2154 INFSCI 2410 1250 INTRO TO NEURAL NETWORKS

Leon Lai <Leon.Lai@pitt.edu>

Project Report

2016-04-08

In this project, 123 autoassociative neural networks were created, trained, and tested for specific classes of grayscale images, the results were observed and analyzed, and the best neural network model was chosen for each class of images. Section “Background” describes the technique used, including a summary of autoassociative neural networks, the method of training, and the simulation environment. Section “Method Description” reports the model used, including the network creation and training parameters as well as the source code for network creation, training, and testing. Section “Problem Description” states the tasks being learned and computed by the network, including the grayscale image data sets, expected results, and the source code that generated the data sets. The results of learning and computation are presented in section “Results”. This project report concludes with a discussion of the results, including whether the network performed as expected and the lessons learned.

Background

As its name suggests, an autoassociative neural network is a neural network that associates some data with itself; in other words, it tries to learn the identity function within a given data domain. Fig. 1 illustrates the structure of an autoassociative neural network: it is a multilayer feedforward neural network, necessarily has as many input layer neurons as output layer neurons, and moreover has an intermediate layer, the “bottleneck” layer, with very few neurons compared to the input layer [1]. When stimulated with an entry from the data set the network was trained for, the bottleneck layer activity is a compressed representation of that entry, and such representations are used for dimensionality reduction and feature abstraction. Parameters of an autoassociative neural network are minimally m, the number of input layer neurons, and f, the number of bottleneck layer neurons. In the five-layer version shown in Fig. 1, the bottleneck layer is the second intermediate layer, the first and third intermediate layers are known as “mapping” and “demapping” layers, respectively, and the network additionally has parameters M_1 and M_2, the number of neurons in the mapping and demapping layers, respectively [1]. Kramer suggested the following inequality for choosing M_1 and M_2, where n is the number of training set cases [2].

$$M_1 + M_2 \ll m \frac{n-f}{m+f+1}$$

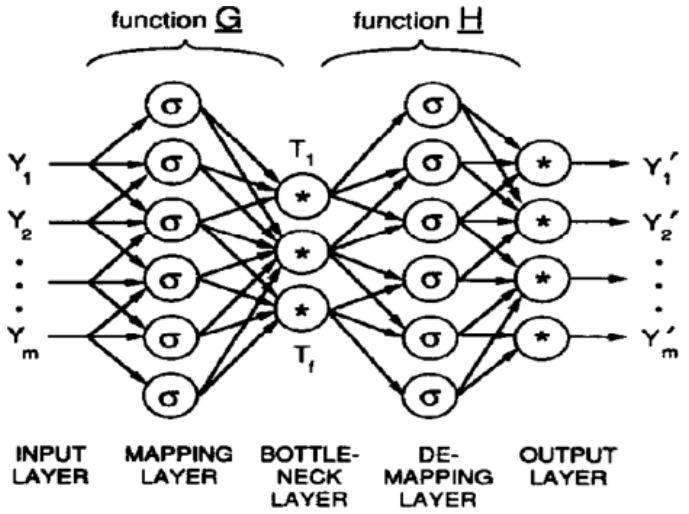

Fig. 1. Architecture of the autoassociative network [1].

Like other feedforward neural networks, autoassociative neural networks commonly learn via backpropagation training. Backpropagation minimizes a "cost" function that quantifies the difference between actual network output vector and desired output vector [3]. It does so by iteratively adjusting weights and biases by gradient descent against an activity, as follows [3], where ε, E, y, z, and w_{ji} respectively denote learning rate, cost function, neuron output activity, neuron pre-squash activity, and weight of neuron i output going into neuron j:

$$\Delta w_{ji} = -\varepsilon \cdot \partial E/\partial y_j \cdot \partial y_j/\partial z_j \cdot \partial z_j/\partial w_{ji} = -\varepsilon \cdot \partial E/\partial y_j \cdot \partial y_j/\partial z_j \cdot y_i$$
$$\Delta b_j = -\varepsilon \cdot \partial E/\partial y_j \cdot \partial y_j/\partial z_j \cdot \partial z_j/\partial b_j = -\varepsilon \cdot \partial E/\partial y_j \cdot \partial y_j/\partial z_j$$

While solutions for these equations should be immediately transparent for the output layer, for intermediate layers, they need perception of the following equality, which holds for every intermediate layer neuron i [3]:

$$\partial E/\partial y_i = \sum_j \partial E/\partial z_j \cdot \partial z_j/\partial y_i = \sum_j \partial E/\partial z_j \cdot w_{ji}$$

Adjustments quantities are calculated first for the output layer, then for the penultimate layer, and so on, lest the value of $\partial E/\partial z_j$ remain unknown to the neuron i. After all adjustment quantities are calculated, adjustments are made.

Online learning is a commonly employed learning technique. It is an iteration. During each episode, one entry is randomly chosen from the training set, the network is stimulated with it, and then one iteration of backpropagation is performed.

Neural networks are commonly simulated purely with software. A popular software package for this purpose is MATLAB, a matrix-based numerical computing environment. The version of MATLAB available to this project was R2015a.

Method Description

Forty autoassociative neural network models were used. Their network parameters are shown in the following table. Models with five layers have mapping layers; those with three layers do not. M_1 and M_2 were fixed as f plus one to reduce the model space while satisfying the formula for choosing them as discussed in Background.

	Models 1…20	**Models 21…40**
Number of layers	3	5
m	100	100
f	1…20	1…20
M_1	Not applicable	f + 1
M_2	Not applicable	f + 1

Output layer neurons used sigmoid function as activation function. Intermediate layer neurons used twice-as-wide hyperbolic tangent function as activation function, as follows, where j indexes an intermediate layer neuron:

$$y_j = (1 - e^{-z_j})/(1 + e^{-z_j})$$

Networks were initialized as follows: each weight and bias was set to a uniformly distributed random number in the interval (-0.5, 0.5). Networks were trained via online learning with backpropagation, which are discussed in Background. The learning rate was 0.05. Maximum number of online learning iterations was 500000. A second criterion for conclusion of learning was reaching a target cost function value, which is discussed in Problem Description. The cost function used was the half-sum-square function, as follows, where d_j denotes desired output activity for output layer neuron j:

$$E = \tfrac{1}{2}\sum_j (y_j - d_j)^2$$

Given desired output vector, every Δw and Δb can now be calculated using backpropagation equations discussed in Background.

