ function use. The output is equal the output of $C1$ in the model. We use this output (e.i. the output of $S1$ and $C1$ layer, which are filtered images) and extract random patch. For each direction $(0, 45, -45, 90)$, four random patch in size of $(4, 8, 12, 16)$ extract. We have 16 patch for each image.

```
        for i = 1:numPatchesPerSize  %%% want to choose 250 patch
            ii = floor(rand*nImages) + 1;
                stim = cItrainingOnly{ii}; %%% randomly choose one of train
                    images
            img_siz = size(stim);  %%% random train image size

            %%% call 'C1' for this image. It return 'S1' and 'C1' layer output.
                (explain in section 2.7)
            [c1source,s1source] = C1(stim, filters, fSiz, c1SpaceSS, c1ScaleSS,
                c1OL);
            %%% new C1 interface and its size.
            b = c1source{1};
            bsize(1) = size(b,1);
            bsize(2) = size(b,2);

            for j = 1:numPatchSizes  %%%  for each patch size [4,8,12,16]
                xy = floor(rand(1,2).*(bsize-patchSizes(j)))+1;  %%% randomly
                    select patch index
                tmp = b(xy(1):xy(1)+patchSizes(j)-1,xy(2):xy(2)+patchSizes(j)
                    -1,:);  %%% randomly select patch for each 4 direction (third
                     dimention relate to this)
                pind(j) = pind(j) + 1;
                cPatches{j}(:,pind(j)) = tmp(:);  %%% save patches for patch
                    size (j). pind(j) is the count for 250 patch
            end
        end
```

2.9 sumfilter

- input: I (input image), radius (window length)
- output: I3 (image after local sum)

```matlab
    switch length(radius)
        case 4,  %%% I don't compeletly understand this part
          I2 = conv2(ones(1,radius(2)+radius(4)+1), ones(radius(1)+radius(3)+1,1), I);
          I3 = I2((radius(4)+1:radius(4)+size(I,1)), (radius(3)+1:radius(3)+size(I,2)));
        case 1,
          mask = ones(2*radius+1,1);
          I2 = conv2(mask, mask, I);  %%% convolve with a one vector equal local summation
          I3 = I2((radius+1:radius+size(I,1)), (radius+1:radius+size(I,2)));  %%% only keep I2 as same as input image.
    end
```

2.10 WindowedPatchDistance

- input: Im (the prototype (patches) to be used in the extraction of $s2$), Patch (one band of $'C1'$ output)
- output: D (euclidean distance)

Computes the euclidean distance between each Patch and all crops of image ($C1$ layer output) of similar size.

```
        dIm = size(Im,3); %%% dIM= number of filter use in C1 (=8)
        dPatch = size(Im,3);  %%% dPatch= the number of patch layer
        if(dIm ~= dPatch)  %%% check patch and input image have a same layer
           fprintf('The patch and image must be of the same number of layers');
        end
        s = size(Patch);
        s(3) = dIm;

        Psqr = sum(sum(sum(Patch.^2)));  %%% sum square of all sample in Patch
        Imsq = Im.^2; %%% sum square of each layer of Im (line - )
        Imsq = sum(Imsq,3);
        sum_support = [ceil(s(2)/2)-1,ceil(s(1)/2)-1,floor(s(2)/2),floor(s(1)/2)
           ]; %%% the window for convolution (use in 'sumfilter' function)

        Imsq = sumfilter(Imsq,sum_support); %%% after local with window size
           sum_support

        PI = zeros(size(Imsq));
        for i = 1:dIm
            PI = PI + conv2(Im(:,:,i),Patch(:,:,i), 'same');  %%% for each layer
               , convolve Patch with Im
        end

        D = Imsq - 2 * PI + Psqr + 10^-10;      %%% euclidean distance (I don't
           know what is this equation)
```

