ECE 559 – Neural Networks

Hand Written Character Recognition using Convolutional Neural Network

Assignment – IV

Arindam Bose
UIN: 665387232
10-8-2015

1. *Problem Statement*

Use Convolutional Neural network (ConvNet) for character recognition. Use 8x8and 10x10 grid to define your characters. The network should also be able to identify 'other' character i.e. any character other than the ones used for training should be classified as 'other'. So, in all, your CPNN should be able to classify any 8x8 and also 10x10 characters.

Test your network for characters with different bit noise.

Also, test it for characters other than the 3 characters used for training.

2. *Introduction*

Convolutional Neural Networks are very similar to ordinary Neural. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network express a single differentiable score function: From the raw image pixels on one end to class scores at the other. And they have a loss function on the last (fully-connected) layer. But ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduces the amount of parameters in the network

3. *Description of Architecture of the ConvNet*

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. For example, the input images in MNIST are an input volume of activations, and the volume has dimensions 28x28x3 (width, height, depth respectively) and CIFAR-10 dataset has 32x32x3 images. The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

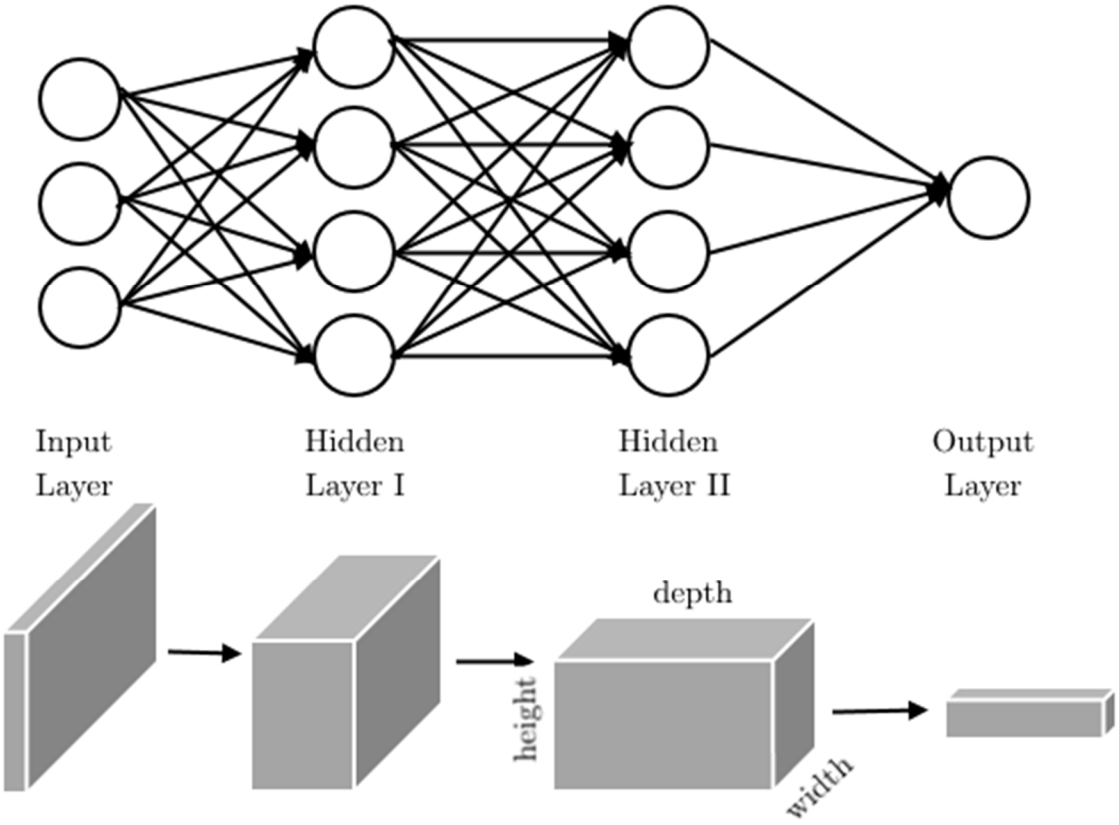


Fig. 1. The image at the top is a regular 3-layer Neural Network, and at the bottom is the ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers.

4. Layers used to build ConvNets

Every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer. Following are some details of different layers used in ConvNet. A simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC].

a. INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R-G-B.

b. CONV [32x32x12] layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and the region they are connected to in the input volume.

c. ReLU (rectified linear unit) [32x32x12] layer will apply an element-wise non-saturating activation function, such as the $f(x) = max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged [32x32x12]. It increases the nonlinear properties of the decision function and of the overall network without

affecting the receptive fields of the convolution layer. Other functions are also used to increase nonlinearity. For example the saturating hyperbolic tangent, $f(x) = \tanh(x), f(x) = |\tanh(x)|$, and the sigmoid function $f(x) = \frac{1}{(1+e^{-x})}$.

d. POOL [16x16x12] layer will perform a down-sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12]. In order to reduce variance, pooling layers compute the max or average value of a particular feature over a region of the image. This will ensure that the same result will be obtained, even when image features have small translations. This is an important operation for object classification and detection.

e. FC (Fully-Connected) [1x1x10] layer will compute the class scores, resulting in volume of size where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the ReLU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

5. *Learning Process*

In this project learning process consists of 2 steps: forward and backward passes that repeat for all objects in a training set. On the forward pass each layer transforms the output from the previous layer according to its function. The output of the last layer is compared with the label values and the total error is computed. On the backward pass the corresponding transformation happens with the derivatives of error with respect to outputs and weights of this layer. After the backward pass finished, the weights are changed in the direction that decreases the total error. This process is performed for a batch of objects simultaneously, in order to decrease the sample bias. After all the object have been processed, the process might repeat for different batch splits.

In this project total of 7 layers have been implemented. Layers can be one of 5 types:

a. *i - input layer*: Must be the first and only first one. Must contain the 'mapsize' field, that is a vector with 2 integer values, representing the objects size. May also contain the following additional fields:

1) outputmaps - that specifies the number of data channels, if it differs from 1.

2) norm - determines the desired norm of input vectors. It performs a normalization within a sample.

3) mean - determines the desired mean value of each feature across all samples.

4) maxdev - determines the maximum standard deviation of each feature across all samples.

b. *j - jitter layer:* Specify possible transformations of the input maps, that might be used to achieve transformation invariance. Must have the parameter 'mapsize'. Other possible parameters are:

1) shift - specifies the maximum shift of the image in each dimension,

2) scale - specifies the maximum scale in each dimension. Must be more than 1. The image scales with the random factors from $[\frac{1}{x}\ x]$.

3) mirror - binary vector, that determines if the image might be mirrored in a particular dimension or not.

4) angle - scalar, that specifies the maximum angle of rotation. Must be from [0, 1]. The value 1 corresponds to 180 degrees.

5) defval - specifies the value that is used when the transformed image lies outside the borders of the original image. If this value is not specified, the transformed value should be always inside the original one, otherwise there will be an error.

On the test set the images are just centrally cropped to the size 'mapsize', like there were no additional parameters.

c. c - convolutional *layer*: Must contain the 'filtersize' field that identifies the filter size. Must not be greater than the size of maps on the previous layer. Must also contain the 'outputmaps' field that is the number of maps for each objects on this layer. If the previous layer has m maps and the current one has n maps, the total number of filters on it is $m * n$. Despite that it is called convolutional, it performs filtering, that is a convolution operation with flipped dimensions. May contain the following additional fields:

1) padding - specifies the size of zero padding around the maps for each dimension. The default value is 0.

2) initstd - the standard deviation of normal distribution that is used to generate the weights. The default value is 0.01. Biases are always initialized by 0.

3) biascoef - specifies the multiplier for bias learning rate. Might be used if for some reason you decided to use another learning rate than for other weights. The default value is 1.

d. *s - scaling layer*: Reduces the map size by pooling. Must contain the 'scale' field, that is also a vector with 2 integer values. May additionally contain 'stride' field, that determines the distance between neighboring blocks in each dimension. By default is equal to 'scale'.

e. *f - fully connected layer*: Must contain the 'length' field that defines the number of its outputs. The last layer must have this type. For the last layer the length must coincide with the number of classes. May also contain the following additional fields:

1) 'dropout' - determines the probability of dropping the activations on this layer. Cannot be used on the last layer. Should not be too large, otherwise it drops everything.

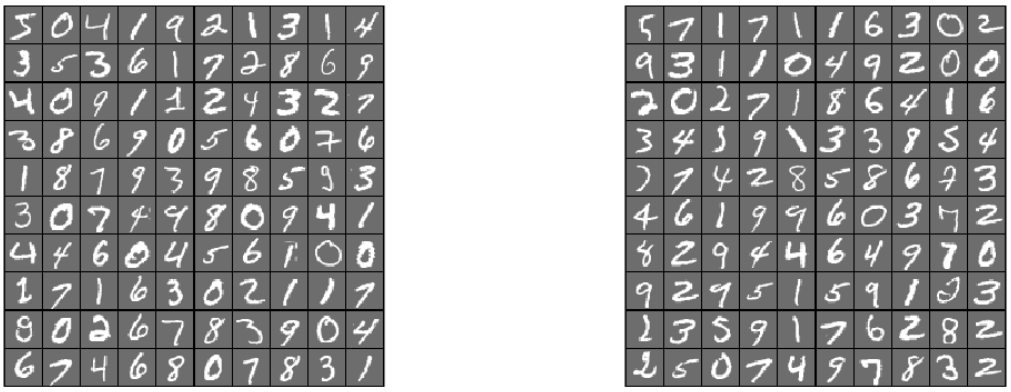
2) 'initstd' - the same as for convolutional layers. The default value is 0.1.

3) 'biascoef' - the same as for convolutional layers.

All layers except *i* may contain the 'function' field that defines their action. For: *c* and *f* - it defines the non-linear transformation function. It can be 'soft', 'sigm' or 'relu' that correspond to 'softmax', 'sigmoid' and 'rectified linear unit' respectively. The default value is 'relu'. The value 'soft' must be used only on the last layer. *s* - it defines the pooling procedure, that can be either 'mean' or 'max'. The default value is 'mean'.

6. *DataSet*

For this project I have used The MNIST Dataset of handwritten digits. The MNIST dataset (LeCun et al., 1998) contains handwritten digits, stored as black-and-white images of size 28 × 28. The total number of classes is 10, one for each digit. There are 60000 training instances and 10000 test instances in this dataset. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. Fig. 2 and 3 show some of the handwritten characters from the training set and sample test set respectively.



(a) (b)

Fig. 2. Sample Training Dataset

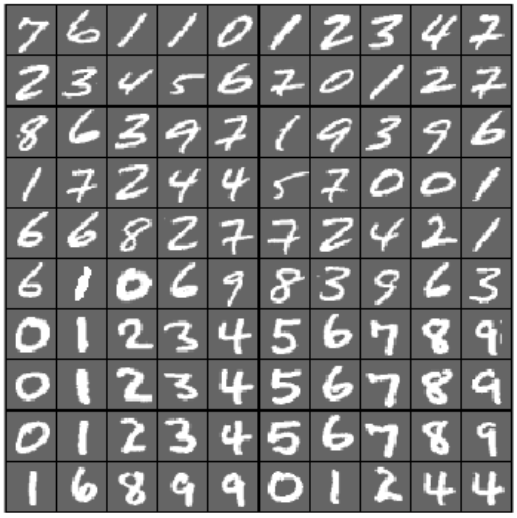


Fig. 3. Sample Testing Dataset

7. *Results*

a. Network Training: For this project I used 12000 out of 60000 training sets for my training and 1000 out of 10000 different test sets. So my training set composed with 28x28x12000 matrix. I have trained the whole network for 10 epochs (i.e. iterations). Then after each epoch I tested my test dataset with the trained network. Fig. 4 depicts the Epoch number vs Error rate curve. All error rates are in [0-1] range, 1 being the maximum and 0 being the minimum error rate. From the graph it is evident that as the Epoch number increases Error rate also decreases.

