Katholieke
Universiteit
Leuven

Faculteit Toegepaste Wetenschappen

Specialized advanced studies
Master in Artificial Intelligence

Artificial Neural Network: Assignment

Jin Yu (s0105853)
MAI-ECS: 2003-2004

Table of Contents

1 Exercise 1 2
1.1 Data Preparation 2
1.2 Network Design 2
1.3 Network Training 3
1.3.1 Training Functions 3
1.3.2 Early Stopping 5
1.3.3 Test Error Monitored 5
1.4 Network Testing 6
1.5 Conclusion 6
2 Exercise 2a 7
2.1 Data Preparation 7
2.2 Network Design 7
2.3 Network Training 8
2.4 Network Testing 9
2.5 Conclusion 9
3 Exercise 2b 9
3.1 Data Preparation 9
3.2 Network Design 10
3.3 Network Testing 10
3.4 Conclusion 11
A Matlab Code 12
A.1 Exercise 1 12
A.1.1 main_exe1.m 12
A.1.2 generate_data.m 12
A.1.3 create_network.m 14
A.1.4 train_network.m 14
A.1.5 plot_result.m 15
A.2 Exercise 2a 17
A.2.1 main_exe2a.m 17
A.2.2 generate_chars.m 18
A.2.3 generate_charsn.m 19
A.2.4 create_network.m 20
A.2.5 train_network.m 20
A.2.6 test_network.m 21
A.2.7 plot_result.m 22
A.3 Exercise 2b 23
A.3.1 main_exe2b.m 23
A.3.2 create_network.m 24
A.3.3 test_network.m 24
A.3.4 plot_result.m 26

1 Exercise 1

This exercise deals with the approximation of functions by neural networks. The so called function approximation (regression), is to find a mapping f' satisfying $\| f'(x) - f(x) \| < e$, (e is the tolerance; $\|\cdot\|$ can be any error measurement). In general, it is enough to have a single layer of nonlinear neurons in a neural network in order to approximate a nonlinear function.The goal of this exercise is then to build a feedforward neural network that approximates the following function:

$$f(x,y) = cos(x + 6*0.35y) + 2*0.35xy \quad x,y \in [-1\ 1]$$

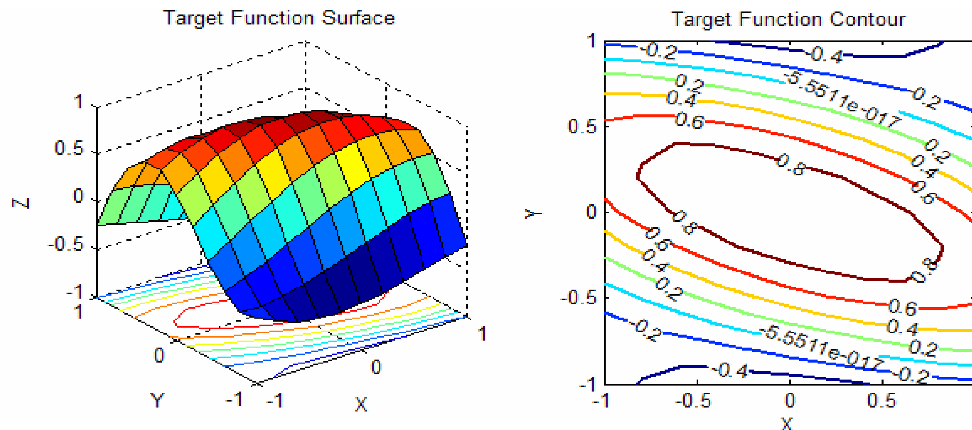

Fig. 1: Parametric surface and contour of the target function

1.1 Data Preparation

For this function approximation problem, three kinds of data sets are prepared, namely the training set, the validation set and the test set. The training set is set of value pairs which comprise information about the target function for training the network. The validation set is associated with the early stopping technique. During the training phase, the validation error is monitored in order to prevent the network from overfitting the training data. Normally, the test set is just used to evaluate the network performance afterwards. But, in this exercise the root mean-square error (Erms) on the test set is used as the performance goal of the network training.

For the current problem, the training and the test data are taken from uniform grids (10x10 pairs of values for the training data, 9x9 pairs for the test data). As shown in Fig.1 the range of the function output is already within the interval [-1 1]. So, it is not necessary to scale the target function. For the validation data, in order to make it a better representation of the original function, it is taken randomly from the function surface.

1.2 Network Design

Theoretical results indicate that given enough hidden (non-linear) units, a feedforward neural network can approximate any non-linear functions (with a finite number of discontinuities) to a required degree of accuracy. In other words, any non-linear function can be expressed as a linear combination of non-linear basis functions. Therefore, a two-layer feedforward neural network with

one layer of non-linear hidden neurons and one linear output neuron seems a reasonable design for a function approximation task. The target function as defined above has two inputs (x, y), and one output (z = *f*(x,y)). Thus, as shown in Fig.2, the network solution consists of two inputs, one layer of *tansig* (Tan-Sigmoid transfer function) neurons and one *purelin* (linear transfer function) output neuron.