Below are the annotated source code of the three MATLAB functions that perform network creation, training, and testing. “lffnn” instantiates and initializes a layered feedforward neural network. “passforward” does a feed-forward pass to compute activations at all hidden layers and then at the output layer. “passbackward” performs backpropagation for network training. These functions are called by the program whose source code can be found in Appendix A: Project.m.

lffnn.m

```
% 2015-04-29 Leon Lai <Leon.Lai@pitt.edu>
%
% This function generates a layered feed-forward neural network with a given
% number of layers, a given number of neurons for each layer, and random
% weights and bias within a given peak amplitude for each neuron not of the
% input layer.
%
% Parameters:
%   layer_spec:
%     array whose length is the number of layers that the network shall have
%     and whose (k)th element is the number of neurons that the (k)th layer of
%     lffnn shall have, where the first layer is the input layer.
%   rand_amp:
%     peak-to-peak amplitude of the randomly generated weights and biases.
%   hidden_layers_are_sigpn:
%     true if the twice-as-wide hyperbolic tangent function should be used for
%     hidden layer neurons, false if the sigmoid function should be used.
%
% Returns:
%   lffnn:
%     structure array whose fields 'weights' and 'biases' contain cell arrays;
%     the (L)th cell of 'weights' contains a matrix whose (j)th row's (k)th
%     column is the weight for the connection into the (L+1)th layer's (j)th
%     neuron from the (L)th layer's (k)th neuron; the (L)th cell of 'biases'
%     contains a column vector whose (j)th row is the bias of the (L+1)th
%     layer's (j)th neuron; the first layer is the input layer, whose neurons
%     have zero as biases and a single one as weights.
%
function lffnn = lffnn (layer_spec, rand_amp, hidden_layers_are_sigpn)
for L = 1 : length(layer_spec) - 1
  lffnn.weights{L} = rand_amp * (rand(layer_spec(L+1), layer_spec(L)) - 0.5);
  lffnn.biases {L} = rand_amp * (rand(layer_spec(L+1), 1            ) - 0.5);
end
lffnn.hidden_layers_are_sigpn = hidden_layers_are_sigpn;
```

passforward.m

```
% 2016-04-10 Leon Lai <Leon.Lai@pitt.edu>
%
% This function stimulates a layered feed-forward neural network with an input
% vector and captures the output values of each layer.
%
% Parameters:
%   lffnn:
%     layered feed-forward neural network like that returned by lffnn.
%   X:
%     matrix whose columns are input value vectors to stimulate lffnn with.
%
% Returns:
%   activity:
%     cell array whose (L)th cell contains a matrix whose (j)th row's (i)th
```

```
%     column is the output value of lffnn's (L)th layer's (j)th neuron when
%     stimulating lffnn with the (i)th input value vector, where the first
%     layer is the input layer. Note that lffnn_state{1} and lffnn_state{end}
%     are thus respectively the network's input and output.
%
function activity = passforward (lffnn, X)
% Note: activity{L} -> lffnn{L} -> activity{L+1}
L = 0 ;
activity{L+1} = X ;
%%
% Calculate activity for each intermediate layer.
for L = 1 : length(lffnn.weights) - 1 % is empty if length < 1
  % zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
  z=lffnn.biases{L}*ones(1,size(activity{L},2))+lffnn.weights{L}*activity{L};
  if lffnn.hidden_layers_are_sigpn
    % y = sigpn(z)
    activity{L+1} = (1 - exp(-z)) ./ (1 + exp(-z)) ;
  else
    % y = sig01(z)
    activity{L+1} =  1            ./ (1 + exp(-z)) ;
  end
end
%%
% Calculate activity for output layer.
L = length(lffnn.weights);
% zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
z = lffnn.biases{L}*ones(1,size(activity{L},2))+lffnn.weights{L}*activity{L};
% y = sig01(z)
activity{L+1} = 1 ./ (1 + exp(-z)) ;
```