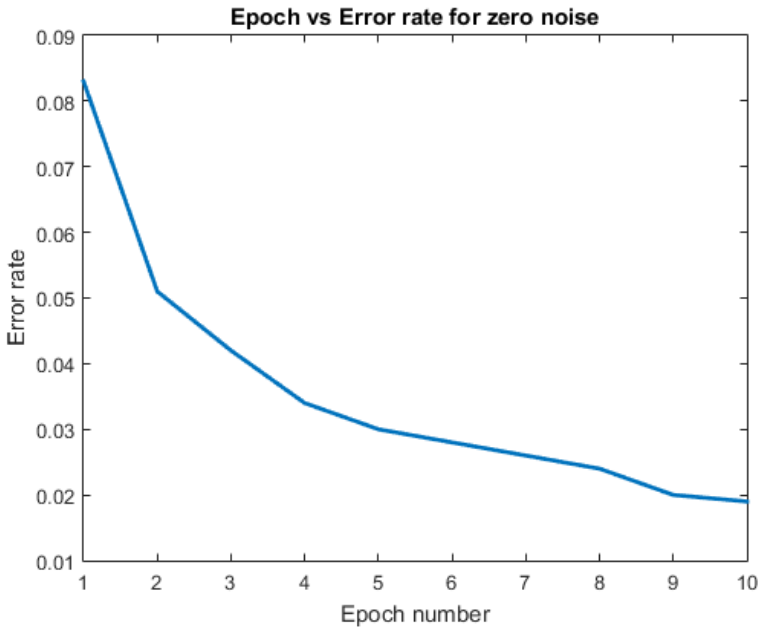


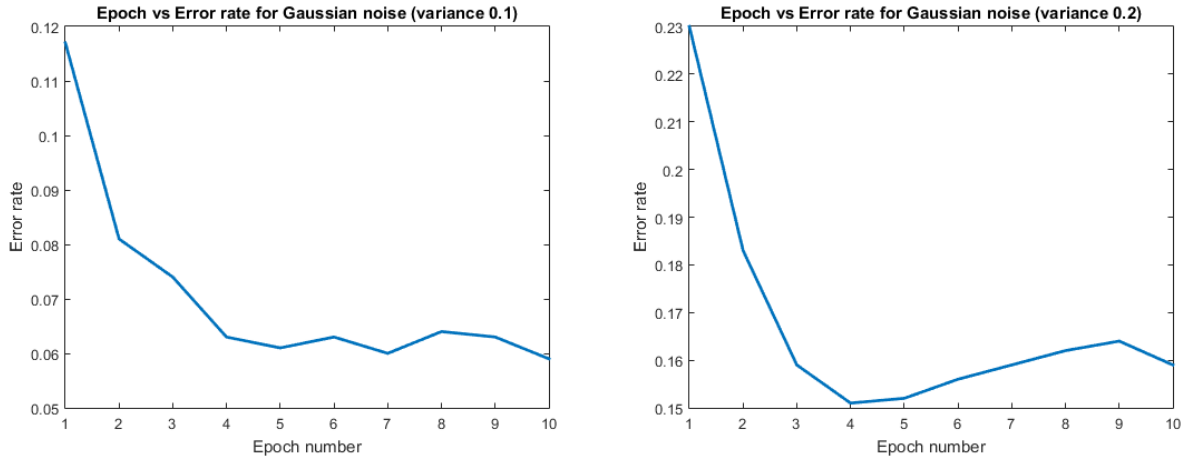
Fig. 4. Graph: Epoch vs Error rate

b. Robustness: To investigate the robustness of the neural network, I added Gaussian noise with different variances to my test data set and got the following results. Fig 5 depicts the Error rate with 0.1, 0.2, 0.3, 0.4, 0.5 variances respectively. It is evident from the following graphs that with the increase of variance of Gaussian noise from 0.1 to 0.5, the average error rate also increases gradually. But it is also seen in individual graphs until epoch 5 error rate decreases, but after this it increases especially when the noise is too high.

8. *Conclusions*

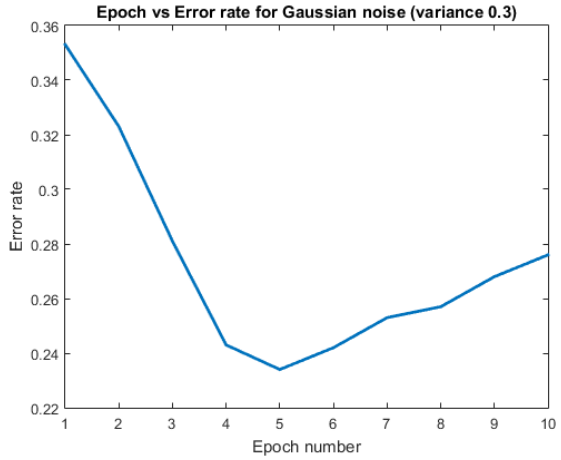
The ConvNet discussed here performed quiet well with both noise and no-noise dataset. With each epoch it becomes more efficient trained network. Following is the table depicting the time elapsed for training 10000 training samples to the whole network. The ConvNet is also highly robust with high convergence rate and also it is seen that it has very high success rate even if in the case of high noise.

Epoch:	1	2	3	4	5	6	7	8	9	10
Training Time (sec)	1241	906	897	899	899	895	897	896	900	901

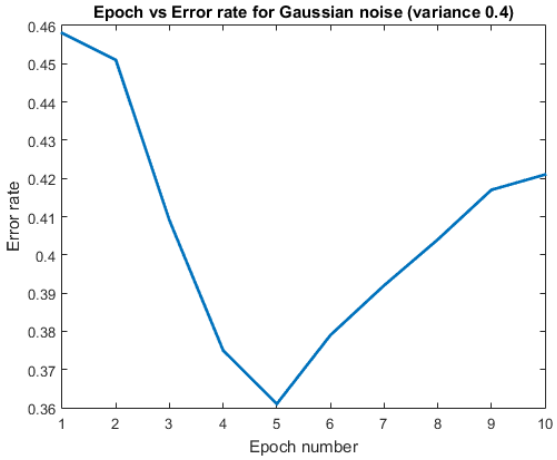


(a)

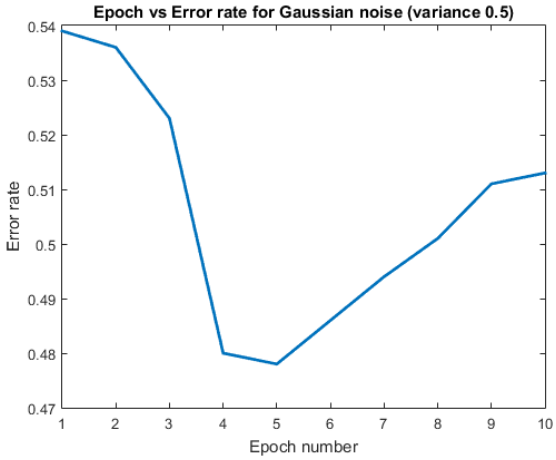
(b)



(c)



(d)



(e)

Fig. 5. Graph: Epoch vs Error rate for Gaussian Noise of (a)variance: 0.1, (b)variance: 0.2, (c)variance: 0.3, (d)variance: 0.4, (e)variance: 0.5.

9. References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[2] S. Demyanov, J. Bailey, R. Kotagiri, C. Leckie, "Invariant backpropagation: how to train a transformation-invariant neural network", arXiv:1502.04434v1 [stat.ML], February 2015.

[3] S. Demyanov, ConvNet Library for Convolutional Neural Network, https://github.com/sdemyanov/ConvNet

10. Appendix: Source Code (MATLAB R2015b)

a. main.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ECE 599 Neural Networs                                                  %
% Name: Arindam Bose                                                      %
% UIN: 665387232                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Implementation of Character Recognition system using
Convolutional Neural Network

close all;
cd(fileparts(mfilename('fullpath')));
funtype = 'matlab';

addpath('./matlab');
addpath('./data');
load mnist;

kSampleDim = ndims(TrainX);
kXSize = size(TrainX);
kXSize(kSampleDim) = [];
if (kSampleDim == 3)
  kXSize(3) = 1;
end;
kWorkspaceFolder = './workspace';
if (~exist(kWorkspaceFolder, 'dir'))
  mkdir(kWorkspaceFolder);
end;

kTrainNum = 12000;
kOutputs = size(TrainY, 2);
train_x = single(TrainX(:, :, 1:kTrainNum));
train_y = single(TrainY(1:kTrainNum, :));

kTestNum = 1000;
test_x = single(TestX(:, :, 1:kTestNum));
test_y = single(TestY(1:kTestNum, :));

test_x1 = imnoise(test_x(:,:,1:kTestNum),'gaussian',0,0.1);
test_x2 = imnoise(test_x(:,:,1:kTestNum),'gaussian',0,0.2);
```

```
test_x3 = imnoise(test_x(:,:,1:kTestNum),'gaussian',0,0.3);
test_x4 = imnoise(test_x(:,:,1:kTestNum),'gaussian',0,0.4);
test_x5 = imnoise(test_x(:,:,1:kTestNum),'gaussian',0,0.5);


clear params;
params.epochs = 1;
params.alpha = 0.1;
params.beta = 0;
params.momentum = 0.9;
params.lossfun = 'logreg';
params.shuffle = 1;
params.seed = 0;
dropout = 0;

layers = {
  struct('type', 'i', 'mapsize', kXSize(1:2), 'outputmaps',
kXSize(3))
  struct('type', 'c', 'filtersize', [4 4], 'outputmaps', 32)
  struct('type', 's', 'scale', [3 3], 'function', 'max', 'stride',
[2 2])
  struct('type', 'c', 'filtersize', [5 5], 'outputmaps', 64,
'padding', [2 2])
  struct('type', 's', 'scale', [3 3], 'function', 'max', 'stride',
[2 2])
  struct('type', 'f', 'length', 256, 'dropout', dropout)
  struct('type', 'f', 'length', kOutputs, 'function', 'soft')
};
weights = single(genweights(layers, params, funtype));
EpochNum = 10;
errors = zeros(EpochNum, 1);
for i = 1 : EpochNum
  % Training
  disp(['Epoch: ' num2str((i-1) * params.epochs + 1)])
  [weights, trainerr] = cnntrain(layers, weights, params, train_x,
train_y, funtype);
  disp([num2str(mean(trainerr(:, 1))) ' train loss']);
  trainerr(i) = mean(trainerr(:, 1));

  % Testing for zero noise
  [err0, bad, pred] = cnntest(layers, weights, params, test_x,
test_y, funtype);
  disp([num2str(err0*100) '% zero noise error']);
  errors0(i) = err0;

  % Testing for gaussian noise : 0.1 variance
  [err1, bad, pred] = cnntest(layers, weights, params, test_x1,
test_y, funtype);
  disp([num2str(err1*100) '% 0.1 variance error']);
  errors1(i) = err1;

  % Testing for gaussian noise : 0.2 variance
  [err2, bad, pred] = cnntest(layers, weights, params, test_x2,
test_y, funtype);
  disp([num2str(err2*100) '% 0.2 variance error']);
  errors2(i) = err2;

  % Testing for gaussian noise : 0.3 variance
```

```
  [err3, bad, pred] = cnntest(layers, weights, params, test_x3,
test_y, funtype);
  disp([num2str(err3*100) '% 0.3 variance error']);
  errors3(i) = err3;

  % Testing for gaussian noise : 0.4 variance
  [err4, bad, pred] = cnntest(layers, weights, params, test_x4,
test_y, funtype);
  disp([num2str(err4*100) '% 0.4 variance error']);
  errors4(i) = err4;

  % Testing for gaussian noise : 0.5 variance
  [err5, bad, pred] = cnntest(layers, weights, params, test_x5,
test_y, funtype);
  disp([num2str(err5*100) '% 0.5 variance error']);
  errors5(i) = err5;

  params.alpha = params.alpha * 0.95;
  params.beta = params.beta * 0.95;
end;
figure, plot(trainerr, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Training Loss'), title('Epoch vs Training Loss');
figure, plot(errors0, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for zero noise');
figure, plot(errors1, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for Gaussian noise
(variance 0.1)');
figure, plot(errors2, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for Gaussian noise
(variance 0.2)');
figure, plot(errors3, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for Gaussian noise
(variance 0.3)');
figure, plot(errors4, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for Gaussian noise
(variance 0.4)');
figure, plot(errors5, 'LineWidth', 2); xlabel('Epoch number'),
ylabel('Error rate'), title('Epoch vs Error rate for Gaussian noise
(variance 0.5)');
disp('Done!');
```

b. cnnclassify.m

```
function pred = cnnclassify(layers, weights, params, test_x, type)
if (length(size(test_x)) == 3)
  % insert singletone maps index
  test_x = permute(test_x, [1 2 4 3]);
end;
tic;
if strcmp(type, 'matlab')
  pred = classify_mat(layers, weights, params, test_x);
else
  error('"%s" - wrong type, must be "matlab"', type);
end;
t = toc;
disp(['Total classification time: ' num2str(t)]);
end
```

c. cnntest.m