The number of the hidden neurons is an important design issue. On the one hand, having more hidden neurons allows the network to approximate functions of greater complexity. But, as a result of network's high degree of freedom, it may overfit the training data while the unseen data will be poorly fit to the desired function. On the other hand, although a small network won't have enough power to overfit the training data, it may be too small to adequately represent the target function. In order to choose a reasonable amount of hidden neurons, three different networks with 2, 8 and 50 hidden neurons are examined. The training result (see Fig.3) shows the network with 8 hidden neurons outperforms the other two networks after they are trained with the same training parameters.

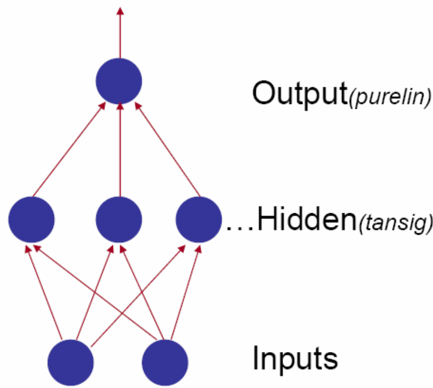

Fig.2: Network Architecture

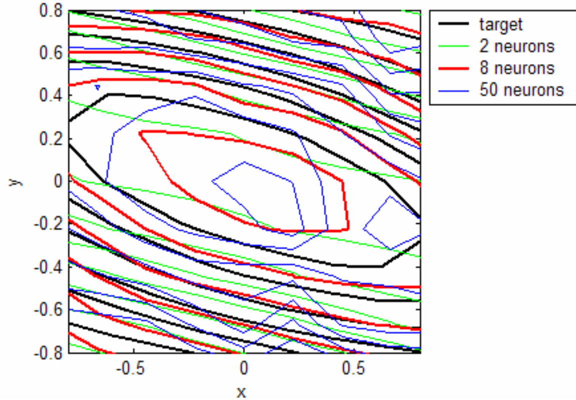

Fig.3: Function Contours

1.3 Network Training

In general, we can train a network in two kinds of styles: batch training or incremental training. In batch training, weights and biases of the network are only updated after all of the inputs are presented to the network, while in incremental (on-line) training the network parameters are updated each time an input is presented to it. I choose to apply the batch training to the current network, because it is a static network (has no feedback or delays), and the batch training is supposed to work faster and reasonably well on a static network.

1.3.1 Training Functions

There are a number of batch training algorithms which can be used to train a network. In this exercise, the following four training algorithms are examined.

- *trainbfg* implements BFGS (Shanno) quasi-Newton algorithm, which is based on the Newton's method ($x_{k+1} = x_k - A_k^{-1} g_k$, A_k: second derivatives of the performance). This training algorithm computes the update of approximated A_k (Hessian matrix) as a function of the gradient.

Generally, it converges in a few iterations. However for very large networks *trainbfg* may not be a good choice because of its computation and memory overhead. For small networks, however, *trainbfg* can still be an efficient training function.

- *traingd* implements a basic gradient descent algorithm. It updates weights and biases in the direction of the negative gradient of the performance function. The mayor drawback of *traingd* is that it is relatively slow (especially when the learning rate is small) and has a tendency to get trapped in local minima of the error surface (where the gradient is zero.).

- *traingdm* improves *traingd* by using momentum during the training. Momentum allows a network to ignore the shallow local minimum of the error surface. In addition, *traingdm* often provides a faster convergence than *traingd*.

- *trainlm* implements the Levenberg-Marquardt algorithm, which works in such a way that performance function will always be reduced at each iteration of the algorithm. This feature makes *trainlm* the fastest training algorithm for networks of moderate size. Similar to *trainbfg*, *trainlm* suffers from the memory and computation overhead caused by the calculation of the approximated Hessian matrix and the gradient.

In order to examine the performance of the training functions mentioned above, they are applied to the two-layer feedforward network respectively with the performance goal (MSE= 0.02 for the training set), maximum number of epochs to train (100) and the learning rate (0.02) being the same (without using early stopping). Within 100 epochs, *trainbfg* and *trainlm* achieve the performance goal while *traingd* and *traindm* fail. Table 1 provides detailed information on the training performance of these four functions. It shows that *trainbfg* and *trainlm* spend more time in each epoch than the gradient descent algorithms, which is the result of their computation overhead. Although more time is spent in each epoch, the total time spent by *trainbfg* and *trainlm* to reach the goal is less. As indicated by the correlation value (see Fig.4), *trainlm* produces the highest correlation between targets and network outputs. Fig.5 shows that *trainlm* is the best choice for the current problem.