2.11 C2

- input: **stim** (the input image must be grayscale (single channel)), **filters** (Matrix of Gabor filters of size max_fSiz x num_filters, where max_fSiz is the length of the largest filter and num_filters the total number of filters. Column j of filters matrix contains a n_jxn_j filter (reshaped as a column vector and padded with zeros).In our model: $39^2 * 68$), **fSiz** (Vector of size num_filters containing the various filter sizes. In our model: $1 * 68$), **c1ScaleSS** (Vector defining the scale bands, i.e. a group of filter sizes over which a local max is taken to get the $C1$ unit responses, e.g. c1ScaleSS $= [1knum\_filters + 1]$ means 2 scale bands, the first one contains filters$(:, 1 : k-1)$ and the second one contains filters$(:, k : num_filters)$. If N pooling bands, c1ScaleSS should be of length $N + 1$. In our model: $[8 : 2 : 22]$ (7 scale band)), **c1SpaceSS** (Vector defining the spatial pooling range for each scale band, i.e. c1SpaceSS(i) $=$ m_i means that each $C1$ unit response in band i is obtained by taking a max over a local neighborhood of $m_i * m_i$ S1 units. If N bands then c1SpaceSS should be of size NIn the model: $[1 : 2 : 18]$.), **c1OL** (Scalar value defining the overlap between $C1$ units. In scale band i, the $C1$ unit responses are computed every $c1Space(i)/c1OL$.), **s2Target** (the prototype to be used in the extraction of $s2$. Each patch of size $[n, n, d]$ is stored as a column in s2Target, which has itself a size of $[n * nd, n\_patches]$), **c1** (if available, a precomputed $c1$ layer can be used to save computation time)
- output: c2,s2,c1,s1 (the output of $S1, S2, C1$ and $C2$ layer)

At first $'C1'$ is called and return the filtered images. Then we create the $'S2'$ layer as a RBF network. 'WindowedPatchDistance' is called. Then, calculate minimum distance (maximum stimulation) across position and scales, as $'C2'$ layer do.

```
        nbands = length(c1); %%% number of ScaleBands, used in C1
        c1BandImage = c1;  %%% the output of C1 layer
        nfilts = size(c1{1},3);  %%% floor(length(fSiz)/numScales), fsize: the
            output init_gabor, a vector with filter size. numScales: c1ScaleSS(
            end)-1
        n_rbf_centers = size(s2Target,2); %%% number of RBF unit (equal number
            of patch=4)
        L = size(s2Target,1) / nfilts;
        PatchSize = [L^.5,L^.5,nfilts];

        s2 = cell(n_rbf_centers,1);

        %Build s2:
        for iCenter = 1:n_rbf_centers  %%% for all prototypes in s2Target (RBF
            centers)
            Patch = reshape(s2Target(:,iCenter),PatchSize);
            s2{iCenter} = cell(nbands,1);
            for iBand = 1:nbands   %%% for all bands
                %%% Euclidean distance between a new input patch and the stored
                    prototype
                s2{iCenter}{iBand} = WindowedPatchDistance(c1BandImage{iBand},
                    Patch);
            end
        end

        %Build c2:
        c2 = inf(n_rbf_centers,1);
        for iCenter = 1:n_rbf_centers  %%% for all prototypes in s2Target (RBF centers)
            for iBand = 1:nbands %%% for all bands
                %%% calculate minimum distance (maximum stimulation) across position and scales
                c2(iCenter) = min(c2(iCenter),min(min(s2{iCenter}{iBand})));
            end
        end
```

2.12 extractC2forcell

- input: filters,fSiz,c1SpaceSS,c1ScaleSS,c1OL,cPatches,cImages,numPatchSizes (The $C1$ parameters used are given as the variables, for more detail regarding these parameters see section 2.7)
- output: mC2 (a matrix of size total_number_of_patches * number_of_images where total_number_of_patches is the sum over $i = 1 : numPatchSizes$ of length(cPatches$i$) (in the model = 4) and number_of_images is length(cImages)) (the number of positive images)