passbackward.m

```
% 2016-04-10 Leon Lai <Leon.Lai@pitt.edu>
%
% This function propagates back errors along an activated layered feed-forward
% neural network assuming online learning (stochastically choosing one case
% per epoch) and half-of-sum-of-square cost function.
%
% Parameters:
%   lffnn:
%     layered feed-forward neural network like that returned by lffnn.
%   activity:
%     network activity like that returned by passforward.
%   Y:
%     matrix whose columns are desired output value vectors.
%   dx:
%     learning rate, denoted by ε in Rumelhart et al. (1986).
%
% Returns:
%   lffnn:
%     same as parameter lffnn but with weights and biases updated by
%     backpropagation.
%
```

```
function lffnn = passbackward (lffnn, activity, Y, dx)
% Note: activity{L} -> lffnn{L} -> activity{L+1}
%%
% Calculate deltas for output layer.
L = length(lffnn.weights) ;
% ∂E/∂yⱼ where E = ½∑ⱼ(yⱼ-dⱼ)² assuming online learning
dE_dy = activity{L+1} - Y ;
% ∂E/∂zⱼ  = ∂E/∂yⱼ · ∂yⱼ/∂zⱼ  where yⱼ = sig01(zⱼ)
dE_dz = dE_dy .* (activity{L+1}) .* (1 - activity{L+1}) ;
% ∂E/∂wⱼᵢ = ∂E/∂zⱼ · ∂zⱼ/∂wⱼᵢ where zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
dE_dw = dE_dz * activity{L}' ;
% ∂E/∂bⱼ  = ∂E/∂zⱼ · ∂zⱼ/∂bⱼ  where zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
dE_db = dE_dz * 1 ;
% Δwⱼᵢ = -ε∂E/∂wⱼᵢ
delta_weights{L} = -dx * dE_dw ;
% Δbⱼ  = -ε∂E/∂bⱼ
delta_biases {L} = -dx * dE_db ;
%%
% Calculate deltas for each intermediate layer.
for L = L-1 : -1 : 1
  % ∂E/∂yᵢ = ∑ⱼ ∂E/∂zⱼ·∂zⱼ/∂yᵢ = ∑ⱼ ∂E/∂zⱼ·wⱼᵢ
  dE_dy = (dE_dz' * lffnn.weights{L+1})' ;
  % ∂E/∂zⱼ  = ∂E/∂yⱼ · ∂yⱼ/∂zⱼ
  if lffnn.hidden_layers_are_sigpn
    % yⱼ = sigpn(zⱼ)
    dE_dz = dE_dy .* (1 + activity{L+1}) .* (1 - activity{L+1}) .* 0.5 ;
  else
    % yⱼ = sig01(zⱼ)
    dE_dz = dE_dy .*      activity{L+1}  .* (1 - activity{L+1})        ;
  end
  % ∂E/∂wⱼᵢ = ∂E/∂zⱼ · ∂zⱼ/∂wⱼᵢ where zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
  dE_dw = dE_dz * activity{L}' ;
  % ∂E/∂bⱼ  = ∂E/∂zⱼ · ∂zⱼ/∂bⱼ  where zⱼ = bⱼ + ∑ᵢ wⱼᵢ·yᵢ
  dE_db = dE_dz * 1 ;
  % Δwⱼᵢ = -ε∂E/∂wⱼᵢ
  delta_weights{L} = -dx * dE_dw ;
  % Δbⱼ  = -ε∂E/∂bⱼ
  delta_biases {L} = -dx * dE_db ;
end
%%
% Adjust each layer's weights and biases.
for L = length(lffnn.weights) : -1 : 1
  lffnn.weights{L} = lffnn.weights{L} + delta_weights{L} ;
  lffnn.biases {L} = lffnn.biases {L} + delta_biases {L} ;
end
```

Problem Description

Nine sets of 10×10 8-bit grayscale images were created. They are verbally described in the following list and visually described in Fig. 2.

> "Horizontal line": 8 images with a 3-pixel wide white horizontal line spanning the image; each image has it at a unique position—all possible positions are present in this set.
> "Square": 64 black images with a 3-pixel wide white square; each image has it at a unique position—all possible positions are present in this set.
> "Triangle": 64 black images with a 3-pixel wide white isosceles triangle pointing northwest; each image has it at a unique position—all possible positions are present in this set.
> "Noisy horizontal line": horizontal line set with 25% salt-and-pepper noise.
> "Noisy square": square set with 25% salt-and-pepper noise.
> "Noisy triangle": triangle set with 25% salt-and-pepper noise.
> "Zeros": one entirely black image.
> "Ones": one entirely white image.
> "Random": one image that is uniformly distributed random noise.

For each of the forty network models, one network was created and trained to autoassociate the horizontal line, square, and triangle sets, respectively. Consequently, 120 autoassociative neural networks were created and trained. Value of pixel at column i and row j of a training image was fed into the input layer neuron indexed by 10(i-1)+j. Network structure, initialization, and learning configurations are discussed in Method Description. The second criterion for conclusion of learning was that the network could reproduce without error all entries of the training set as far as 8-bit grayscale images are concerned; in other words, that the maximum difference, that is, the maximum among the absolute values of differences between desired and actual values for all output neurons' output activities for all training set entries, was less than 1/256.

For each training set, the trained network that concluded learning due to the second criterion with the least layers and the lowest f was chosen as the best autoassociative network for that set; if such network does not exist, by existence of mapping layers, the lowest f whose network's maximum difference saw a sudden decrease compared to the previous f and did not vary much for the remaining f's, or the last f if such an f value could not be found, was chosen and its associated network was chosen as the best network. If such a network exists for both with and without mapping layers, the one without mapping layers was chosen.

Next, three networks were trained: best network models were cloned, reinitialized, and retrained; outputs were grabbed for eight iterations logarithmically equally spaced apart from before first to after last iteration to observe learning progressions.