Table 1: Training performance of 4 different training functions

training function	epochs	time per epoch(sec)	total time (sec)	Correlation
trainbfg	30	0.0253	0.761	0.962
traingdm	100	0.00851	0.8510	0.739
traingd	100	0.00871	0.8710	0.443
trainlm	6	0.055	0.330	0.998

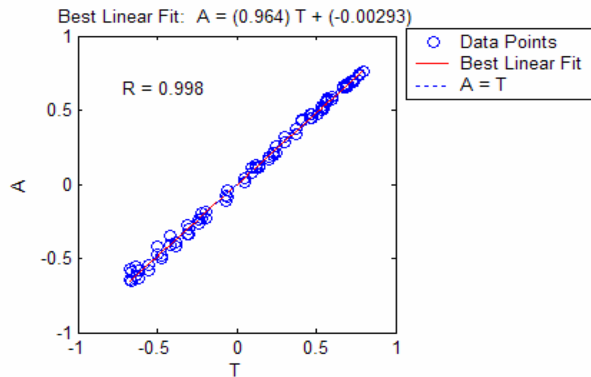

Fig.4: Linear Regression Analysis

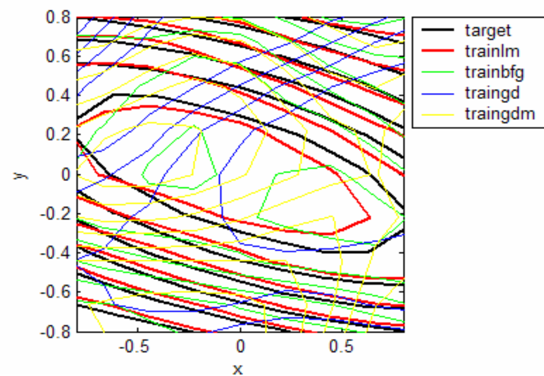

Fig.5: Function Contours

1.3.2 Early Stopping

As mentioned in section 1.2, the number of hidden neurons has a large influence on the behavior of the network. If the size of the network is too large it may run a risk of overfitting the training set and looses its generalization ability for unseen data. One method for improving network generalization ability is to use a network that is just large enough to provide an adequate fit to the target function. But sometimes it is hard to know beforehand how large a network should be for a specific application. One commonly used technique for improving network generalization is *early stopping*. This technique monitors the error on a subset of the data (validation data) that does not actually take part in the training. The training stops when the error on the validation data increases for a certain amount of iterations.

In order to examine the effect of early stopping on the training process, a randomly generated validation set is used during the *trainlm* training (Maximum validation failures=10, Erms=0.02 for the test set). As indicated by Fig.6, the early stopping mechanism is not triggered during the training. That is because the validation error keeps decreasing during the whole training process. In Fig.7, we can see that both networks (trained with and without early stopping) work equally well on the current approximation problem. This result also indicates that 8 hidden neurons is a good choice for the current problem.

1.3.3 Test Error Monitored

Normally, the test set doesn't take part in the training process. However, in this exercise, it is required that the network should be trained until Erms = 0.02 for the test set. The training process then goes as followings. Initially, the performance goal for the training set is set to be a relatively large value (MSE=0.02). Then, after each training process, the network is simulated and Erms on the test set is monitored. If Erms is larger than 0.02, the training is resumed for a lower performance goal for the training set (e.g. decreases by a factor of 0.5). Otherwise, the training stops. In current Matlab program, the performance of the trained network is evaluated by using the test set. Actually, it may introduce some bias on the result, because the test set is virtually used in the training phase. So, it would be better, if some other randomly generated data can be used for testing the network performance.

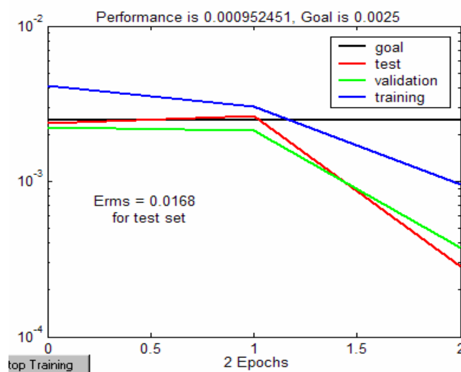

Fig.6: Evolution of the MSE

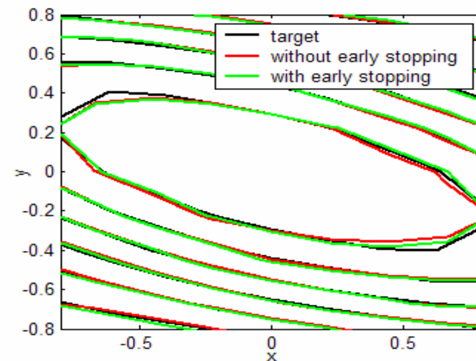

Fig.7: Function Contours

1.4 Network Testing

After the training process, the performance of the trained network will be evaluated by applying unseen data to it and checking whether its outputs are still relevant to the targets. We can use Matlab routine *postreg* to measure the network performance, which implements a regression analysis between the network response and the corresponding targets. Fig.8 (a) is the graphical output provided by *postreg*. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In Fig.8 (a) it is difficult to distinguish the best linear fit line from the perfect fit line, which indicates that the trained network has a good performance.