```
function [err, bad, pred] = cnntest(layers, weights, params, test_x,
test_y, type)
pred = cnnclassify(layers, weights, params, test_x, type);
[~, pred_ind] = max(pred, [], 2);
[~, y_ind] = max(test_y, [], 2);
bad = find(pred_ind ~= y_ind);
err = length(bad) / size(pred_ind, 1);
end
```

d. cnntrain.m

```
function [weights, trainerr] = cnntrain(layers, weights_in, params,
train_x, train_y, type)
if (length(size(train_x)) == 3)
  % insert singletone maps index
  train_x = permute(train_x, [1 2 4 3]);
end;
tic;
if strcmp(type, 'matlab')
  [weights, trainerr] = cnntrain_mat(layers, weights_in, params,
train_x, train_y);
else
  error('"%s" - wrong type, must be "matlab"', type);
end;
t = toc;
disp(['Total training time: ' num2str(t)]);
end
```

e. classify_mat.m

```
function pred = classify_mat(layers, weights, params, test_x)
layers = cnnsetup(layers, 0);
layers = setweights(layers, weights);
assert(size(test_x, 1) == layers{1}.mapsize(1) && ...
       size(test_x, 2) == layers{1}.mapsize(2), ...
       'Data and the first layer must have equal sizes');
assert(size(test_x, 3) == layers{1}.outputmaps, ...
       'The number of the input data maps must be as specified');
test_x = normalize(layers{1}, test_x);
layers = initact(layers, test_x);
[~, pred] = forward(layers, 0);
end
```

f. genweights.m

```
function weights = genweights(layers, params, type)
if (strcmp(type, 'cpu') || strcmp(type, 'gpu'))
  weights = genweights_mex(layers, params);
elseif strcmp(type, 'matlab')
  weights = genweights_mat(layers, params);
else
```

```
  error('"%s" - wrong type, must be either "cpu", "gpu" or
"matlab"', type);
end;
end
```

g. backward.m

```
function layers = backward(layers, params)
n = numel(layers);
batchsize = size(layers{1}.a, 4);
for l = n : -1 : 1
  if strcmp(layers{l}.type, 'c') || strcmp(layers{l}.type, 'f')
    if strcmp(layers{l}.function, 'soft') % for softmax
      if (~strcmp(params.lossfun, 'logreg'))
        layers{l}.d = softder(layers{l}.d, layers{l}.a);
      end;
    elseif strcmp(layers{l}.function, 'sigm') % for sigmoids
      layers{l}.d = layers{l}.d .* layers{l}.a .* (1 - layers{l}.a);
    elseif strcmp(layers{l}.function, 'relu')
      layers{l}.d = layers{l}.d .* (layers{l}.a > 0);
    end;
    if (strcmp(layers{l}.function, 'soft') ||
strcmp(layers{l}.function, 'sigm'))
      layers{l}.d(-layers{l}.eps < layers{l}.d & layers{l}.d <
layers{l}.eps) = 0;
    end;
  end;
  if strcmp(layers{l}.type, 'c')
    d_cur = layers{l}.d;
    if (layers{l}.padding(1) > 0 || layers{l}.padding(2) > 0)
      ds = size(layers{l}.d); ds(end+1:4) = 1;
      padding = layers{l}.filtersize - 1 - layers{l}.padding;
      d_cur = zeros([ds(1:2) + 2*padding ds(3:4)]);
      d_cur(padding(1)+1:padding(1)+ds(1),
padding(2)+1:padding(2)+ds(2), :, :) =  layers{l}.d;
    end;
    layers{l-1}.d = zeros(size(layers{l-1}.a));
    for i = 1 : layers{l}.outputmaps
      for j = 1 : layers{l-1}.outputmaps
        if (layers{l}.padding(1) > 0 || layers{l}.padding(2) > 0)
          layers{l-1}.d(:, :, j, :) = layers{l-1}.d(:, :, j, :) +
...
            convn(d_cur(:, :, i, :), layers{l}.k(:, :, j, i),
'valid');
        else
          layers{l-1}.d(:, :, j, :) = layers{l-1}.d(:, :, j, :) +
...
            convn(d_cur(:, :, i, :), layers{l}.k(:, :, j, i),
'full');
        end;
      end
    end;
    layers{l-1}.d(-layers{l-1}.eps < layers{l-1}.d & layers{l-1}.d <
layers{l-1}.eps) = 0;
    %disp(sum(layers{l-1}.d(:)));
    %disp(layers{l-1}.d(1, 1:5, 1, 1));
```

```
  elseif strcmp(layers{l}.type, 's')
    sc = [layers{l}.scale 1 1];
    st = [layers{l}.stride 1 1];
    targsize = layers{l-1}.mapsize;
    curder = expand(layers{l}.d, sc);
    if strcmp(layers{l}.function, 'max')
      curval = expand(layers{l}.a, sc);
      prevval = stretch(layers{l-1}.a, sc, st);
      maxmat = (prevval == curval);
      %maxmat = uniq(maxmat, sc);
      curder = curder .* maxmat;
      layers{l-1}.d = shrink(curder, sc, st);
    elseif strcmp(layers{l}.function, 'mean')
      curder = curder / prod(sc);
      if (~isequal(sc, st))
        curder = shrink(curder, sc, st);
      end;
      layers{l-1}.d = curder;
    end;
    ind = (layers{l}.mapsize - 1) .* st(1:2);
    realnum = targsize - ind;
    if (sc(1) > realnum(1))
      extra = sum(layers{l-1}.d(targsize(1)+1:end, :, :, :), 1) /
realnum(1);
      layers{l-1}.d(targsize(1)+1:end, :, :, :) = [];
      layers{l-1}.d(ind(1)+1 : targsize(1), :, :, :) = ...
        layers{l-1}.d(ind(1)+1 : targsize(1), :, :, :) + ...
        repmat(extra, [realnum(1) 1 1 1]);
    end;
    if (sc(2) > realnum(2))
      extra = sum(layers{l-1}.d(:, targsize(2)+1:end, :, :), 2) /
realnum(2);
      layers{l-1}.d(:, targsize(2)+1:end, :, :) = [];
      layers{l-1}.d(:, ind(2)+1 : targsize(2), :, :) = ...
        layers{l-1}.d(:, ind(2)+1 : targsize(2), :, :) + ...
        repmat(extra, [1 realnum(2) 1 1]);
    end;
  elseif strcmp(layers{l}.type, 'f')
    layers{l}.di = layers{l}.d * layers{l}.w;
    if strcmp(layers{l-1}.type, 'f')
      layers{l-1}.d = layers{l}.di;
      if (layers{l-1}.dropout > 0) % dropout
        layers{l-1}.d = layers{l-1}.d .* layers{l-1}.dropmat;
      end;
    else
      layers{l-1}.d = reshape(layers{l}.di', [layers{l-1}.mapsize
layers{l-1}.outputmaps batchsize]);
    end;
    layers{l-1}.d(-layers{l-1}.eps < layers{l-1}.d & layers{l-1}.d <
layers{l-1}.eps) = 0;
  end;
end
end
```

h. calcweights.m

```
function layers = calcweights(layers, passnum)
n = numel(layers);
batchsize = size(layers{1}.a, 4);
for l = 1 : n
  if strcmp(layers{l}.type, 'n')
    meander = layers{l}.d;
    if (layers{l}.is_dev == 1)
      meander = meander .* repmat(layers{l}.w(:, :, :, 2), [1 1 1
batchsize]);
      stdevder = layers{l-1}.a + repmat(layers{l}.w(:, :, :, 1), [1
1 1 batchsize]);
      layers{l}.dw(:, :, :, 2) = mean(layers{l}.d .* stdevder, 4);
    end;
    layers{l}.dw(:, :, :, 1) = mean(meander, 4);
    layers{l}.dw(-layers{l}.eps < layers{l}.dw & layers{l}.dw <
layers{l}.eps) = 0;
  elseif strcmp(layers{l}.type, 'c')
    a_prev = layers{l-1}.a;
    if (layers{l}.padding(1) > 0 || layers{l}.padding(2) > 0)
      as = size(layers{l-1}.a);
      padding = layers{l}.padding;
      a_prev = zeros([as(1:2) + 2*padding size(layers{l-1}.a, 3)
size(layers{l-1}.a, 4)]);
      a_prev(padding(1)+1:padding(1)+as(1),
padding(2)+1:padding(2)+as(2), :, :) = layers{l-1}.a;
    end;
    for i = 1 : layers{l}.outputmaps
      for j = 1 : layers{l-1}.outputmaps
        dk = filtn(a_prev(:, :, j, :), layers{l}.d(:, :, i, :),
'valid') / batchsize;
        if (passnum == 2)
          layers{l}.dk(:, :, j, i) = dk;
        elseif (passnum == 3)
          layers{l}.dk2(:, :, j, i) = dk;
        end;
      end
    end;
    if (passnum == 2)
      layers{l}.db = squeeze(sum(sum(sum(layers{l}.d, 4), 2), 1)) /
batchsize;
      layers{l}.db = layers{l}.db * layers{l}.biascoef;
      layers{l}.db(-layers{l}.eps < layers{l}.db & layers{l}.db <
layers{l}.eps) = 0;
      layers{l}.dk(-layers{l}.eps < layers{l}.dk & layers{l}.dk <
layers{l}.eps) = 0;
    elseif (passnum == 3)
      layers{l}.dk2(-layers{l}.eps < layers{l}.dk2 & layers{l}.dk2 <
layers{l}.eps) = 0;
    end;
    %disp('calc_weights');
    %disp(sum(layers{l}.dk(:)));
    %disp(layers{l}.dk(1, 1:5, 1, 1));

  elseif strcmp(layers{l}.type, 'f')
    dw = layers{l}.d' * layers{l}.ai / batchsize;
    if (passnum == 2)
```

```
      layers{l}.dw = dw;
      layers{l}.db = mean(layers{l}.d, 1);
      layers{l}.db = layers{l}.db * layers{l}.biascoef;
    elseif (passnum == 3)
      layers{l}.dw2 = dw;
    end;
    if (passnum == 2)
      layers{l}.db(-layers{l}.eps < layers{l}.db & layers{l}.db <
layers{l}.eps) = 0;
      layers{l}.dw(-layers{l}.eps < layers{l}.dw & layers{l}.dw <
layers{l}.eps) = 0;
    elseif (passnum == 3)
      layers{l}.dw2(-layers{l}.eps < layers{l}.dw2 & layers{l}.dw2 <
layers{l}.eps) = 0;
    end;

    %disp('calc_weights');
    %disp(sum(layers{l}.dk(:)));
    %disp(layers{l}.dw(1, 1:5, 1, 1));
  end;
end
end
```

i. classify_ext_mat.m

```
function [pred, loss, loss2]  = classify_ext_mat(layers, weights,
test_x, test_y, ceps)
layers{1} = cnnsetup(layers{1}, 0);
layers{1} = initact(layers{1}, test_x);
layers{2} = cnnsetup(layers{2}, 0);
layers{2} = setweights(layers{2}, weights{2});
assert(size(test_x, 1) == layers{1}{1}.mapsize(1) && ...
       size(test_x, 2) == layers{1}{1}.mapsize(2), ...
       'Data and the first layer must have equal sizes');
assert(size(test_x, 3) == layers{1}{1}.outputmaps, ...
       'The number of the input data maps must be as specified');
%ceps = 1e-3;
test_w = weights{1};
pixels_num = size(test_x, 4);
batchsize = size(test_w, 2);
imlayers = cell(1, batchsize);
images = zeros([pixels_num batchsize]);
for m = 1 : batchsize
  imlayers{m} = setweights(layers{1}, test_w(:, m));
  pix_shift = rand(1, 2) - 0.5;
  pix_shift = pix_shift / sqrt(sum(pix_shift.^2, 2));
  cur_x = test_x + ceps * repmat(pix_shift, [1 1 1 size(test_x,
4)]);
  imlayers{m} = initact(imlayers{m}, cur_x);
  [imlayers{m}, pred] = forward(imlayers{m}, 1);
  images(:, m) = pred;
end;
images_act = reshape(images, [layers{2}{1}.mapsize
layers{2}{1}.outputmaps batchsize]);
layers{2} = initact(layers{2}, images_act);
[layers{2}, pred] = forward(layers{2}, 1);
% second pass
```