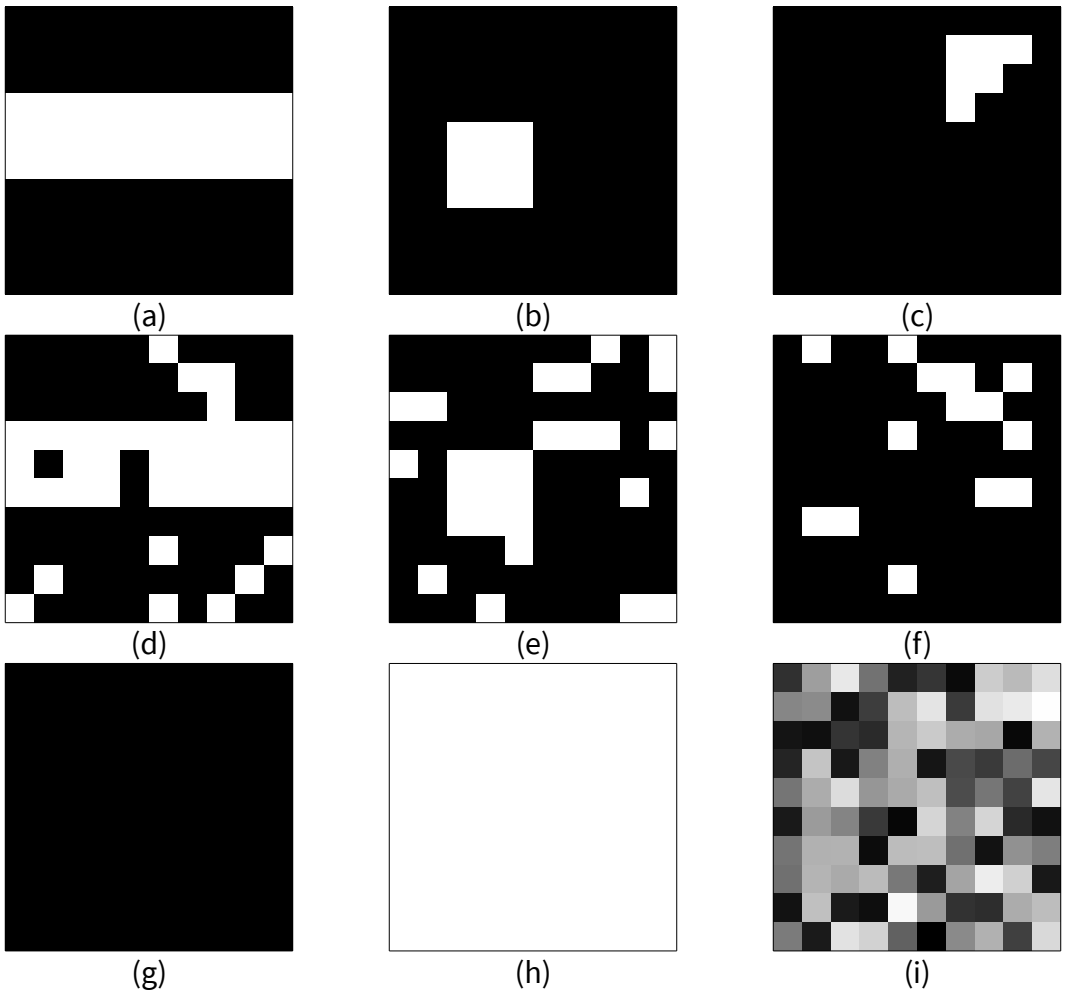

Fig. 2. Rendered image sets. (a)(b)(c): exemplars for horizontal line, square, and triangle respectively. (d)(e)(f): exemplars for noisy horizontal line, noisy square, and noisy triangle whose features' positions correspond to (a), (b), and (c), respectively. (g): zeros. (h): ones. (i): random.

Finally, to observe the effect of noise in input data on the best trained networks, the best networks were stimulated with the set that is the noisy counterpart to what they were trained on. Moreover, each best network was stimulated with the zeros, ones, and random sets just to see what the output would look like.

Expected results are that the horizontal line set would require only f=1 with no mapping layers due to the one-dimensional nature of the feature that distinguishes its members, the vertical position of the horizontal line. Likewise, the other two sets were expected to require f=2 with no mapping layers.

Below are the annotated source code of the three MATLAB functions that generated the image sets. These functions are called by the program whose source code can be found in Appendix A: Project.m.

all_position_horizontal_line.m

```
% 2015-04-29 Leon Lai <Leon.Lai@pitt.edu>
%
% This function generates all unique images containing a horizontal line of
% specified stroke width and color. Warning: this function does not verify
% that the specified stroke width remains under the minimum canvas dimension.
%
% Parameters:
%   m:
%     height of image.
%   n:
%     width of image.
%   width:
%     stroke width.
%   color:
%     value between 0 and 1 where 0 is black and 1 is white.
%   colmode:
%     if true, instead of returning a cell array, returns a matrix where each
%     column is an image after reshape.
%
% Returns:
%   C:
%     cell array or matrix of image data.
%
function C = all_position_horizontal_line (m, n, width, color, colmode)
Y = m - (width - 1) ;
C = cell (1, Y) ;
k = 1 ;
for y = 1 : Y
  C {k} = zeros (m, n) ;
  for j = y : y + (width - 1)
    C {k} (j, :) = color;
  end
  k = k + 1 ;
end
if colmode
  C = cell2mat (cellfun (@ (x) reshape (x, m * n, 1), C, 'un', 0)) ;
end
```

all_position_square.m

```
% 2015-04-29 Leon Lai <Leon.Lai@pitt.edu>
%
% This function generates all unique images containing a square of specified
% width and color. Warning: this function does not verify that the specified
% width remains under the minimum canvas dimension.
%
% Parameters:
%   m:
%     height of image.
%   n:
%     width of image.
```

```
%   width:
%     width of square.
%   color:
%     value between 0 and 1 where 0 is black and 1 is white.
%   colmode:
%     if true, instead of returning a cell array, returns a matrix where each
%     column is an image after reshape.
%
% Returns:
%   C:
%     cell array or matrix of image data.
%
function C = all_position_square (m, n, width, color, colmode)
Y = m - (width - 1) ;
X = n - (width - 1) ;
C = cell (1, Y * X) ;
k = 1 ;
for y = 1 : Y
  for x = 1 : X
    C {k} = zeros (m, n) ;
    for j = y : y + (width - 1)
      for i = x : x + (width - 1)
        C {k} (j, i) = color;
      end
    end
    k = k + 1 ;
  end
end
if colmode
  C = cell2mat (cellfun (@ (x) reshape (x, m * n, 1), C, 'un', 0)) ;
end
```

all_position_triangle.m

```
% 2015-04-29 Leon Lai <Leon.Lai@pitt.edu>
%
% This function generates all unique images containing an isosceles triangle
% of specified width and color. Warning: this function does not verify that
% the specified width remains under the minimum canvas dimension.
%
% Parameters:
%   m:
%     height of image.
%   n:
%     width of image.
%   width:
%     width of triangle.
%   color:
%     value between 0 and 1 where 0 is black and 1 is white.
%   colmode:
%     if true, instead of returning a cell array, returns a matrix where each
%     column is an image after reshape.
%
```

```
% Returns:
%   C:
%     cell array or matrix of image data.
%
function C = all_position_triangle (m, n, width, color, colmode)
Y = m - (width - 1) ;
X = n - (width - 1) ;
C = cell (1, Y * X) ;
k = 1 ;
for y = 1 : Y
  for x = 1 : X
    C {k} = zeros (m, n) ;
    for j = y : y + (width - 1)
      for i = x : x + (width - 1) - (j - y)
        C {k} (j, i) = color;
      end
    end
    k = k + 1 ;
  end
end
if colmode
  C = cell2mat (cellfun (@ (x) reshape (x, m * n, 1), C, 'un', 0)) ;
end
```

Results

No network concluded training due to the second criterion. In other words, no network was able to reproduce all of its training set exactly as far as 8-bit grayscale images are concerned. Training outcomes, namely maximum and mean differences between inputs and outputs, are plotted in Fig. 3. The best network for each training set is decided and tabulated as follows.