1.5 Conclusion

A two-layer network with two inputs, eight *tansig* hidden units and one *purelin* output unit is built for the approximation problem mentioned above. The network is trained by *trainlm* until the performance goal Erms=0.02 is achieved for the test set. No early stopping is used during the training. The maximum number of epochs to train and the learning rate are set to be 5000 and 0.02 respectively. As shown in Fig.8 (b) and Fig.8(c) the network can approximate the original function well, as a result the error surface (*error = test_target - network_output*) is low.

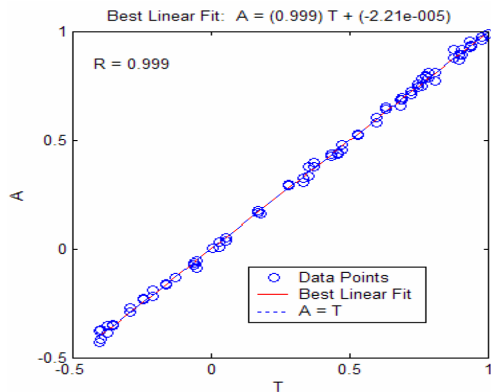

Fig.8 (a): Linear Regression Analysis

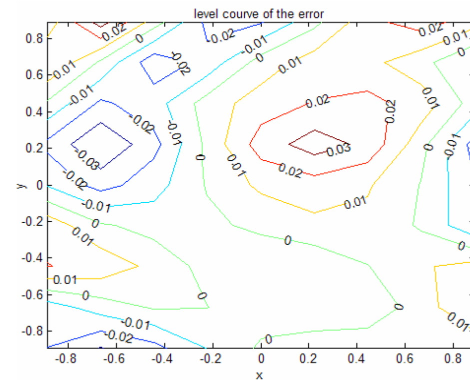

Fig.8 (b): Level Curve of the error

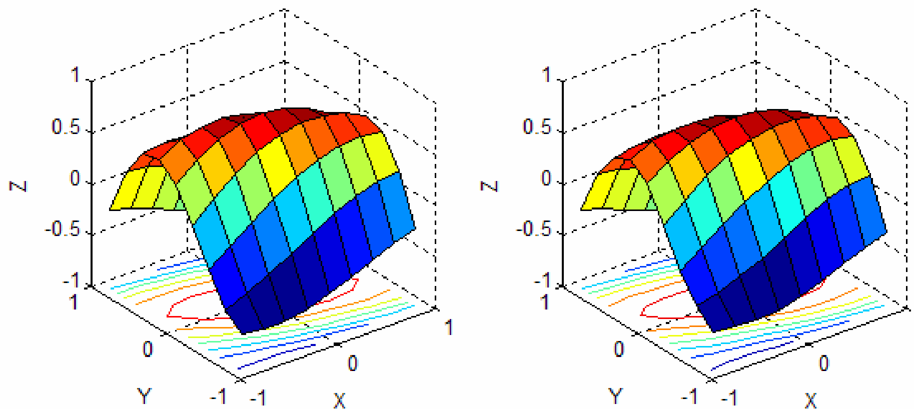

Fig.8 (c): Parametric Surfaces of the original and the approximated functions

2 Exercise 2a

The task of this exercise is to build a multilayer feedforward network for pattern recognition. The network is trained as a character classifier for a collection of characters given as 7x5 black- white pixel maps. Ideally, the trained network can recognize characters it has learnt even when some of them are distorted.

2.1 Data Preparation

In general, there are two kinds of data prepared for training and testing the network. One is the collection of thirty-one 35-element input vectors, which represent the target patterns (see Fig.9): 26 capital characters and 5 lower case characters of my name: jinyu. Another part of the data is collected by randomly reversing three bits of original characters (see Fig.10). This time, in stead of using early stopping to improve the generalization ability of the network, the network is trained on both parts of the data mentioned above, which enables it response correctly to both ideal and partially corrupted patterns.

Fig.9: Target Patterns

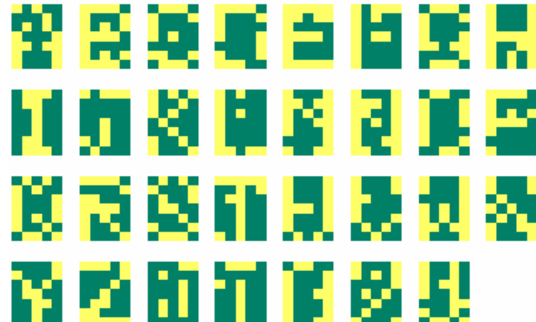

Fig.10: Distorted Patterns

2.2 Network Design

In principle, two-layer networks with sigmoidal hidden units can approximate arbitrarily well any functional continuous mapping from one finite-dimensional space to another, provided the number of hidden units is sufficiently large. As the target patterns defined in this exercise is relatively simple, which are defined by only 35 Boolean values. Therefore, a two-layer feedforward network is supposed to be power enough for this character recognition task. As 31 target characters are represented by 35-element input vectors, the neural network needs 35 inputs and 31 output neurons (see Fig.11). The network receives 35 Boolean values, which represents one character. It is then required to identify the character by give an output vector, the element of which with highest value indicates the class of input character. The *logsig* (Log Sigmoid) is chosen as the transfer function for both hidden and output layers. This is because it has a suitable output range ([0 1]) for the current problem.