```
[layers{2}, loss] = initder(layers{2}, test_y);
layers{2} = backward(layers{2});
images_der = reshape(layers{2}{1}.d, [pixels_num batchsize]);
for m = 1 : batchsize
  imlayers{m}{end}.d = images_der(:, m);
  imlayers{m} = backward(imlayers{m});
end;
% third pass
loss2 = 0; invnum = 0;
for m = 1 : batchsize
  [imlayers{m}, curloss] = initder2(imlayers{m});
  if (curloss > 0)
    loss2 = loss2 + curloss;
  else
    invnum = invnum + 1;
  end;
end;
if (invnum < batchsize)
  loss2 = loss2 / (batchsize - invnum);
end;
end
```

j. classify_mat.m

```
function pred = classify_mat(layers, weights, params, test_x)
layers = cnnsetup(layers, 0);
layers = setweights(layers, weights);

assert(size(test_x, 1) == layers{1}.mapsize(1) && ...
       size(test_x, 2) == layers{1}.mapsize(2), ...
       'Data and the first layer must have equal sizes');
assert(size(test_x, 3) == layers{1}.outputmaps, ...
       'The number of the input data maps must be as specified');
test_x = normalize(layers{1}, test_x);
layers = initact(layers, test_x);
[~, pred] = forward(layers, 0);
end
```

k. classify_norm_mat.m

```
function pred = classify_norm_mat(layers, weights, test_x)
layers{1} = cnnsetup(layers{1}, 0);
layers{1} = initact(layers{1}, test_x);
layers{2} = cnnsetup(layers{2}, 0);
layers{2} = setweights(layers{2}, weights{2});
assert(size(test_x, 1) == layers{1}{1}.mapsize(1) && ...
       size(test_x, 2) == layers{1}{1}.mapsize(2), ...
       'Data and the first layer must have equal sizes');
assert(size(test_x, 3) == layers{1}{1}.outputmaps, ...
       'The number of the input data maps must be as specified');
test_w = weights{1};
pixels_num = size(test_x, 4);
batchsize = size(test_w, 2);
images = zeros([pixels_num batchsize]);
```

```
for m = 1 : batchsize
  layers{1} = setweights(layers{1}, test_w(:, m));
  [~, pred] = forward(layers{1}, 0);
  images(:, m) = pred;
end;
images_act = reshape(images, [layers{2}{1}.mapsize
layers{2}{1}.outputmaps batchsize]);
layers{2} = initact(layers{2}, images_act);
[layers{2}, pred] = forward(layers{2}, 0);
end
```

l. cnnsetup.m

```
function layers = cnnsetup(layers, isgen)
assert(strcmp(layers{1}.type, 'i'), 'The first layer must be the
type of "i"');
n = numel(layers);
for l = 1 : n   %  layer
  if strcmp(layers{l}.type, 'i') % scaling
    assert(isfield(layers{l}, 'mapsize'), 'The "i" type layer must
contain the "mapsize" field');
    if (~isfield(layers{l}, 'outputmaps'))
      layers{l}.outputmaps = 1;
    end;
    layers{l}.mw = single(zeros([layers{l}.mapsize
layers{l}.outputmaps]));
    layers{l}.sw = single(zeros([layers{l}.mapsize
layers{l}.outputmaps]));
    outputmaps = layers{l}.outputmaps;
    mapsize = layers{l}.mapsize;
  elseif strcmp(layers{l}.type, 'n') % normalization
    layers{l}.w = single(zeros([mapsize outputmaps 2]));
    layers{l}.w(:, :, :, 2) = single(ones([mapsize outputmaps]));
    if (isfield(layers{l}, 'mean'))
      layers{l}.w(:, :, :, 1) = -layers{l}.mean;
    end;
    layers{l}.is_dev = 1;
    if (isfield(layers{l}, 'stdev'))
      if (ischar(layers{l}.stdev) && strcmp(layers{l}.stdev, 'no'))
        layers{l}.is_dev = 0;
        layers{l}.w(:, :, :, 2) = [];
      else
        layers{l}.w(:, :, :, 2) = 1 ./ layers{l}.stdev;
      end;
    end;
    layers{l}.dw = single(zeros(size(layers{l}.w)));
    layers{l}.dw2 = single(zeros(size(layers{l}.w)));
    layers{l}.dwp = single(zeros(size(layers{l}.w)));
    layers{l}.gw = single(ones(size(layers{l}.w)));
  elseif strcmp(layers{l}.type, 'j') % scaling
    assert(isfield(layers{l}, 'mapsize'), 'The "j" type layer must
contain the "mapsize" field');
    mapsize = layers{l}.mapsize;
  elseif strcmp(layers{l}.type, 's') % scaling
    assert(isfield(layers{l}, 'scale'), 'The "s" type layer must
contain the "scale" field');
```

```
    if (~isfield(layers{l}, 'function'))
      layers{l}.function = 'mean';
    end;
    if ~strcmp(layers{l}.function, 'max') &&
~strcmp(layers{l}.function, 'mean')
      error('"%s" - unknown function for the layer %d',
layers{l}.function, l);
    end;
    if (~isfield(layers{l}, 'stride'))
      layers{l}.stride = layers{l}.scale;
    end;
    mapsize = ceil(mapsize ./ layers{l}.stride);
  elseif strcmp(layers{l}.type, 'c') % convolutional
    assert(isfield(layers{l}, 'filtersize'), 'The "c" type layer
must contain the "filtersize" field');
    assert(isfield(layers{l}, 'outputmaps'), 'The "c" type layer
must contain the "outputmaps" field');
    if (~isfield(layers{l}, 'function'))
      layers{l}.function = 'relu';
    end;
    if (~strcmp(layers{l}.function, 'sigm') && ...
       ~strcmp(layers{l}.function, 'relu')) % REctified Linear Unit
      error('"%s" - unknown function for the layer %d',
layers{l}.function, l);
    end;
    if (~isfield(layers{l}, 'padding'))
      layers{l}.padding = [0 0];
    end;
    if (~isfield(layers{l}, 'initstd'))
      layers{l}.initstd = 0.01;
    end;
    if (~isfield(layers{l}, 'biascoef'))
      layers{l}.biascoef = 1;
    end;
    %fan_in = outputmaps * layers{l}.filtersize(1) *
layers{l}.filtersize(2);
    %fan_out = layers{l}.outputmaps * layers{l}.filtersize(1) *
layers{l}.filtersize(2);
    %rand_coef = 2 * sqrt(6 / (fan_in + fan_out));
    layers{l}.k = single(zeros([layers{l}.filtersize outputmaps
layers{l}.outputmaps]));
    layers{l}.dk = single(zeros([layers{l}.filtersize outputmaps
layers{l}.outputmaps]));
    layers{l}.dk2 = single(zeros([layers{l}.filtersize outputmaps
layers{l}.outputmaps]));
    layers{l}.dkp = single(zeros([layers{l}.filtersize outputmaps
layers{l}.outputmaps]));
    layers{l}.gk = single(ones([layers{l}.filtersize outputmaps
layers{l}.outputmaps]));
    if (isgen)
      layers{l}.k = single(randn([layers{l}.filtersize outputmaps
layers{l}.outputmaps]) * layers{l}.initstd);
    else
      layers{l}.k = single(zeros([layers{l}.filtersize outputmaps,
layers{l}.outputmaps]));
    end;
    layers{l}.b = single(zeros(layers{l}.outputmaps, 1));
    layers{l}.db = single(zeros(layers{l}.outputmaps, 1));
```

```
    layers{l}.db2 = single(zeros(layers{l}.outputmaps, 1));
    layers{l}.dbp = single(zeros(layers{l}.outputmaps, 1));
    layers{l}.gb = single(ones(layers{l}.outputmaps, 1));
    mapsize = mapsize + 2*layers{l}.padding - layers{l}.filtersize +
1;
    outputmaps = layers{l}.outputmaps;

  elseif strcmp(layers{l}.type, 'f') % fully connected
    if (~isfield(layers{l}, 'dropout'))
      layers{l}.dropout = 0; % no dropout
    end;
    if (~isfield(layers{l}, 'function'))
      layers{l}.function = 'relu';
    end;
    if (~strcmp(layers{l}.function, 'relu') && ...
        ~strcmp(layers{l}.function, 'sigm') && ...
        ~strcmp(layers{l}.function, 'soft'))
      error('"%s" - unknown function for the layer %d',
layers{l}.function, l);
    end;
    if (~isfield(layers{l}, 'initstd'))
      layers{l}.initstd = 0.1;
    end;
    if (~isfield(layers{l}, 'biascoef'))
      layers{l}.biascoef = 1;
    end;
    assert(isfield(layers{l}, 'length'), 'The "f" type layer must
contain the "length" field');
    weightsize(1) = layers{l}.length;
    if ~strcmp(layers{l-1}.type, 'f')
      maplen = prod(layers{l-1}.mapsize);
      weightsize(2) = maplen * outputmaps;
    else
      weightsize(2) = layers{l-1}.length;
    end;
    layers{l}.weightsize = weightsize;
    if (isgen)
      layers{l}.w = single(randn(weightsize) * layers{l}.initstd);
    else
      layers{l}.w = single(zeros(weightsize));
    end;
    layers{l}.dw = single(zeros(weightsize));
    layers{l}.dw2 = single(zeros(weightsize));
    layers{l}.dwp = single(zeros(weightsize));
    layers{l}.gw = single(ones(weightsize));

    layers{l}.b = single(zeros(1, weightsize(1)));
    layers{l}.db = single(zeros(1, weightsize(1)));
    layers{l}.db2 = single(zeros(1, weightsize(1)));
    layers{l}.dbp = single(zeros(1, weightsize(1)));
    layers{l}.gb = single(ones(1, weightsize(1)));
    mapsize = [0 0];
    outputmaps = 0;
  else
    error('"%s" - unknown type of the layer %d', layers{l}.type, l);
  end
  if (~isfield(layers{l}, 'function'))
    layers{l}.function = 'none';
```

```
  end;
  layers{l}.outputmaps = outputmaps;
  layers{l}.mapsize = mapsize;
  layers{l}.eps = 1e-6;
end
%assert(strcmp(layers{n}.type, 'f'), 'The last layer must be the
type of "f"');
end
```

m. cnntrain_inv_mat.m

```
function [weights, trainerr] = cnntrain_inv_mat(layers, weights,
train_x, train_y, params)
params = setparams(params);
assert(length(layers) == 2);
assert(length(weights) == 2);
layers{1} = cnnsetup(layers{1}, 0);
if (~iscell(train_x))
  layers{1} = initact(layers{1}, train_x);
else
  layers{1} = initact(layers{1}, train_x{1}); % just to get
pixels_num
end;
layers{2} = cnnsetup(layers{2}, 0);
layers{2} = setweights(layers{2}, weights{2});
rng(params.seed);
pixels_num = size(layers{1}{1}.a, 4);
channels_num = layers{1}{end}.length;
train_num = size(train_y, 1);
numbatches = ceil(train_num/params.batchsize);
trainerr = zeros(params.numepochs, numbatches, 2);
for i = 1 : params.numepochs
  if (params.shuffle == 0)
    kk = 1:train_num;
  else
    kk = randperm(train_num);
  end;
  for j = 1 : numbatches
    batch_ind = kk((j-1)*params.batchsize + 1 :
min(j*params.batchsize, train_num));
    if (iscell(train_x))
      batch_x = train_x(batch_ind);
    end;
    batch_y = train_y(batch_ind, :);
    batch_w = weights{1}(:, batch_ind);
    batchsize = length(batch_ind);
    imlayers = cell(1, batchsize);
    % first pass
    images = zeros([pixels_num channels_num batchsize]);
    for m = 1 : batchsize
      imlayers{m} = setweights(layers{1}, batch_w(:, m));
      if (iscell(train_x))
        imlayers{m} = initact(imlayers{m}, batch_x{m});
      end;
      [imlayers{m}, pred] = forward(imlayers{m}, 1);
      images(:, :, m) = pred;
    end;
```