	Horizontal line	**Square**	**Triangle**
f	3	5	7
Mapping layers?	No	No	No

Some of the best networks' actual outputs are displayed in Fig. 4. Learning progression snapshots for one of the best networks are displayed in Fig. 5. Outcomes of testing with the noisy, zeros, ones, and random sets are plotted in Fig. 6. Some actual outputs of testing with the aforementioned sets are displayed in Fig. 7 and Fig. 8.

As seen in Fig. 3, it was relatively clear which network model was best for the horizontal line image set. For the square set, it was much less clear due to the gradualness and unexpected spikes in the curve of maximum difference with respect to f. These observations apply to the triangle set as well. Nevertheless, from Fig. 4, actual outputs are largely the same as inputs to the human eye.

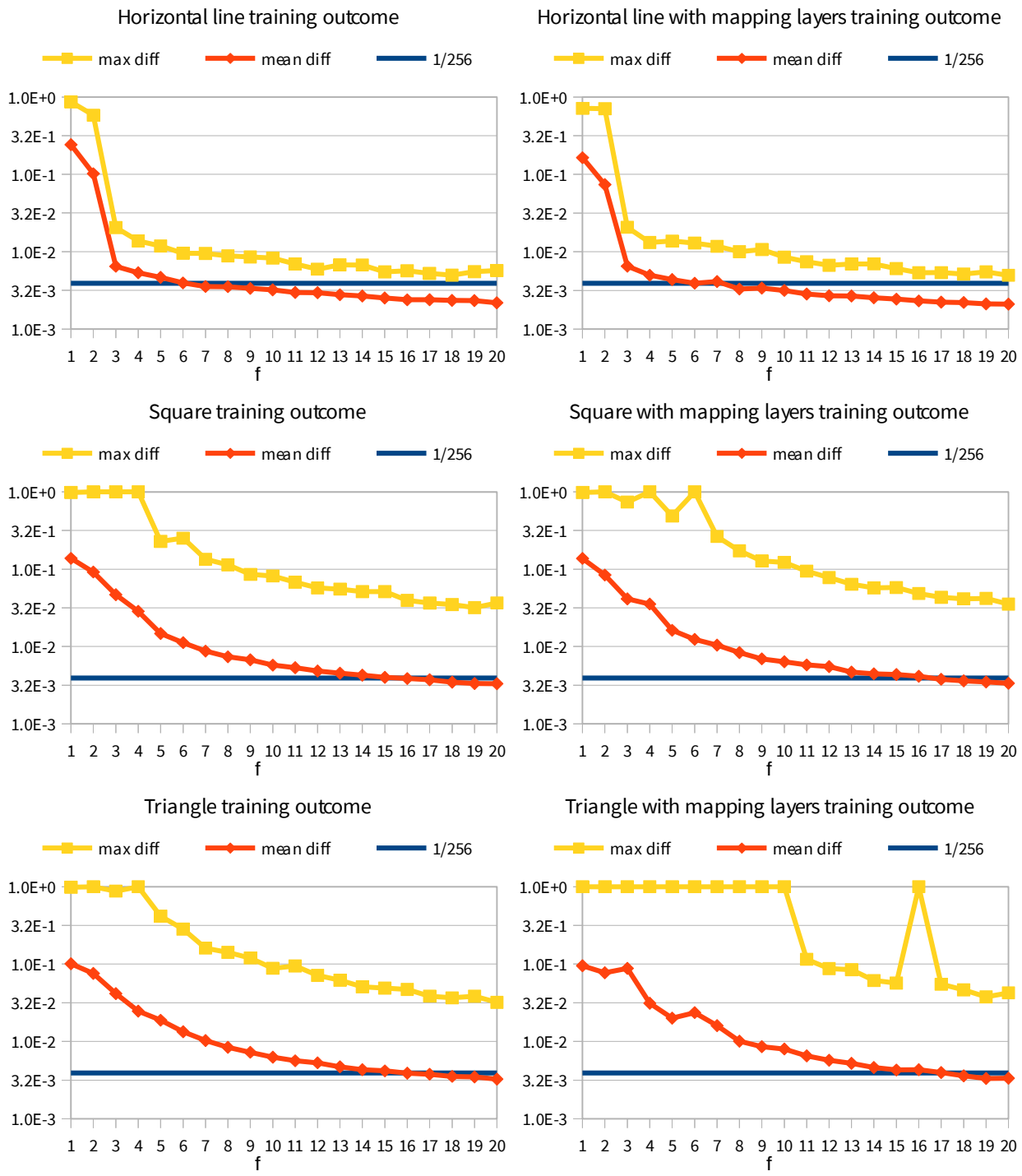

Fig. 3. Network training outcomes, grouped by training set and existence of mapping layers.