The number of hidden neurons is initially set to be 10. But, the network with 10 hidden neurons has trouble recognizing distorted patterns even if it has been trained on noisy patterns. The percentage of recognition errors is about 21% when it is tested on patterns with 3 bits noise (see Fig.12). Therefore, 5 neurons are added to the hidden layer. As shown in Fig.12, the recognition error rate decreases to an acceptable value (12%). Thus, 15 is chosen as the number of the hidden neurons

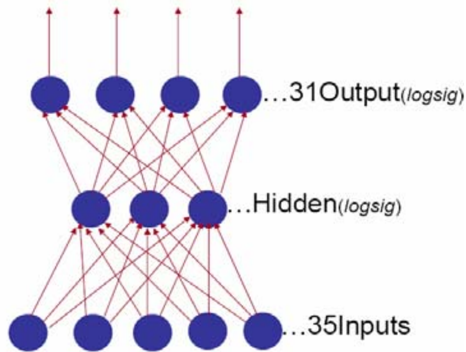

Fig.11: Network Architecture

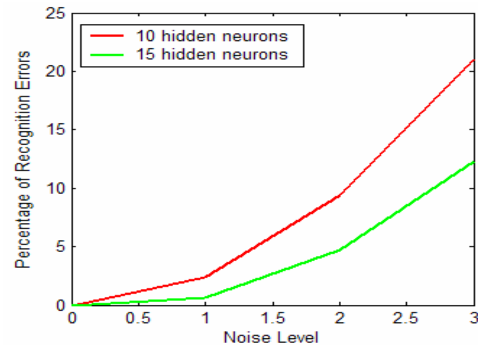

Fig.12: Percentage of Recognition Errors

Another important design issue is the choice of the initial weights and bias. In general, weights and bias should be initialized to small values so that the active region of each neuron is not close to the irresponsive (saturate) part of the transfer function; otherwise the network won't be able to learn. When using the Matlab routine *newff* to create a network, each layer's weights and biases are initialized automatically. In the program, the automatically created layer weight from the hidden layer to the output layer and the bias of the output layer are scaled down by a factor of 0.01.

2.3 Network Training

After the network is created, it is then ready for training. A gradient decent training function with momentum and adaptive learning rate (*traingdx*) is chosen to train the network. For the pattern recognition task, it is important that the noisy patterns can still be correctly classified. Thus, in order to make the network insensitive to the presence of noise, it is trained on not only ideal patterns but also noisy patterns. In the program, a three-step training process is implemented. In the first step, the network is trained on the ideal data for zero decision errors (see Fig.13 (a)). In the second step, the network is trained on noisy data as shown in Fig.10 for several passes (e.g.10 passes) for a proper performance goal (0.01 is used in the program). Unfortunately, after the network is trained for recognizing noisy patterns, it will probably "forget" those noise-free patterns it has learnt before. Therefore, in order to recall the network of these non-distorted characters, in the final step, it is trained again on just ideal data for zero decision errors (see Fig.13 (b)). The three-step training process mentioned above enables the trained network to identify both noise-free and noisy characters (within certain error tolerance).

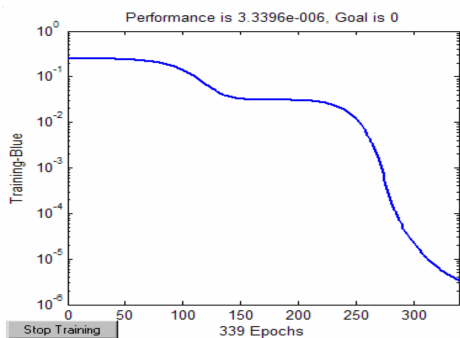

Fig.13 (a): Evolution of the MSE in Step 1

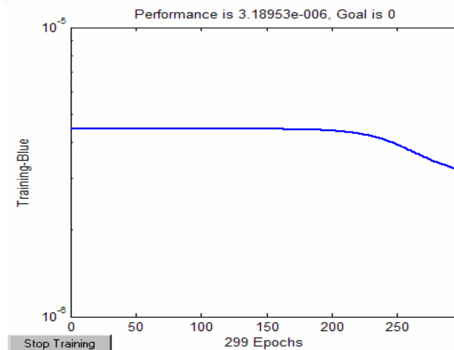

Fig.13 (b): Evolution of the MSE in Step 3

2.4 Network Testing

Once the network is trained, the test data which consists of both noise-free and slightly distorted patterns are fed to the network to check the training result. Here, the average recognition error rate is used as the performance measure. As shown in Fig.14, the network trained with noise works perfectly on noise-free patterns. And, it outperforms the one trained only on the noise-free data when input patterns are distorted to certain levels.