```
    images_act = reshape(images, [layers{2}{1}.mapsize channels_num
batchsize]);
    layers{2} = initact(layers{2}, images_act);
    layers{2} = updateweights(layers{2}, params, i, 0); %
preliminary update
    [layers{2}, pred1] = forward(layers{2}, 1);
    %disp(['batch: ' num2str(j)]);
    %disp(pred1(1,1:5));
    % second pass
    [layers{2}, loss] = initder(layers{2}, batch_y);
    trainerr(i, j, 1) = loss;
    % disp(loss);
    %layersff{1} = imlayers;
    %layersff{2} = layers{2};
    layers{2} = backward(layers{2});
    layers{2} = calcweights(layers{2});
    images_der = reshape(layers{2}{1}.d, [pixels_num channels_num
batchsize]);
    for m = 1 : batchsize
      imlayers{m}{end}.d = images_der(:, :, m);
      imlayers{m} = backward(imlayers{m});
    end;
    % third pass
    loss2 = 0;
    for m = 1 : batchsize
      [imlayers{m}, curloss] = initder2(imlayers{m});
      [imlayers{m}, pred] = forward(imlayers{m}, 3);
      images(:, :, m) = pred;
      loss2 = loss2 + curloss;
    end;
    loss2 = loss2 / batchsize;
    trainerr(i, j, 2) = loss2;
    images_act = reshape(images, [layers{2}{1}.mapsize channels_num
batchsize]);
    layers{2} = initact(layers{2}, images_act);
    [layers{2}, pred2] = forward(layers{2}, 3);
    %disp(['loss2: ' num2str(loss2)]);
    layers{2} = calcweights2(layers{2});
    %disp(['pred2: ' num2str(pred2(1,1:5))]);
    layers{2} = updateweights(layers{2}, params, i, 1);
    if (params.verbose == 2)
      disp(['Epoch: ' num2str(i) ', batch: ', num2str(j)]);
    end;
  end
  if (params.verbose == 1)
    disp(['Epoch: ' num2str(i)]);
  end;
end
weights{2} = getweights(layers{2});
trainerr = permute(trainerr, [2 1 3]);
end
```

n. cnntrain_inv_mat.m

```
function [weights, trainerr] = cnntrain_inv_mat(layers, weights,
train_x, train_y, params)
params = setparams(params);
assert(length(layers) == 2);
assert(length(weights) == 2);
layers{1} = cnnsetup(layers{1}, 0);
if (~iscell(train_x))
  layers{1} = initact(layers{1}, train_x);
else
  layers{1} = initact(layers{1}, train_x{1}); % just to get
pixels_num
end;
layers{2} = cnnsetup(layers{2}, 0);
layers{2} = setweights(layers{2}, weights{2});

rng(params.seed);
pixels_num = size(layers{1}{1}.a, 4);
channels_num = layers{1}{end}.length;
train_num = size(train_y, 1);
numbatches = ceil(train_num/params.batchsize);
trainerr = zeros(params.numepochs, numbatches, 2);
for i = 1 : params.numepochs
  if (params.shuffle == 0)
    kk = 1:train_num;
  else
    kk = randperm(train_num);
  end;
  for j = 1 : numbatches
    batch_ind = kk((j-1)*params.batchsize + 1 :
min(j*params.batchsize, train_num));
    if (iscell(train_x))
      batch_x = train_x(batch_ind);
    end;
    batch_y = train_y(batch_ind, :);
    batch_w = weights{1}(:, batch_ind);
    batchsize = length(batch_ind);
    imlayers = cell(1, batchsize);
    % first pass
    images = zeros([pixels_num channels_num batchsize]);
    for m = 1 : batchsize
      imlayers{m} = setweights(layers{1}, batch_w(:, m));
      if (iscell(train_x))
        imlayers{m} = initact(imlayers{m}, batch_x{m});
      end;
      [imlayers{m}, pred] = forward(imlayers{m}, 1);
      images(:, :, m) = pred;
    end;
    images_act = reshape(images, [layers{2}{1}.mapsize channels_num
batchsize]);
    layers{2} = initact(layers{2}, images_act);
    layers{2} = updateweights(layers{2}, params, i, 0); %
preliminary update
    [layers{2}, pred1] = forward(layers{2}, 1);
    %disp('pred1');
    %disp(pred1(1:10,1));
```

```
    % second pass
    [layers{2}, loss] = initder(layers{2}, batch_y);
    trainerr(i, j, 1) = loss;
    % disp(loss);
    %layersff{1} = imlayers;
    %layersff{2} = layers{2};
    layers{2} = backward(layers{2});
    layers{2} = calcweights(layers{2});
    images_der = reshape(layers{2}{1}.d, [pixels_num channels_num
batchsize]);
    for m = 1 : batchsize
      imlayers{m}{end}.d = images_der(:, :, m);
      imlayers{m} = backward(imlayers{m});
    end;
    % third pass
    loss2 = 0; invnum = 0;
    invalid = zeros(batchsize, 1);
    for m = 1 : batchsize
      [imlayers{m}, curloss] = initder2(imlayers{m});
      if (curloss > 0)
        [imlayers{m}, pred] = forward(imlayers{m}, 3);
        images(:, :, m) = pred;
        loss2 = loss2 + curloss;
      else
        invnum = invnum + 1;
        invalid(invnum) = m;
      end;
    end;
    %disp(sum(sum(images)));
    invalid(invnum+1 : end) = [];
    if (invnum < batchsize)
      loss2 = loss2 / (batchsize - invnum);
      trainerr(i, j, 2) = loss2;
      images_act = reshape(images, [layers{2}{1}.mapsize
channels_num batchsize]);
      layers{2} = initact(layers{2}, images_act);
      [layers{2}, pred2] = forward(layers{2}, 3);
    end;
    %disp(['loss2: ' num2str(loss2)]);
    layers{2} = calcweights2(layers{2}, invalid);
    %disp(['pred2: ' num2str(pred2(1,1:5))]);
    layers{2} = updateweights(layers{2}, params, i, 1);
    if (params.verbose == 2)
      disp(['Epoch: ' num2str(i) ', batch: ', num2str(j)]);
    end;
  end
  if (params.verbose == 1)
    disp(['Epoch: ' num2str(i)]);
  end;
end
weights{2} = getweights(layers{2});
trainerr = permute(trainerr, [2 1 3]);
end
```

o. cnntrain_mat.m

```
function [weights, trainerr] = cnntrain_mat(layers, weights, params,
train_x, train_y)
params = setparams(params);
layers = cnnsetup(layers, 0);
assert(length(size(train_y)) == 2, 'The label array must have 2
dimensions');
train_num = size(train_y, 1);
classes_num = size(train_y, 2);
assert(classes_num == layers{end}.length, 'Labels and last layer
must have equal number of classes');
if (params.balance == 1)
  layers{end}.coef = (ones(1, classes_num) ./ mean(train_y, 1)) /
classes_num;
elseif (params.balance == 0)
  layers{end}.coef = ones(1, classes_num);
end;
layers = setweights(layers, weights);

assert(size(train_x, 1) == layers{1}.mapsize(1) && ...
       size(train_x, 2) == layers{1}.mapsize(2), ...
       'Data and the first layer must have equal sizes');
assert(size(train_x, 3) == layers{1}.outputmaps, ...
       'The number of the input data maps must be as specified');
assert(size(train_x, 4) == train_num, ...
       'Data and labels must have equal number of objects');
train_x = normalize(layers{1}, train_x);
layers{1} = initnorm(layers{1}, train_x);
rng(params.seed);
numbatches = ceil(train_num/params.batchsize);
trainerr = zeros(params.numepochs, 2);
for epoch = 1 : params.numepochs
  if (length(params.beta) == 1)
    beta = params.beta;
  else
    beta = params.beta(epoch);
  end;
  if (params.shuffle == 0)
    kk = 1:train_num;
  else
    kk = randperm(train_num);
  end;
  for batch = 1 : numbatches
    batch_x = train_x(:, :, :, kk((batch-1)*params.batchsize + 1 :
min(batch*params.batchsize, train_num)));
    batch_y = train_y(kk((batch-1)*params.batchsize + 1 :
min(batch*params.batchsize, train_num)), :);
    % first pass
    layers = initact(layers, batch_x);
    layers = updateweights(layers, params, epoch, 0); % preliminary
update
    [layers, pred] = forward(layers, 1);
    %disp(['pred: ' num2str(pred(1,1:5))]);
    % second pass
    [layers, loss] = initder(layers, params, batch_y);
    trainerr(epoch, 1) = trainerr(epoch, 1) + loss;
```

```
    %disp(['loss: ' num2str(loss)]);
    layers = backward(layers, params);
    layers = calcweights(layers, 2);
    % third pass
    [layers, loss2] = initder2(layers);
    trainerr(epoch, 2) = trainerr(epoch, 2) + loss2;
    %disp(['loss2: ' num2str(loss2)]);
    if (beta > 0)
      [layers, pred2] = forward(layers, 3);
      %disp(['pred2: ' num2str(pred2(1,1:5))]);
      layers = calcweights(layers, 3);
    end;
    layers = updateweights(layers, params, epoch, 1); % final update
    if (params.verbose == 2)
      disp(['Epoch: ' num2str(epoch) ', batch: ', num2str(batch)]);
    end;
  end
  trainerr = trainerr / numbatches;
  if (params.verbose == 1)
    disp(['Epoch: ' num2str(epoch)]);
  end;
end
weights = getweights(layers);
end
```

p. classify_mat.m

```
function [weights, trainerr] = cnntrain_norm_mat(layers, weights,
train_x, train_y, params)
params = setparams(params);
assert(length(layers) == 3);
assert(length(weights) == 3);
layers{1} = cnnsetup(layers{1}, 0);
layers{1} = setweights(layers{1}, weights{1});
layers{2} = cnnsetup(layers{2}, 0);
layers{3} = cnnsetup(layers{3}, 0);
layers{3} = setweights(layers{3}, weights{3});
mapsize = layers{1}{1}.mapsize;
pixels_num = prod(mapsize);
dimens_num = layers{1}{1}.outputmaps;
channels_num = layers{2}{end}.length;
imnetsize = [1 dimens_num 1 pixels_num];
train_num = size(train_y, 1);
numbatches = ceil(train_num/params.batchsize);
trainerr = zeros(params.numepochs, numbatches);
for i = 1 : params.numepochs
  if (params.shuffle == 0)
    kk = 1:train_num;
  else
    kk = randperm(train_num);
  end;
  for j = 1 : numbatches

    batch_ind = kk((j-1)*params.batchsize + 1 :
min(j*params.batchsize, train_num));
    batchsize = length(batch_ind);
```

```
    if (iscell(train_x))
      batch_x = zeros([mapsize dimens_num batchsize]);
      for m = 1 : batchsize
        cellbatch = train_x(batch_ind);
        batch_x(:, :, :, m) = cellbatch{m};
      end;
    else
      batch_x = repmat(train_x, [1 1 1 batchsize]);
    end;
    batch_y = train_y(batch_ind, :);
    batch_w = weights{2}(:, batch_ind);
    imlayers = cell(1, batchsize);
    % first pass
    layers{1} = initact(layers{1}, batch_x);
    layers{1} = updateweights(layers{1}, params, i, 0); %
preliminary update
    [layers{1}, pred1] = forward(layers{1}, 1);
    pred1 = permute(pred1, [2 1 3 4]);
    pred1 = reshape(pred1, [pixels_num dimens_num batchsize]);
    images_act = zeros([mapsize channels_num batchsize]);
    for m = 1 : batchsize
      imlayers{m} = setweights(layers{2}, batch_w(:, m));
      coord_act = reshape(pred1(:, :, m)', imnetsize);
      imlayers{m} = initact(imlayers{m}, coord_act);
      [imlayers{m}, pred] = forward(imlayers{m}, 1);
      pred = reshape(pred, [mapsize channels_num]);
      images_act(:, :, :, m) = permute(pred, [2 1 3]);
    end;
    %load('C:\Users\sergeyd\Workspaces\Normalization\WeightsIn',
'batch_x');
    %disp(sum(abs(batch_x(:) - images_act(:))));
    layers{3} = initact(layers{3}, images_act);
    layers{3} = updateweights(layers{3}, params, i, 0); %
preliminary update
    [layers{3}, pred3] = forward(layers{3}, 1);
    %load('C:\Users\sergeyd\Workspaces\Normalization\WeightsIn',
'pred');
    %disp(sum(abs(pred3(:) - pred(:))));
    %disp(pred1(1:10,1));

    % second pass
    [layers{3}, loss] = initder(layers{3}, batch_y);
    trainerr(i, j) = loss;
    % disp(loss);
    layers{3} = backward(layers{3});
    layers{3} = calcweights(layers{3});
    layers{3} = updateweights(layers{3}, params, i, 1);
    %images_der = layers{3}{1}.d;
    images_der = permute(layers{3}{1}.d, [2 1 3 4]);
    images_der = reshape(images_der, [pixels_num channels_num
batchsize]);
    coord_der = zeros([mapsize dimens_num batchsize]);
    for m = 1 : batchsize
      imlayers{m}{end}.d = images_der(:, :, m);
      imlayers{m} = backward(imlayers{m});
      curder = reshape(imlayers{m}{1}.d, [dimens_num pixels_num]);
      curder = reshape(curder', [mapsize(2) mapsize(1) dimens_num]);
      %coord_der(:, :, :, m) = curder;
```