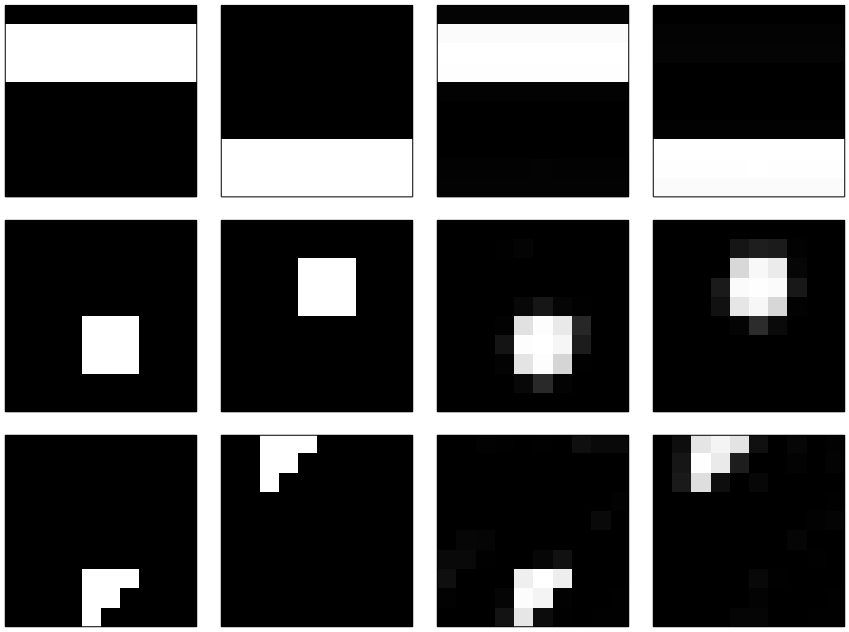

Fig. 4. Some actual input-output pairs of best networks. Left half: input; right half: output. Top: horizontal line; center: square; bottom: triangle.

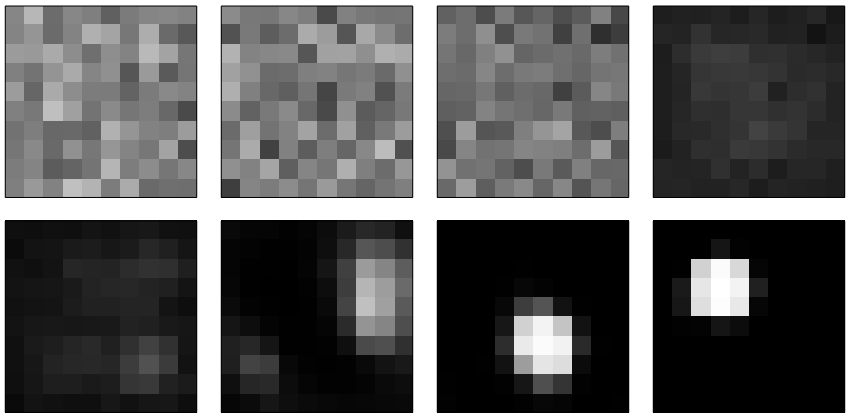

Fig. 5. Output of network for square, f=5, no mapping layers, after initialization, then after 7, 43, 277, 1806, 11768, 76707, and 500000 training iterations.

At first glance of Fig. 6, it would seem that the best networks, with mean differences for noisy sets all being roughly within 0.1 to 0.2, are similarly capable at removing salt-and-pepper noise, with horizontal line being a little better. As seen in Fig. 7, however, it is not the case: while all removed noise well, horizontal line clearly retained the feature, square distorted the feature to the point of being unrecognizable, and triangle removed the feature as well. This discrepancy could be explained as follows: 0.2 ought to be seen as twice as much as and not similar to 0.1; but if 0.2 and 0.1 could be seen as similar, inferring similar noise reduction capabilities from similar mean differences assumes the features occupy the same proportions of the images, but horizontal lines occupy eight times as much of the image as do squares and even more than do triangles.