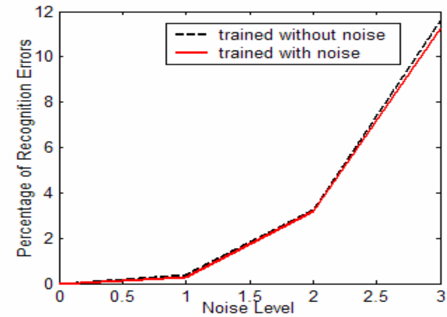

Fig.14: Percentage of Recognition Errors

2.5 Conclusion

In this exercise, a two-layer feedforward network, which has 35 inputs, 15 hidden *logsing* units and 31 *logsig* output units, is built for the character recognition task. In order to make sure the network has the ability to identify both noise-free and distorted input patterns. The three-step training method as described in section 2.4 is implemented. The trained network works excellently on noise-free patterns (almost 0% error rate is achieved.). In addition, about 90% of noisy patterns can also be correctly classified. As can been seen in Fig.15 (b), when noisy patterns are presented (see Fig.15 (a)) to the network, it can still recognize most of these distorted patterns.

Fig.15 (a): Distorted Input Patterns

Fig.15 (b): Network Output

3 Exercise 2b

For the same character recognition problem, in this exercise a Hopfield network is designed to store these patterns so that they can be retrieved from noisy cues. The Hopfield network does this by creating an energy surface (see Fig.16) which has attractors (local minimum of energy function) representing each of the patterns. The noisy cues are states of the system which are close to the attractors. As a Hopfield network evolves it slides from the noisy pattern down the energy surface into the closest attractor - representing the closest stored pattern.

3.1 Data Preparation

Hopfield networks learn by being presented with target patterns. So those noise-free patterns as used in exercise2a are prepared as the target stable states of the network. For testing purpose, noisy

patterns are also prepared. Moreover, inputs patterns are encoded by binary values {-1 1} (instead of Boolean values {0 1}).

3.2 Network Design

In Matlab, a Hopfield network can be created by calling *newhop*. *newhop* creates a Hopfield network with stable points at the target vectors, which are presented to it as the argument. But, Theoretical results indicate that if we require almost all the required patterns to be stored accurately, then the maximum number of patterns P_{max} is $N/(2*lnN)$ (N: the number of neurons). Therefore, theoretically, the storage capacity of the network created here is about 5 patterns (35/(2*ln35)). But, it is found that for the character problem defined here 8 patterns (stable states) can be stored accurately by the current network. As can be seen in Fig.17, the network trained for 8 patterns can successfully converge to the designed stable states even when there is noise present in input patterns.

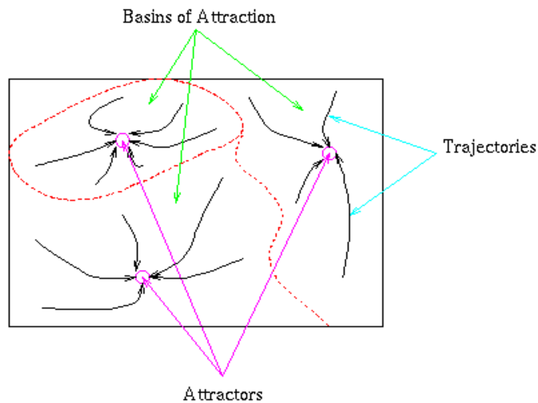

Fig.16: Energy Landscape

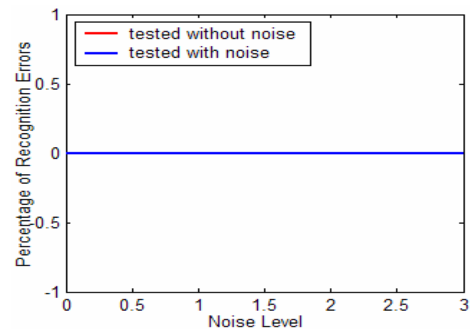

Fig.17: Recognition Error Rate
(8 patterns stored)

3.3 Network Testing

For Hopfield network, there is no need to perform iterative training on it. That is because Hopfield networks learn patterns in a one-shot style. So, once the network is created by supplying target vectors, or pattern vectors, stable equilibrium points at these target vectors are store in the network. As indicated by Fig.17, the network stores 8 patterns can achieve 0% recognition error rate. However, if more patterns are stored (e.g. 20 patterns), the recognition error may increase a lot when there is noise present in input patterns (see Fig.18). In addition, as shown in Fig.19 the network converges to some stable states, which are not related to any of the original patterns. These unexpected stable states are referred to as spurious states, which are of three types:

- If y is stable, then $–y$ is stable due to the ± symmetry of the network dynamics
- “Mixtures” of an odd number of stable states are stable ($y = sgn(y_1+y_2+y_3)$)
- Spin glass states (random uncorrelated states)

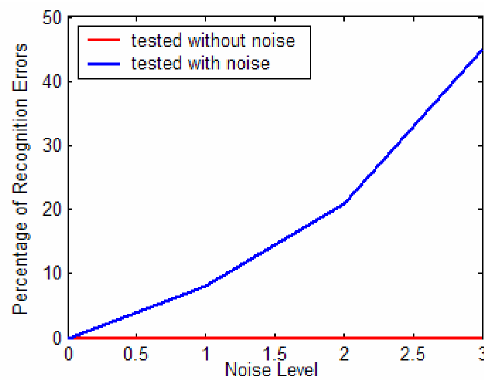

Fig.18: Recognition Error Rate
(20 patterns stored)

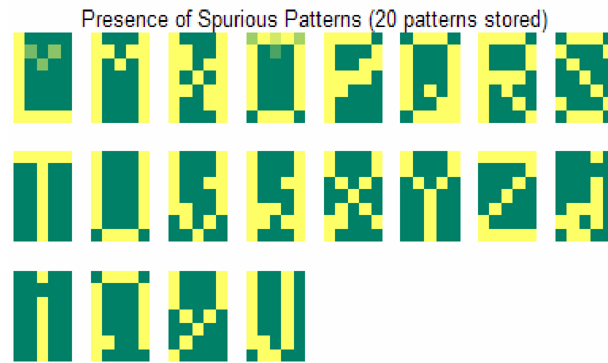

Fig.19: Presence of Spurious Patterns
(20 patterns stored)

3.4 Conclusion

In this exercise, a Hopfield network is constructed for the character recognition task. 8 patterns can be accurately stored in the network. 0% recognition error rate is achieved even when input patterns are corrupted. However, when the number of the stored patterns increases, the network degrades a lot. At the same time, it is found that after several iterations the network converges to some unexpected patterns, which are referred to as spurious patterns. From this exercise, we can see that Hopfield networks have two major limitations. First, the number of patterns that can be stored and accurately recalled is severely limited. Second, if too many patterns are stored, the network may converge to unwanted spurious states which are different from all original patterns. These drawbacks of Hopfield networks make them seldom used in practice.

Compared to Hopfield networks, feedforward networks as discussed in section 2 are more efficient and reliable networks, especially for pattern recognition tasks. In exercise 2a the properly trained feedforward network can perfectly recognize all the original patterns. Even if the patterns presented to the network are partially distorted, the network can still correctly identify those patterns in 90% of the cases. Therefore, it is clear that feedforward networks are more suitable for pattern recognition tasks than Hopfield networks.

A Matlab Code

A.1 Exercise 1

A.1.1 main_exe1.m

```
% Main script for problem_1: Function Approximation

% clear the memory of the workspace
% clear all
% clear the command window
clc
% close all the figures
close all

fprintf ('\t-------------------------------------\n');
fprintf ('\t- Problem 1: Function Approximation -\n');
fprintf ('\t-------------------------------------\n\n');

% >>>>> STEP 1: Generate training and test data <<<<<
fprintf ('Step 1: Generate training and test data...\n');
fprintf ('===========================================\n');
[train_input,train_target,test_input,test_target,val_input,val_target] = generate_data;
fprintf ('Data generation is finished ! \n\n');

% >>>>> STEP 2: Create a two layer feedforward network <<<<<
fprintf ('Step 2: Create a two layer feedforward network...\n');
fprintf ('=================================================\n');
net = create_network;
fprintf ('Network creation is finished ! \n\n');

% >>>>>STEP 3: Train the network for Erms=0.02 for test set <<<<<
fprintf ('Step 3: Train the network...\n');
fprintf ('============================\n');
[error,network_output]=
train_network( net,train_input,train_target,test_input,test_target,val_input,val_target);
fprintf ('Network training is finished ! \n\n');

% >>>>>FINAL step: Plot the result... <<<<<
fprintf ('FINAL step: Plot the result...\n');
fprintf ('==============================\n');
plot_result(test_input,test_target,network_output,error);
fprintf ('Hope the training result is good : )');
```