```
      coord_der(:, :, :, m) = permute(curder, [2 1 3]);
    end;
    layers{1}{end}.d = coord_der;
    layers{1} = backward(layers{1});
    layers{1} = calcweights(layers{1});
    layers{1} = updateweights(layers{1}, params, i, 1);

    if (params.verbose == 2)
      disp(['Epoch: ' num2str(i) ', batch: ', num2str(j)]);
    end;
  end
  if (params.verbose == 1)
    disp(['Epoch: ' num2str(i)]);
  end;
end
weights{1} = getweights(layers{1});
weights{3} = getweights(layers{3});
trainerr = trainerr';
end
```

q. expand.m

```
function B = expand(varargin)
%EXPAND Replicate and tile each element of an array, similar to
repmat.
% EXPAND(A,SZ), for array A and vector SZ replicates each element of
A by
% SZ.  The results are tiled into an array in the same order as the
% elements of A, so that the result is size:  size(A).*SZ. Therefore
the
% number of elements of SZ must equal the number of dimensions of A,
or in
% MATLAB syntax: length(size(A))==length(SZ) must be true.
% The result will have the same number of dimensions as does A.
% There is no restriction on the number of dimensions for input A.
%
% Examples:
%
%   A = [1 2; 3 4]; % 2x2
%   SZ = [6 5];
%   B = expand(A,[6 5]) % Creates a 12x10 array.
%
%   The following demonstrates equivalence of EXPAND and expansion
acheived
%   through indexing the individual elements of the array:
%
%   A = 1; B = 2; C = 3; D = 4;  % Elements of the array to be
expanded.
%   Mat = [A B;C D];  % The array to expand.
%   SZ = [2 3];  % The expansion vector.
%   ONES = ones(SZ);  % The index array.
%   ExpMat1 = [A(ONES),B(ONES);C(ONES),D(ONES)]; % Element
expansion.
%   ExpMat2 = expand(Mat,SZ); % Calling EXPAND.
%   isequal(ExpMat1,ExpMat2)  % Yes
%
```

```
%
% See also, repmat, meshgrid, ones, zeros, kron
%
% Author: Matt Fig
% Date: 6/20/2009
% Contact:  popkenai@yahoo.com

if (nargin < 2 || nargin > 3)
    error('Wrong number of arguments.  See help.');
end
A = varargin{1};
sc = varargin{2};
if (nargin == 3)
  st = varargin{3};
end;

SA = size(A);  % Get the size (and number of dimensions) of input.
if (length(SA) < length(sc))
  SA = [SA ones(1, length(sc) - length(SA))];
end;

if length(SA) ~= length(sc)
   error('Length of size vector must equal ndims(A).  See help.')
elseif any(sc ~= floor(sc))
   error('The size vector must contain integers only.  See help.')
end

T = cell(length(SA), 1);
for ii = length(SA) : -1 : 1
    H = zeros(SA(ii) * sc(ii), 1);   %  One index vector into A for
each dim.
    H(1 : sc(ii) : SA(ii) * sc(ii)) = 1;   %  Put ones in correct
places.
    T{ii} = cumsum(H);   %  Cumsumming creates the correct order.
end

B = A(T{:});   %  Feed the indices into A.
end
```

r. filtn.m

```
function c = filtn(a, b, type)
   c = convn(a, flipall(b), type);
end
```

s. flipall.m

```
function X = flipall(X)
  for dimind = 1 : ndims(X)
    X = flip(X, dimind);
  end
end
```

t. forward.m

```
function [layers, pred] = forward(layers, passnum)

n = numel(layers);
batchsize = size(layers{1}.a, 4); % number of examples in the
minibatch

for l = 1 : n   %  for each layer

  if strcmp(layers{l}.type, 'i')
    if (passnum == 3)
      continue;
    end;
    if (isfield(layers{l}, 'mean'))
      layers{l}.a = layers{l}.a + repmat(layers{l}.mw, [1 1 1
batchsize]);
    end;
    if (isfield(layers{l}, 'maxdev'))
      layers{l}.a = layers{l}.a .* repmat(layers{l}.sw, [1 1 1
batchsize]);
    end;

  elseif strcmp(layers{l}.type, 'j')
    assert(0, 'Jittering is not implemented in Matlab version');

  elseif strcmp(layers{l}.type, 'n')
    layers{l}.a = layers{l-1}.a + repmat(layers{l}.w(:, :, :, 1), [1
1 1 batchsize]);
    if (layers{l}.is_dev == 1)
      layers{l}.a = layers{l}.a .* repmat(layers{l}.w(:, :, :, 2),
[1 1 1 batchsize]);
    end;

  elseif strcmp(layers{l}.type, 'c')
    if (passnum == 0 || passnum == 1)
      layers{l}.a = repmat(permute(layers{l}.b, [3 4 1 2]),
[layers{l}.mapsize 1 batchsize]);
    elseif (passnum == 3)
      a = layers{l}.a;
      layers{l}.a = zeros([layers{l}.mapsize layers{l}.outputmaps
batchsize]);
    end;
    as = size(layers{l-1}.a); as(end+1:4) = 1;
    a_prev = layers{l-1}.a;
    if (layers{l}.padding(1) > 0 || layers{l}.padding(2) > 0)
      padding = layers{l}.padding;
      a_prev = zeros([as(1:2) + 2*padding as(3:4)]);
      a_prev(padding(1)+1:padding(1)+as(1),
padding(2)+1:padding(2)+as(2), :, :) =  layers{l-1}.a;
    end;
    for i = 1 : layers{l}.outputmaps
      for j = 1 : layers{l-1}.outputmaps
        layers{l}.a(:, :, i, :) = layers{l}.a(:, :, i, :) + ...
          filtn(a_prev(:, :, j, :), layers{l}.k(:, :, j, i),
'valid');
```

```
      end
    end
    layers{l}.a(-layers{l}.eps < layers{l}.a & layers{l}.a <
layers{l}.eps) = 0;
    %{
    disp(layers{l-1}.a(1, 1:5, 1, 1));
    disp(layers{l}.k(1, 1:4, 1, 1));
    disp(layers{l}.k(1:4, 1, 1, 1));
    disp(layers{l}.a(1, 1:5, 1, 1));
    %}

  elseif strcmp(layers{l}.type, 's')
    sc = [layers{l}.scale 1 1];
    st = [layers{l}.stride 1 1];
    mapsize = layers{l-1}.mapsize;
    newsize = layers{l}.mapsize;
    b = strel('rectangle', [sc(1) sc(2)]);
    fi = ceil((sc+1)/2);
    if strcmp(layers{l}.function, 'max')
      if (passnum == 0 || passnum == 1)
        %layers{l}.a{j} = maxscale(layers{l-1}.a{j}, s);
        a = double(zeros(fi(1)+st(1)*(newsize(1)-1),
fi(2)+st(2)*(newsize(2)-1), layers{l}.outputmaps, batchsize));
        a(1:mapsize(1), 1:mapsize(2), :, :) = layers{l-1}.a;
        z = imdilate(a, b);
        layers{l}.a = z(fi(1):st(1):end, fi(2):st(2):end, :, :);
      elseif (passnum == 3)
        curval = expand(layers{l}.d, sc);
        prevval = stretch(layers{l-1}.d, sc, st);
        maxmat = (prevval == curval);
        %maxmat = uniq(maxmat, sc);
        curder = stretch(layers{l-1}.a, sc, st);
        curder = curder .* maxmat;
        z = convn(curder, ones(sc(1:2)), 'valid');
        layers{l}.a = z(1:sc(1):end, 1:sc(2):end, :, :);
      end;

    elseif strcmp(layers{l}.function, 'mean')
      a = double(zeros([(newsize-1).*st(1:2)+sc(1:2)
layers{l}.outputmaps batchsize]));
      a(1:mapsize(1), 1:mapsize(2), :, :) = layers{l-1}.a;
      if (size(a, 1) > mapsize(1))
        meanvals = mean(a((newsize(1)-1)*st(1)+1 : mapsize(1), :, :,
:), 1);
        extind = mapsize(1)+1 : size(a, 1);
        a(extind, :, :, :) = repmat(meanvals, [length(extind) 1 1
1]);
      end;
      if (size(a, 2) > mapsize(2))
        meanvals = mean(a(:, (newsize(2)-1)*st(2)+1 : mapsize(2), :,
:), 2);
        extind = mapsize(2)+1 : size(a, 2);
        a(:, extind, :, :) = repmat(meanvals, [1 length(extind) 1
1]);
      end;
      z = convn(a, ones(sc(1:2)) / prod(sc), 'valid');
      layers{l}.a = z(1:st(1):end, 1:st(2):end, :, :);
    end;
```

```
  elseif strcmp(layers{l}.type, 'f')
    %  concatenate all end layer feature maps into vector
    if strcmp(layers{l-1}.type, 'f')
      layers{l}.ai = layers{l-1}.a;
    else
      %a_trans = permute(layers{l-1}.a, [4 1 2 3]);
      %disp('prev_layer->activ_mat');
      %disp(a_trans(1, 1:5, 1, 1));
      %disp('weights');
      %disp(layers{l}.w(1, 1:5));
      layers{l}.ai = reshape(layers{l-1}.a, layers{l}.weightsize(2),
batchsize)';
    end;
    if (passnum == 0 || passnum == 1)
      layers{l}.a = bsxfun(@plus, layers{l}.ai * layers{l}.w',
layers{l}.b);
    elseif (passnum == 3)
      a = layers{l}.a;
      layers{l}.a = layers{l}.ai * layers{l}.w';
    end;
    if (layers{l}.dropout > 0) % dropout
      if (passnum == 1) % training 1
        dropmat = rand(batchsize, layers{l}.length);
        dropmat(dropmat < layers{l}.dropout) = 0;
        dropmat(dropmat > 0) = 1;
        layers{l}.dropmat = dropmat;
        layers{l}.a = layers{l}.a .* dropmat;
      elseif (passnum == 3)
        layers{l}.a = layers{l}.a .* dropmat;
      elseif (passnum == 0) % testing
        layers{l}.a = layers{l}.a * (1 - layers{l}.dropout);
      end;
    end;
    layers{l}.a(-layers{l}.eps < layers{l}.a & layers{l}.a <
layers{l}.eps) = 0;
    %disp('activ_mat');
    %disp(layers{l}.a(1, 1:5, 1, 1));
  end


  if strcmp(layers{l}.type, 'c') || strcmp(layers{l}.type, 'f')
    if (passnum == 0 || passnum == 1)
      if strcmp(layers{l}.function, 'soft')
        layers{l}.a = soft(layers{l}.a);
      elseif strcmp(layers{l}.function, 'sigm')
        layers{l}.a = sigm(layers{l}.a);
      elseif strcmp(layers{l}.function, 'relu')
        layers{l}.a = max(layers{l}.a, 0);
      end;
    elseif (passnum == 3)
      if strcmp(layers{l}.function, 'soft')
        layers{l}.a = softder(layers{l}.a, layers{l}.d);
      elseif strcmp(layers{l}.function, 'sigm')
        layers{l}.a = layers{l}.a .* a .* (1 - a);
      elseif strcmp(layers{l}.function, 'relu')
```

```
        layers{l}.a = layers{l}.a .* (a > 0);
      end;
    end;
    if (strcmp(layers{l}.function, 'soft') ||
strcmp(layers{l}.function, 'sigm'))
      layers{l}.a(-layers{l}.eps < layers{l}.a & layers{l}.a <
layers{l}.eps) = 0;
    end;
  end;
end

pred = layers{n}.a;

end
```

u. genweights_mat.m

```
function weights = genweights_mat(layers, params)
rng(params.seed);
layers = cnnsetup(layers, 1);
weights = getweights(layers);
end
```

v. getweights.m

```
function der = getweights(layers)

n = numel(layers);
ind = 0;
der = double([]);
for l = 1 : n   %  layer
  if strcmp(layers{l}.type, 'i') % input
    if (isfield(layers{l}, 'mean'))
      curlen = numel(layers{l}.mw);
      %w_trans = permute(layers{l}.mw, [2 1 3 4]);
      %der(ind+1:ind+curlen, 1) = w_trans(:);
      der(ind+1:ind+curlen, 1) = layers{l}.mw(:);
      ind = ind + curlen;
    end;
    if (isfield(layers{l}, 'maxdev'))
      curlen = numel(layers{l}.sw);
      %w_trans = permute(layers{l}.sw, [2 1 3 4]);
      %der(ind+1:ind+curlen, 1) = w_trans(:);
      der(ind+1:ind+curlen, 1) = layers{l}.sw(:);
      ind = ind + curlen;
    end;
  elseif strcmp(layers{l}.type, 'n') % normalization
    curlen = numel(layers{l}.w);
    w_trans = permute(layers{l}.w, [2 1 3 4]);
    der(ind+1:ind+curlen, 1) = w_trans(:);
    ind = ind + curlen;
  elseif strcmp(layers{l}.type, 'c') % convolutional
    curlen = length(layers{l}.k(:));
    k_trans = permute(layers{l}.k, [4 1 2 3]);
```