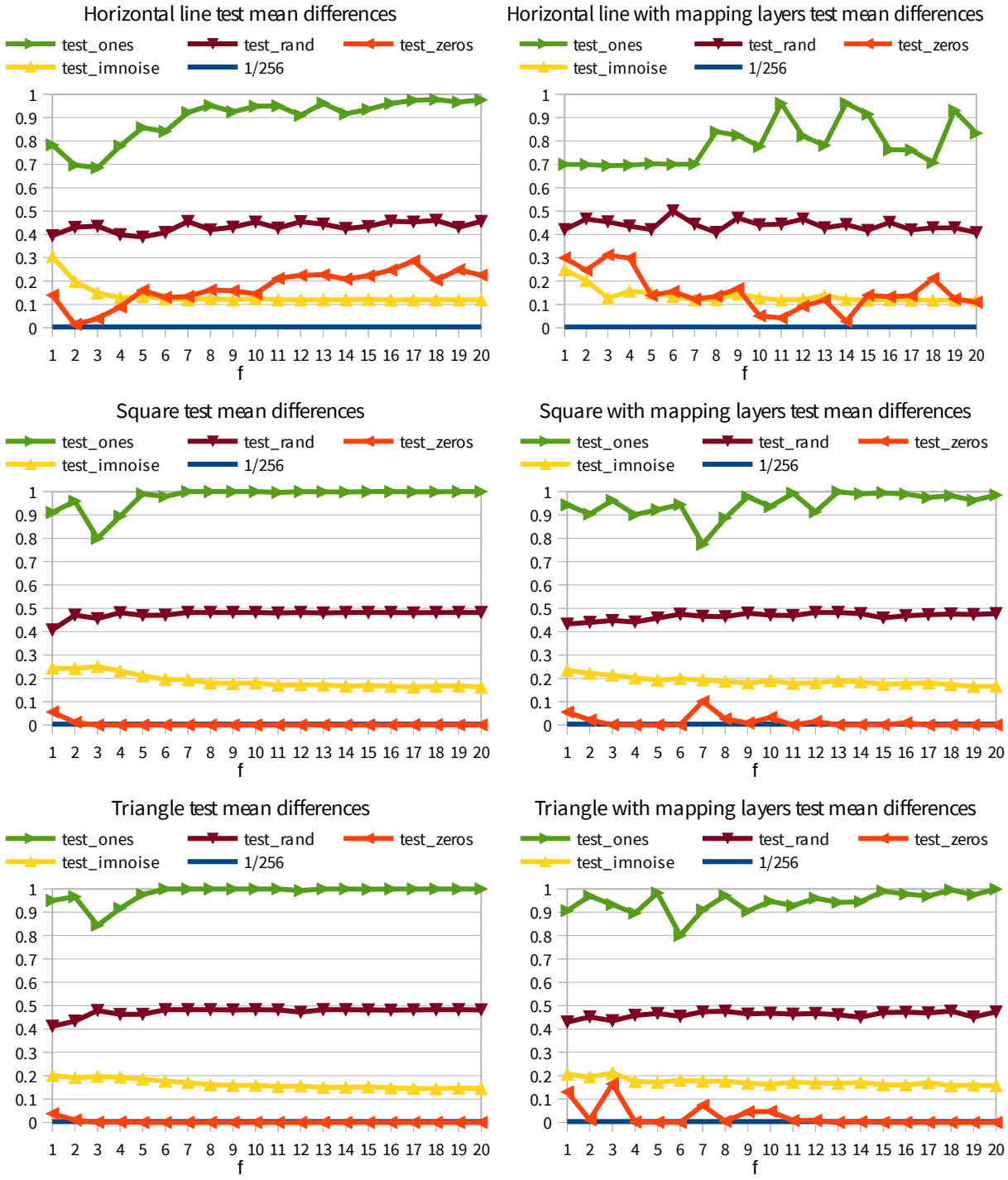

Fig. 6. Network testing outcomes, grouped by training set and existence of mapping layers.

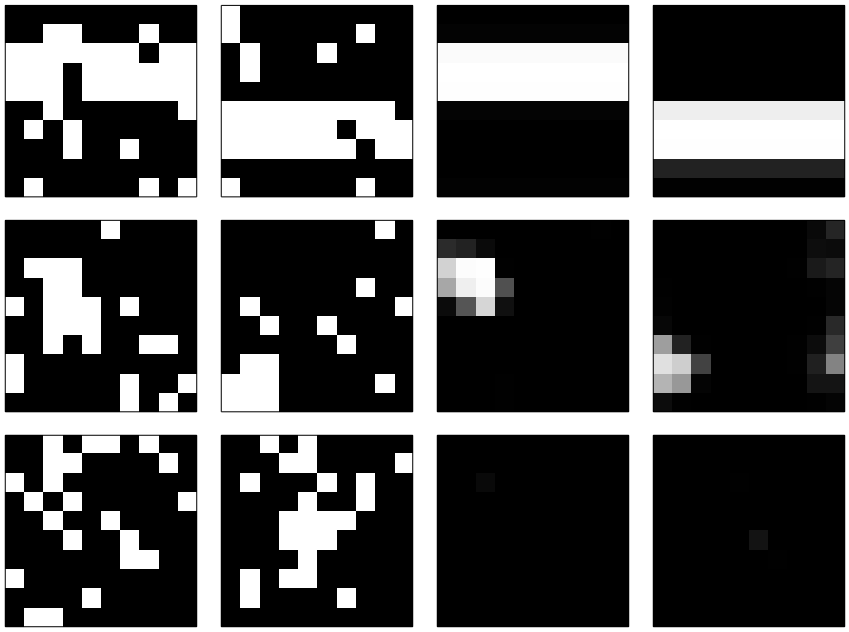

Fig. 7. Some actual input-output pairs of best networks against noisy sets. Left half: input; right half: output. Top: horizontal line; center: square; bottom: triangle.

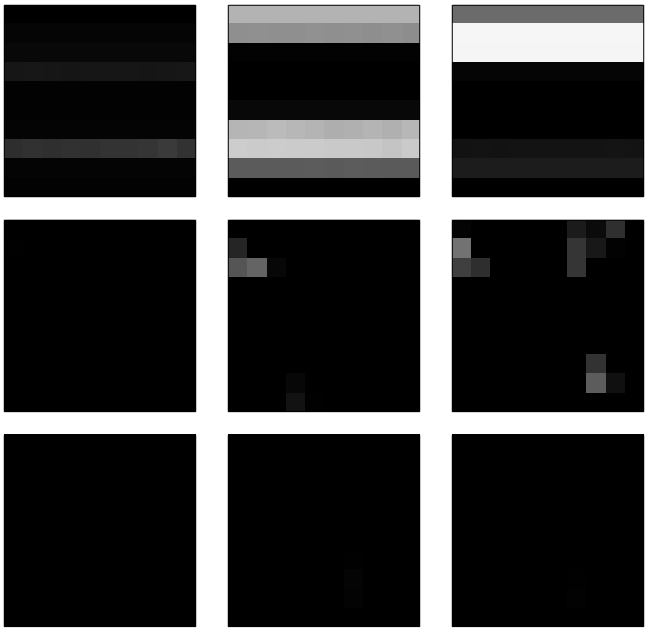

Fig. 8. Best network actual outputs against zeros, ones, and random sets. Left: zeros; middle: ones; right: random. Top: horizontal line; center: square; bottom: triangle.

As seen in Fig. 8, horizontal line, square, and triangle best networks tended to produce images similar to what it was trained for, contiguous blobs, and no features at all, respectively, when stimulated with images they were not trained for. This also holds for the noisy sets. For horizontal line, the similarity could be explained by that all images containing only horizontal lines look more similar to each other than do images produced by the other two networks to images they were trained for because they all contain only horizontal lines.

Conclusion

For all three classes of grayscale images, trained autoassociative neural networks did not perform as expected. None of the models produced a network that could reproduce its training set within the error margin. Among the accepted best model for each training set, none had the expected value for f. However, all the best networks could reproduce their training sets well enough to cause no confusion to the human eye. In addition, it was discovered that the best network for horizontal line images could remove moderate salt-and-pepper noise from such images and tended to produce images containing horizontal lines when stimulated with images it was not trained for.