For each images and each four patch, it extract $C2$ feature ($'C2'$ function is called) and save them.

```
        numPatchSizes = min(numPatchSizes,length(cPatches));
        %all the patches are being flipped. This is becuase in matlab conv2 is
            much faster than filter2
        for i = 1:numPatchSizes,
            [siz,numpatch] = size(cPatches{i});
            siz = sqrt(siz/4);
            for j = 1:numpatch,
                tmp = reshape(cPatches{i}(:,j),[siz,siz,4]);
                tmp = tmp(end:-1:1,end:-1:1,:);
                cPatches{i}(:,j) = tmp(:);
            end
        end

        mC2 = [];

        for i = 1:length(cImages), %for every input image
            stim = cImages{i}; %%% selected input images
            img_siz = size(stim);
            c1  = [];
            iC2 = [];
            for j = 1:numPatchSizes, %for every unique patch size
                if isempty(c1),  %compute C2
                   [tmpC2,tmp,c1] = C2(stim,filters,fSiz,c1SpaceSS,c1ScaleSS,
                       c1OL,cPatches{j});
                else
                    [tmpC2] = C2(stim,filters,fSiz,c1SpaceSS,c1ScaleSS,c1OL,
                        cPatches{j},c1);
                end
            iC2 = [iC2;tmpC2];
        end
            mC2 = [mC2, iC2];
        end
```

Section 3: Result

As we see in Thomas Serre article: "A specific implementation of a class of feedforward theories of object recognition (that extend the Hubel and Wiesel simple-to complex cell hierarchy and account for many anatomical and physiological constraints) can predict the level and the pattern of performance achieved by humans on a rapid masked animal vs. non-animal categorization task". I run H-max model and try to see figure 1 as a result.

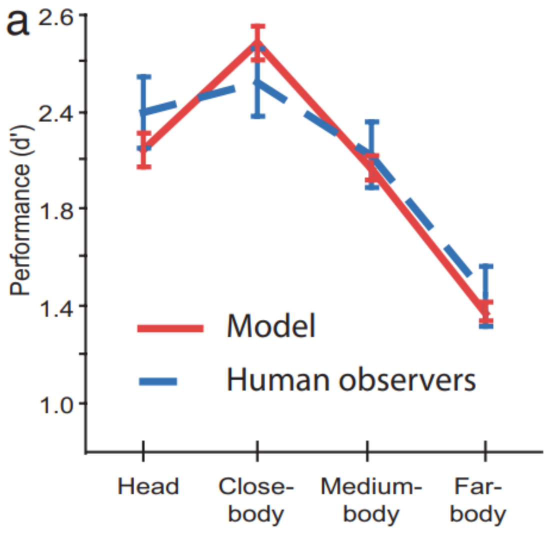

Figure 1: Model- vs. human-level accuracy. Human observers and the model exhibit a very similar pattern of performance (measured with $d^{'}$ measure). Error bars indicate the standard errors for the model (computed over $n = 20$ random runs) and for human observers (computed over $n = 24$ observers)

I run this model one time. You can see the result in figure 2.

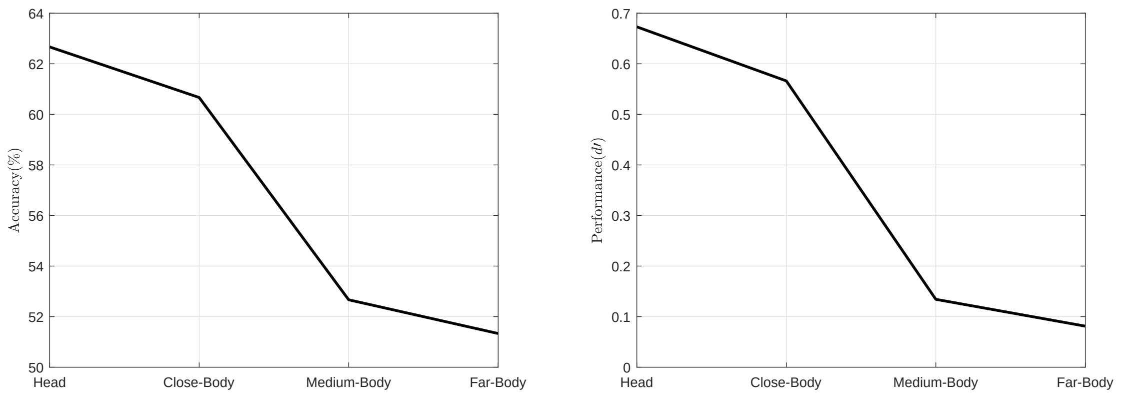

Figure 2: Model performance and accuracy

In this model, we randomly select prototypes. So to be sure the model result, I run the model two times and plot the errorbar in figure .