```
    der(ind+1:ind+curlen, 1) = k_trans(:);
    %der(ind+1:ind+curlen, 1) = layers{l}.k(:);
    ind = ind + curlen;
    curlen = numel(layers{l}.b);
    der(ind+1:ind+curlen, 1) = layers{l}.b(:);
    ind = ind + curlen;
  elseif strcmp(layers{l}.type, 'f') % fully connected
    curlen = numel(layers{l}.w);
    %w_trans = layers{l}.w';
    %der(ind+1:ind+curlen, 1) = w_trans(:);
    der(ind+1:ind+curlen, 1) = layers{l}.w(:);
    ind = ind + curlen;
    curlen = numel(layers{l}.b);
    der(ind+1:ind+curlen, 1) = layers{l}.b(:);
    ind = ind + curlen;
  end;
end

end
```

w. initact.m

```
function layers = initact(layers, data)

layers{1}.a = data;
layers{1}.a(-layers{1}.eps < layers{1}.a & layers{1}.a <
layers{1}.eps) = 0;
end
```

x. classify_mat.m

```
function [layers, loss] = initder(layers, params, y)
n = numel(layers);
assert(strcmp(layers{n}.function, 'sigm') ||
strcmp(layers{n}.function, 'soft'), ...
  'The last layer function must be either "soft" or "sigm"');
batchsize = size(y, 1); % number of examples in the minibatch
if (strcmp(params.lossfun, 'logreg'))
  lossmat = layers{n}.a;
  lossmat(y == 0) = 1;
  lossmat(lossmat == 0) = layers{n}.eps;
  if (strcmp(layers{n}.function, 'soft'))
    layers{n}.d = layers{n}.a - y;
  else
    layers{n}.d = -y ./ lossmat;
  end;
  loss = -sum(log(lossmat(:))) / batchsize;
else
  layers{n}.d = layers{n}.a - y;
  loss = 1/2 * sum(layers{n}.d(:).^2) / batchsize;
end;
layers{n}.d(-layers{n}.eps < layers{n}.d & layers{n}.d <
layers{n}.eps) = 0;
end
```

y. classify_mat.m

```
function [layers, loss] = initder_dir_coord_inv(layers)
activ = zeros(size(layers{1}.a));
pixels_num = size(layers{1}.d, 4);
selfprod = sum(sum(layers{1}.d.^2));
if (selfprod < layers{1}.eps)
  layers{1}.a = zeros(size(layers{1}.a));
  loss = 0; return;
end;
loss = 0;

norm_shift = sqrt(pixels_num);
products_shift = sum(layers{1}.d, 4) ./ norm_shift;
activ = activ + repmat(products_shift ./ norm_shift, [1 1 1
pixels_num]);
loss = loss + sum(products_shift.^2);

vectors_scale = layers{1}.a - repmat(mean(layers{1}.a, 4), [1 1 1
pixels_num]);
norm_scale_sq = sum(vectors_scale.^2, 4);
norm_scale = sqrt(norm_scale_sq);
products_scale = sum(layers{1}.d .* vectors_scale, 4) ./ norm_scale;
activ = activ + vectors_scale .* repmat(products_scale ./
norm_scale, [1 1 1 pixels_num]);
loss = loss + sum(products_scale.^2);

vectors_rot = layers{1}.a(:, [2 1], :, :);
vectors_rot(:, 2, :, :) = -vectors_rot(:, 2, :, :);
vectors_rot = vectors_rot - repmat(mean(vectors_rot, 4), [1 1 1
pixels_num]);
rot_scale_prod = sum(vectors_rot .* vectors_scale, 4) ./
norm_scale_sq;
vectors_rot = vectors_rot - vectors_scale .* repmat(rot_scale_prod,
[1 1 1 pixels_num]);
norm_rot = sqrt(sum(sum(vectors_rot.^2)));
products_rot = sum(sum(layers{1}.d .* vectors_rot)) / norm_rot;
activ = activ + vectors_rot * products_rot / norm_rot;
loss = loss + products_rot^2;

loss = sqrt(loss / selfprod);
layers{1}.a = (activ / loss - layers{1}.d * loss) / selfprod;
layers{1}.a(-layers{1}.eps < layers{1}.a & layers{1}.a <
layers{1}.eps) = 0;

%ind = 1 : size(layers{1}.a, 1) * size(layers{1}.a, 2);
%ind = reshape(ind, size(layers{1}.a, 1), size(layers{1}.a, 2))';
%a = layers{1}.a(:,:,:,ind(:));

end
```

z. initder_dir_int_inv.m

```
function [layers, loss] = initder_dir_int_inv(layers)

selfprod = squeeze(sum(sum(sum(layers{1}.d.^2, 1), 2), 3));
sig_ind = find(selfprod >= layers{1}.eps);
sig_num = length(sig_ind);
selfprod = selfprod(sig_ind);
sigder = layers{1}.d(:, :, :, sig_ind);

activ_size = [size(layers{1}.a, 1) size(layers{1}.a, 2)
size(layers{1}.a, 3)];
batchsize = size(layers{1}.a, 4);
activ = zeros([activ_size sig_num]);
sigloss = zeros(sig_num, 1);

vectors_bright = ones([activ_size sig_num]);
norm_int = sqrt(sum(sum(sum(vectors_bright.^2, 1), 2), 3));
vectors_bright = vectors_bright ./ repmat(norm_int, [activ_size 1]);
products_bright = sum(sum(sum(sigder .* vectors_bright, 1), 2), 3);
activ = activ + vectors_bright .* repmat(products_bright,
[activ_size 1]);
sigloss = sigloss + squeeze(products_bright.^2);

vectors_bright = layers{1}.a(:, :, :, sig_ind);
% to make it orthogonal to uniform vector
vectors_mean = sum(sum(sum(vectors_bright, 1), 2), 3) /
prod(activ_size);
vectors_bright = vectors_bright - repmat(vectors_mean, [activ_size
1]);
norm_int = sqrt(sum(sum(sum(vectors_bright.^2, 1), 2), 3));
vectors_bright = vectors_bright ./ repmat(norm_int, [activ_size 1]);
products_bright = sum(sum(sum(sigder .* vectors_bright, 1), 2), 3);
activ = activ + vectors_bright .* repmat(products_bright,
[activ_size 1]);
sigloss = sigloss + squeeze(products_bright.^2);

sigloss = sqrt(sigloss ./ selfprod);
loss = zeros(batchsize, 1);
loss(sig_ind) = sigloss;
sigloss = repmat(shiftdim(sigloss, -3), [activ_size 1]);
selfprod = repmat(shiftdim(selfprod, -3), [activ_size 1]);
activ = (activ ./ sigloss - sigder .* sigloss) ./ selfprod;
layers{1}.a = zeros(size(layers{1}.a));
layers{1}.a(:, :, :, sig_ind) = activ;

layers{1}.a(-layers{1}.eps < layers{1}.a & layers{1}.a <
layers{1}.eps) = 0;

%ind = 1 : size(layers{1}.a, 1) * size(layers{1}.a, 2);
%ind = reshape(ind, size(layers{1}.a, 1), size(layers{1}.a, 2))';
%a = layers{1}.a(:,:,:,ind(:));

end
```

aa. initder2.m

```
function [layers, loss] = initder2(layers)

batchsize = size(layers{1}.d, 4);
deriv_sq = layers{1}.d.^2;
loss = sum(deriv_sq(:)) / (2 * batchsize);
layers{1}.a = layers{1}.d;

end
```

bb. initnorm.m

```
function [firstlayer] = initnorm(firstlayer, train_x)

if (isfield(firstlayer, 'mean'))
  firstlayer.mw = firstlayer.mean - mean(train_x, 4);
end;
if (isfield(firstlayer, 'maxdev'))
  stdev = std(train_x, 1, 4);
  stdev(stdev <= firstlayer.maxdev) = firstlayer.maxdev;
  firstlayer.sw = firstlayer.maxdev * ones([firstlayer.mapsize
firstlayer.outputmaps]);
  firstlayer.sw = firstlayer.sw ./ stdev;
end;
end
```

cc. logsumexp.m

```
function lcs = logsumexp(X, varargin)
% LOGSUMEXP(X, dim) computes log(sum(exp(X), dim)) robustly. Care
lightspeed users!
%
%     lse = logsumexp(X[, dim]);
%
% This routine works with general ND-arrays and matches Matlab's
default
% behavior for sum: if dim is omitted it sums over the first non-
singleton
% dimension.
%
% Note: Tom Minka's lightspeed has a logsumexp function, which:
%     1) sets dim=1 if dim is missing
%     2) returns Inf for sums containing Infs and NaNs;
%
% This routine is fairly fast and accurate for many uses, including
when all the
% values of X are large in magnitude. There is a corner case where
the relative
% error is avoidably bad (although the absolute error is small),
when the
% largest argument is very close to zero and the next largest is
moderately
```

```
% negative. For example:
%     logsumexp([0 -40])
% Cases like this rarely come up in my work. My LOGPLUSEXP and
LOGCUMSUM
% functions do cover this case.
%
% SEE ALSO: LOGCUMSUMEXP LOGPLUSEXP

% Iain Murray, September 2010

% History: IM wrote a bad logsumexp in ~2002, then used Tom Minka's
version for
% years until eventually wanting something slightly different.

if (numel(varargin) > 1)
    error('Too many arguments')
end

if isempty(X)
    % Easiest way to get this trivial but annoying case right!
    lcs = log(sum(exp(X),varargin{:}));
    return;
end

if isempty(varargin)
    mx = max(X);
else
    mx = max(X, [], varargin{:});
end
Xshift = bsxfun(@minus, X, mx);
lcs = bsxfun(@plus, log(sum(exp(Xshift),varargin{:})), mx);

idx = isinf(mx);
lcs(idx) = mx(idx);
lcs(any(isnan(X),varargin{:})) = NaN;
```

dd. normalize.m

```
function [train_x] = normalize(firstlayer, train_x)

if (isfield(firstlayer, 'norm'))
  datanorm = sqrt(sum(sum(sum(train_x.^2, 1), 2), 3));
  datanorm(datanorm <= firstlayer.eps) = firstlayer.norm;
  train_x = train_x ./ repmat(datanorm, [firstlayer.mapsize
firstlayer.outputmaps 1]);
  train_x = train_x * firstlayer.norm;
end;

end
```

ee. setparams.m

```
function [params] = setparams(params)

if (~isfield(params, 'batchsize'))
  params.batchsize = 128;
end;
if (~isfield(params, 'numepochs'))
  params.numepochs = 1;
end;
if (~isfield(params, 'balance'))
  params.balance = 0;
end;
if (~isfield(params, 'alpha'))
  params.alpha = 1;
end;
if (~isfield(params, 'beta'))
  params.beta = 0;
end;
if (~isfield(params, 'momentum'))
  params.momentum = 0;
end;
if (~isfield(params, 'adjustrate'))
  params.adjustrate = 0;
end;
if (~isfield(params, 'maxcoef'))
  params.maxcoef = 1;
  params.mincoef = 1;
else
  params.mincoef = 1 / params.maxcoef;
end;
if (~isfield(params, 'shuffle'))
  params.shuffle = 0;
end;
if (~isfield(params, 'lossfun'))
  params.lossfun = 'squared';
end;
if (~isfield(params, 'verbose'))
  params.verbose = 0;
end;
if (~isfield(params, 'seed'))
  params.seed = 0;
end;

end
```

ff. setweights.m

```
function layers = setweights(layers, weights)

n = numel(layers);
ind = 0;
for l = 1 : n   %  layer
  if strcmp(layers{l}.type, 'i') % input
    if (isfield(layers{l}, 'mean'))
```