From this project, the author learned the purpose and parameters of autoassociative neural networks, proof of backpropagation, learning techniques such as online learning, and how to implement layered feedforward neural networks, online learning, feedforward function, and backpropagation function in MATLAB. Moreover, the author gained deeper understanding of and appreciation for neural networks as a versatile tool for performing a variety of computational tasks.

References

[1] Mark A. Kramer, “Autoassociative neural networks,” Computers & Chemical Engineering, vol. 16, no. 4, pp. 313-328, Apr. 1992. <info:doi/10.1016/0098-1354(92)80051-A>
[2] Mark A. Kramer, “Nonlinear principal component analysis using autoassociative neural networks,” AIChE Journal, vol. 37, no. 2, pp. 233-243, Feb 1991. <info:doi/10.1002/aic.690370209>
[3] David E. Rumelhart & Geoffrey E. Hinton & Ronald J. Williams, “Learning representations by back-propagating errors,” Nature, vol. 323, no. 6088, pp. 475-566, Oct. 1986. <info:doi/10.1038/323533a0>

Appendix A: Project.m

```
clear all ; close all ; clc ; diary ('B.log') ; rng (12345) ;

% Note: the variable "known" will contain all defined training and test sets,
% the variable "grown" will contain all testing results, and the variable
% "lffnns" will contain the trained neural networks and training metadata.

% Define training and test sets

known.train {1} = all_position_horizontal_line (10, 10, 3, 1, true) ;
known.train {2} = all_position_square          (10, 10, 3, 1, true) ;
known.train {3} = all_position_triangle        (10, 10, 3, 1, true) ;
for setindex = 1 : size (known.train, 2)
  for caseindex = 1 : size (known.train {setindex}, 2)
    im_col = known.train {setindex} (:, caseindex) ;
    im_col = imnoise (im_col, 'salt & pepper', 0.25) ;
    known.test_imnoise {setindex} (:, caseindex) = im_col ;
  end
end
known.test_zeros = zeros (100, 1) ;
known.test_ones  = ones  (100, 1) ;
known.test_rand  = rand  (100, 1) ;
fprintf ('known gathered.\n') ;

% Define neural network creation and training parameters

rand_amp = 1 ;                          % See lffnn.m
hidden_layers_are_sigpn = true ;        % See lffnn.m
max_training_iterations = 5e5 ;         % Upper limit of training iterations
m = 100 ;                               % Number of neurons in in/output layer
dx = 0.05 ;                             % Learning rate
for has_M = [1 2]                       % 1 if no (de)mapping layers
  for f = 1 : 20                        % Number of neurons in bottleneck layer
    for setindex=1:size(known.train,2)% Index of training set
      if has_M == 2
        M = f + 1 ;                     % Number of neurons in (de)mapping layer
        layer_spec = [m M f M m] ;
      else
        layer_spec = [m f m] ;
      end

      % Create neural network

      tic ;

      clear lffnn;
      lffnn = lffnn (layer_spec, rand_amp, hidden_layers_are_sigpn) ;

      % Train neural network: do online learning
```

```
    train = known.train {setindex} ;
    mistakes = zeros (1, size (train, 2)) - 1 ;
    % latest mistake counts per case
    training_iterations = 0 ;       % iters required to train to satisfation
    for caseindex = ceil(size(train,2)*rand(1,max_training_iterations))

      % This iteration's training set case

      X = train (:, caseindex) ;
      Y = X ;

      % Do feedforward

      activity = passforward (lffnn, X) ;

      % Determine if neural network has been sufficiently trained

      Yo = activity {end} ;
      E = abs (Yo - Y) ;
      mistakes (caseindex) = sum (E > 1/256) ;
      if mistakes == 0
        break;
      end

      % Do backpropagation

      lffnn = passbackward (lffnn, activity, Y, dx) ;
      training_iterations = training_iterations + 1 ;

    end

    training_seconds = toc ;

    % Gather training results

    lffnns {has_M, f, setindex} . lffnn = lffnn ;
    lffnns {has_M, f, setindex} . training_iterations = training_iterations;
    lffnns {has_M, f, setindex} . training_seconds = training_seconds ;

    fprintf ('lffnns{%d,%02d,%d} gathered (%f seconds).\n', ...
      has_M, f, setindex, training_seconds) ;

    % Test neural network using training and test sets

    for thing = fieldnames(known)'
      thing = char (thing) ;

      % Test

      if strcmp (thing, 'train') || strcmp (thing, 'test_imnoise')
        X = known.(thing) {setindex} ;
      else
```

```
          X = known.(thing) ;
        end
        Y = X ;
        activity = passforward (lffnn, X) ;
        Yo = activity {end} ;
        E = abs (Yo - Y) ;
        mistakes = E > 1/256 ;
        if strcmp (thing, 'train') || strcmp (thing, 'test_imnoise')
          Emean = mean (mean (E)) ;
        else
          Emean = mean (E) ;
        end
        if strcmp (thing, 'train') || strcmp (thing, 'test_imnoise')
          Emax = max (max (E)) ;
        else
          Emax = max (E) ;
        end
        if has_M
          data = activity {end - 2} ;
        else
          data = activity {end - 1} ;
        end

        % Gather test results

        grown . Yo       . (thing) {has_M, f, setindex} = Yo ;
        grown . E        . (thing) {has_M, f, setindex} = E ;
        grown . mistakes . (thing) {has_M, f, setindex} = mistakes ;
        grown . Emean    . (thing) {has_M, f, setindex} = Emean ;
        grown . Emax     . (thing) {has_M, f, setindex} = Emax ;
        grown . H        . (thing) {has_M, f, setindex} = data ;

      end

      fprintf ('grown.*.*{%d,%02d,%d} gathered.\n', has_M, f, setindex);

    end
  end
end

% Save gathered results

save ('B.mat', 'known', 'lffnns', 'grown') ;
fprintf ('%s, %s, %s saved.\n', 'known', 'lffnns', 'grown') ;
```