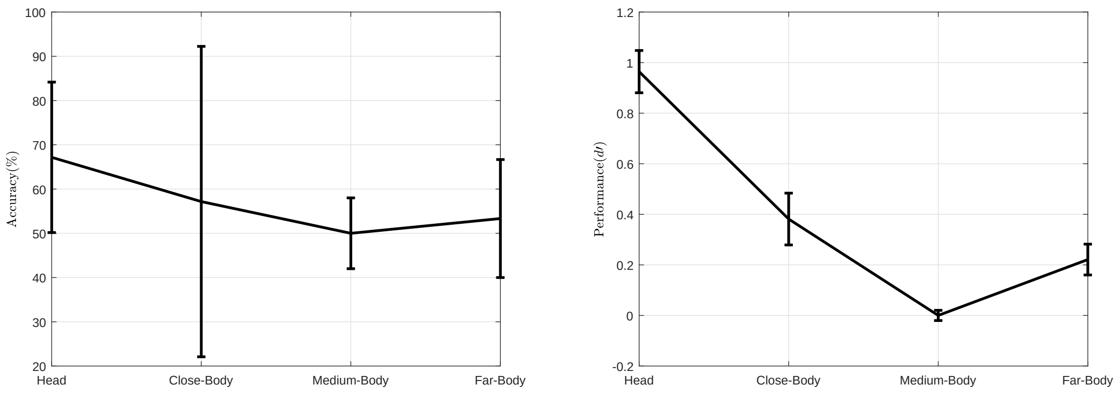

Figure 3: Model performance and accuracy

I also scramble 200 train images (100 target and 100 distractor). I run the model again (figure 4). accuracy doesn't change noticeably but the performance decrease.

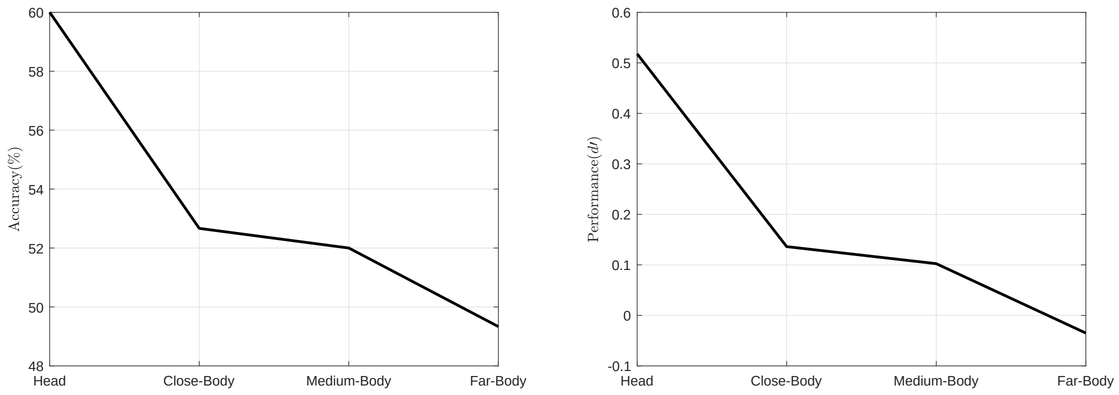

Figure 4: Model performance and accuracy with scramble train images

References

[1] Serre, T., Wolf, L., & Poggio, T. (2005, June). Object recognition with features inspired by visual cortex. In 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) (Vol. 2, pp. 994-1000). Ieee.

[2] Serre, T., Oliva, A.,& Poggio, T. (2007). A feedforward architecture accounts for rapid categorization. Proceedings of the national academy of sciences, 104(15), 6424-6429.

[3] Liu, C., & Sun, F. (2015, July). HMAX model: A survey. In 2015 International Joint Conference on Neural Networks (IJCNN) (pp. 1-7). IEEE.

[4] Ghodrati-Report on Hmax