```
      curlen = length(layers{l}.mw(:));
      %w_trans = permute(layers{l}.mw, [2 1 3 4]);
      %w_trans(:) = weights(ind+1:ind+curlen);
      %layers{l}.mw = permute(w_trans, [2 1 3 4]);
      layers{l}.mw(:) = weights(ind+1:ind+curlen);
      ind = ind + curlen;
    end;
    if (isfield(layers{l}, 'maxdev'))
      curlen = length(layers{l}.sw(:));
      %w_trans = permute(layers{l}.sw, [2 1 3 4]);
      %w_trans(:) = weights(ind+1:ind+curlen);
      %layers{l}.sw = permute(w_trans, [2 1 3 4]);
      layers{l}.sw(:) = weights(ind+1:ind+curlen);
      ind = ind + curlen;
    end;
  elseif strcmp(layers{l}.type, 'n') % normalization
    w_trans = permute(layers{l}.w, [2 1 3 4]);
    curlen = length(w_trans(:));
    w_trans(:) = weights(ind+1:ind+curlen);
    layers{l}.w = permute(w_trans, [2 1 3 4]);
    ind = ind + curlen;
  elseif strcmp(layers{l}.type, 'c') % convolutional
    curlen = length(layers{l}.k(:));
    k_trans = permute(layers{l}.k, [4 1 2 3]);
    k_trans(:) = weights(ind+1:ind+curlen);
    layers{l}.k = permute(k_trans, [2 3 4 1]);
    %layers{l}.k(:) = weights(ind+1:ind+curlen);
    ind = ind + curlen;
    curlen = layers{l}.outputmaps;
    layers{l}.b(:) = weights(ind+1:ind+curlen);
    ind = ind + curlen;
  elseif strcmp(layers{l}.type, 'f') % fully connected
    curlen = numel(layers{l}.w);
    %w_trans = layers{l}.w';
    %w_trans(:) = weights(ind+1:ind+curlen);
    %layers{l}.w = w_trans';
    layers{l}.w(:) = weights(ind+1:ind+curlen);
    ind = ind + curlen;
    curlen = numel(layers{l}.b);
    layers{l}.b(:) = weights(ind+1:ind+curlen);
    ind = ind + curlen;
  end;
end
assert(ind == size(weights, 1), 'Weights vector is too long!');

end
```

gg. shrink.m

```
function A = shrink(A, sc, st)

for d = 1 : length(size(A))
  ind = [];
  for i = 1 : sc(d) - st(d)
    ind = [ind; sc(d)+i : sc(d) : size(A, d)];
  end;
```

```
    newind = ind-sc(d)+st(d);
    if (d == 1)
      A(newind, :, :, :) = A(newind, :, :, :) + A(ind, :, :, :);
      A(ind, :, :, :) = [];
    elseif (d == 2)
      A(:, newind, :, :) = A(:, newind, :, :) + A(:, ind, :, :);
      A(:, ind, :, :) = [];
    elseif (d == 3)
      A(:, :, newind, :) = A(:, :, newind, :) + A(:, :, ind, :);
      A(:, :, ind, :) = [];
    elseif (d == 4)
      A(:, :, :, newind) = A(:, :, :, newind) + A(:, :, :, ind);
      A(:, :, :, ind) = [];
    end;
  end;

  end
```

hh. sigm.m

```
function X = sigm(P)
  X = (1 + tanh(P/2)) / 2;
  %X = 1./(1+exp(-P));
end
```

ii. soft.m

```
function a = soft(a)
a = exp(bsxfun(@minus, a, logsumexp(a, 2)));
%a = exp(a) ./ repmat(sum(exp(a), 2), 1, size(a, 2));
end
```

jj. softder.m

```
function d = softder(d, a)

%comsum = sum(a, 2);
%d = d .* bsxfun(@times, a, 1 - comsum);
comsum = sum(a .* d, 2);
d = a .* bsxfun(@minus, d, comsum);

end
```

kk. stretch.m

```
function B = stretch(A, sc, st)
asize = [size(A, 1) size(A, 2) size(A, 3) size(A, 4)];
dim = length(asize);
s = ceil(asize ./ st);
```

```
newsize = sc .* s;
B = zeros(newsize);
order = cell(dim, 1);
ind = cell(dim, 1);
oldind = cell(dim, 1);
for d = 1 : dim
  ind{d} = [];
  for i = 1 : sc(d) - st(d)
    ind{d} = [ind{d}; sc(d)+i : sc(d) : newsize(d)];
    oldind{d} = ind{d} - sc(d) + st(d);
  end;
  order{d} = 1 : newsize(d);
  order{d}(ind{d}) = [];
  order{d}(size(A, d)+1:end) = [];
end;
B(order{:}) = A;

for d = 1 : dim
  if (d == 1)
    B(ind{d}, :, :, :) = B(oldind{d}, :, :, :);
  elseif (d == 2)
    B(:, ind{d}, :, :) = B(:, oldind{d}, :, :);
  elseif (d == 3)
    B(:, :, ind{d}, :) = B(:, :, oldind{d}, :);
  elseif (d == 4)
    B(:, :, :, ind{d}) = B(:, :, :, oldind{d});
  end;
end;

end
```

ll. uniq.m

```
function mm = uniq(mm, sc)

shift = prod(sc);
shiftmat = reshape(shift-1 : -1 : 0, [sc(2) sc(1)])' / shift;
mmsize = [size(mm, 1) size(mm, 2) size(mm, 3) size(mm, 4)];
coefs = mmsize ./ sc;
shiftmat = repmat(shiftmat, coefs);
mm = mm + shiftmat;

fi = ceil((sc+1)/2);
b = strel('rectangle', [sc(1) sc(2)]);
z = imdilate(mm, b);
maxval = z(fi(1):sc(1):end, fi(2):sc(2):end, :, :);
maxval = expand(maxval, sc);
mm = (mm == maxval);

end
```

mm. updateweights.m

```
function layers = updateweights(layers, params, epoch, regime)
if (length(params.momentum) == 1)
  momentum = params.momentum;
else
  momentum = params.momentum(epoch);
end;
if (length(params.alpha) == 1)
  alpha = params.alpha;
else
  alpha = params.alpha(epoch);
end;
if (length(params.beta) == 1)
  beta = params.beta;
else
  beta = params.beta(epoch);
end;

for l = 1 : numel(layers)

  if strcmp(layers{l}.type, 'n') || strcmp(layers{l}.type, 'f')
    if (regime == 0)
      dw = momentum * layers{l}.dwp;
    else
      dw = alpha * layers{l}.dw + beta * layers{l}.dw2;
      layers{l}.dw2(:) = 0;
      signs = layers{l}.dw .* layers{l}.dwp;
      layers{l}.gw(signs > 0) = layers{l}.gw(signs > 0) +
params.adjustrate;
      layers{l}.gw(signs <= 0) = layers{l}.gw(signs <= 0) * (1 -
params.adjustrate);
      layers{l}.gw(layers{l}.gw > params.maxcoef) = params.maxcoef;
      layers{l}.gw(layers{l}.gw <= params.mincoef) =
1/params.mincoef;
      dw = dw .* layers{l}.gw;
      layers{l}.dwp = dw;
      dw = (1 - momentum) * dw;
    end;
    layers{l}.w = layers{l}.w - dw;
    layers{l}.w(-layers{l}.eps < layers{l}.w & layers{l}.w <
layers{l}.eps) = 0;

  elseif strcmp(layers{l}.type, 'c')
    if (regime == 0)
      dk = momentum * layers{l}.dkp;
    else
      dk = alpha * layers{l}.dk + beta * layers{l}.dk2;
      layers{l}.dk2(:) = 0;
      signs = dk .* layers{l}.dkp;
      layers{l}.gk(signs > 0) = layers{l}.gk(signs > 0) +
params.adjustrate;
      layers{l}.gk(signs <= 0) = layers{l}.gk(signs <= 0) * (1 -
params.adjustrate);
      layers{l}.gk(layers{l}.gk > params.maxcoef) = params.maxcoef;
      layers{l}.gk(layers{l}.gk <= params.mincoef) = params.mincoef;
      dk = dk .* layers{l}.gk;
```

```
      layers{l}.dkp = dk;
      dk = (1 - momentum) * dk;
    end;
    layers{l}.k = layers{l}.k - dk;
    layers{l}.k(-layers{l}.eps < layers{l}.k & layers{l}.k <
layers{l}.eps) = 0;
  end;

  % for all transforming layers
  if strcmp(layers{l}.type, 'c') || strcmp(layers{l}.type, 'f')
    if (regime == 0)
      db = momentum * layers{l}.dbp;
    else
      db = alpha * layers{l}.db + beta * layers{l}.db2;
      layers{l}.db2(:) = 0;
      signs = db .* layers{l}.dbp;
      layers{l}.gb(signs > 0) = layers{l}.gb(signs > 0) +
params.adjustrate;
      layers{l}.gb(signs <= 0) = layers{l}.gb(signs <= 0) * (1 -
params.adjustrate);
      layers{l}.gb(layers{l}.gb > params.maxcoef) = params.maxcoef;
      layers{l}.gb(layers{l}.gb <= params.mincoef) =
1/params.mincoef;
      db = db .* layers{l}.gb;
      layers{l}.dbp = db;
      db = (1 - momentum) * db;
    end;
    layers{l}.b = layers{l}.b - db;
    layers{l}.b(-layers{l}.eps < layers{l}.b & layers{l}.b <
layers{l}.eps) = 0;
  end;
end

end
```

nn. maxscale.cpp

```
#include <mex.h>
#include <algorithm>

#define NARGIN 2
#define IN_I pRhs[0] // image
#define IN_S pRhs[1] // scale

#define NARGOUT 1
#define OUT_F   pLhs[0] // filtered

inline void mexAssert(bool b, char *msg) {
  if (!b) {
        mexErrMsgTxt(msg);
    }
}

#include <string>
inline void mexPrintMsg(std::string msg, long double x) {
```

```
  mexPrintf((msg + ": " + std::to_string(x) + "\n").c_str());
  mexEvalString("drawnow;");
}

void mexFunction(int nLhs, mxArray* pLhs[], int nRhs, const mxArray*
pRhs[]) {

  mexAssert(nRhs == NARGIN, "Number of input arguments in wrong!");
  mexAssert(nLhs == NARGOUT, "Number of output arguments is wrong!"
);

  mexAssert(IN_I != NULL, "Image array is NULL");
  mexAssert(mxIsNumeric(IN_I), "Image is not numeric!" );
  int imdimnum = (int) mxGetNumberOfDimensions(IN_I);
  mexAssert(imdimnum == 2 || imdimnum == 3, "The image must have 2
or 3 dimensions");
  int imdim[2];
  const mwSize *pdim = mxGetDimensions(IN_I);
  imdim[0] = (int) pdim[0];
  imdim[1] = (int) pdim[1];
  int imnum = 1;
  if (imdimnum == 3) {
    imnum = pdim[2];
  }
  double *pim = mxGetPr(IN_I);

  mexAssert(IN_S != NULL, "Image array is NULL");
  mexAssert(mxIsNumeric(IN_S), "Scale is not numeric!" );
  int scdimnum = (int) mxGetNumberOfDimensions(IN_S);
  mexAssert(scdimnum == 2, "The scale parameter has more than 2
dimensions");
  pdim = mxGetDimensions(IN_S);
  mexAssert((int) pdim[0] == 1 || (int) pdim[1] == 1, "The scale
parameter must be a vector");
  double *psc = mxGetPr(IN_S);

  int scale[2];
  scale[0] = (int) psc[0];
  scale[1] = (int) psc[1];

  mwSize scdim[3];
  scdim[0] = ceil((double) imdim[0] / scale[0]);
  scdim[1] = ceil((double) imdim[1] / scale[1]);
  scdim[2] = imnum;
  const mwSize *mwdimc = scdim;

  mxArray *mx_filtered = mxCreateNumericArray(imdimnum, mwdimc,
mxDOUBLE_CLASS, mxREAL);
  double *pfilt = mxGetPr(mx_filtered);
  for (int k = 0; k < scdim[2]; ++k) {
    for (int i = 0; i < scdim[0]; ++i) {
      for (int j = 0; j < scdim[1]; ++j) {
        int maxu = std::min(scale[0], (int) imdim[0] - i*scale[0]);
        int maxv = std::min(scale[1], (int) imdim[1] - j*scale[1]);
        int filtind = k*scdim[0]*scdim[1] + j*scdim[0] + i;
        int imind = k*imdim[0]*imdim[1] + j*scale[1]*imdim[0] +
i*scale[0];
```

```
        pfilt[filtind] = pim[imind];
        for (int u = 0; u < maxu; ++u) {
          for (int v = 0; v < maxv; ++v) {
            if (pfilt[filtind] < pim[imind + v*imdim[0] + u]) {
              pfilt[filtind] = pim[imind + v*imdim[0] + u];
            }
          }
        }
      }
    }
  }
  OUT_F = mx_filtered;
}
```