A.1.2 generate_data.m

```
function [train_input,train_target,test_input,test_target,val_input,val_target] = generate_data()
% GENERATE_DATA - Generate training and test data
```

```
%
%	Returns:
%		train_input - 2 x R matrix of training data for R number of x
%		and y combination in the form of [x y]'
%		train_target - row vector holding values of z (z = f(x,y))
%		test_input    - 2 x R matrix of test data for R number of x and
%		y combination in the form of [x y]'
%		test_target    - row vector holding values of z (z = f(x,y))
%		val_input      - 2 x R matrix of validation data R number of x
%		and y combination in the form of [x y]'
%		val_target      - row vector holding values of z (z = f(x,y))


train_x = -1:2/9:1; % training data [-1 1]
train_y = train_x; % training data
test_x = (-1+1/9):2/9:(1-1/9); % test data [-1 1]
test_y = test_x; % test data
val_x    = premnmx(rand(1,50)); % validation data [-1 1]
val_y    = val_x; % validation data

[train_X, train_Y] = meshgrid(train_x, train_y);
[test_X, test_Y]      = meshgrid(test_x, test_y);
[val_X, val_Y]          = meshgrid(val_x, val_y);

% Studentcard number : s0105853, coefficient a = 35/100
a = 35/100;

% functin output is within [-0.8 0.8],so no need to sacle the function
train_Z = cos(train_X + 6*a*train_Y) + 2.0*a*train_X.*train_Y; % training target
test_Z = cos(test_X + 6*a*test_Y) + 2.0*a*test_X.*test_Y; % test target
val_Z = cos(val_X + 6*a*val_Y) + 2.0*a*val_X.*val_Y; % validation target

% plot the function
[X,Y] = meshgrid(-1:.2:1,-1:.2:1);
Z = cos(X + 6*a*Y) + 2.0*a*X.*Y;

figure,
subplot(1,2,1);
surfc(X,Y,Z); %    plot parametric surface
xlabel('X');
ylabel('Y');
zlabel('Z');
title('Target Function Surface');

subplot(1,2,2);
[C,h] = contour(X, Y, Z); % plot level curve
title('Level curve of the target function');
set(h,'LineWidth',2);
clabel(C,h);
xlabel('X');
ylabel('Y');
```

```
% Return inputs [-1 1] and outputs[-0.8 0.8]
train_input = [train_X(:)'; train_Y(:)'];
train_target = train_Z(:)';

test_input = [test_X(:)'; test_Y(:)'];
test_target = test_Z(:)';

val_input = [val_X(:)'; val_Y(:)'];
val_target = val_Z(:)';
```

A.1.3 create_network.m

```
function net = create_network()
% CREATE_NETWORK - Create a feed-forward backpropagation network with 2
%                  inputs, one hidden layer and one output.MSE is chosen as
%                  the performance function
%
%    Returns:
%         net   -   Network object created
%

% ask the user for the network parameters
num_h      = getInput('Size of the hidden layer[8] -> ',8);
transFcn_h = getInput('Transfer function of the hidden layer[tansig]-> ','tansig','s');
transFcn_o = getInput('Transfer function of the output layer[purelin]-> ','purelin','s');

% create the network based on the user's choice
net=newff([-1 1; -1,1],[num_h 1],{transFcn_h,transFcn_o});
```

A.1.4 train_network.m

```
function [error,network_output] =
train_network( net,train_input,train_target,test_input,test_target,val_input,val_target)
% TRAIN_NETWORK - Train the network
%
%    Arguments:
%         net      -   Neural network.
%         train_input   - Network inputs in the form of [x y].
%         train_target - A row vector of desired target z.
%         test_input     - test inputs in the form of [x y].
%         test_target    - A row vector of test target z.
%         val_input       - Validation inputs in the form of [x y].
%         val_target      - A row vector of validation target z.
%    Returns:
%         error                 - Erms for the test set
%         network_output - Network output
%
%

val.P = val_input;
```

```
val.T = val_target;

test.P = test_input;
test.T = test_target;

% ask the user for the training parameters
epoch = round( getInput('Maximum number of epochs to train [5000]: ', 5000)); % maximum number of epochs to train
Lr = getInput('Learning rate [.02]: ', .02); % learning rate
trainFcn= getInput('Training function [trainlm]-> ','trainlm','s'); % training function (Automated Regularization (trainbr))

net.trainFcn = trainFcn;
net.trainParam.lr = Lr;
net.trainParam.epochs = epoch;
net.trainParam.show = 40;         % Epochs between displays
net.trainParam.goal = 0.02;       % Mean-squared error goal
stop_crit = getInput('Use early stopping ? y/n [n]:', 'n', 's');
erms = 1;
% Training...
if(stop_crit=='n')% no stop criteria
    tic, % start a stopwatch timer.
    while erms > 0.02
        net = train(net,train_input,train_target,[],[],[],test);
        network_output = sim(net,test_input);
        error = test_target - network_output;
        erms = sqrt(mse(error)) % root mean-square error
        net.trainParam.goal = net.trainParam.goal*0.5;
    end
    toc; % prints the elapsed time since tic was used
else % use early stopping
    tic,
    net.trainParam.max_fail    = getInput('Maximum validation failures [10]:', 10);
    while erms > 0.02
        net = train(net,train_input,train_target,[],[],val,test);
        network_output = sim(net,test_input);
        error = test_target - network_output;
        erms = sqrt(mse(error)) % root mean-square error
        net.trainParam.goal = net.trainParam.goal*0.5;
    end
    toc;
end
```

A.1.5 plot_result.m

```
function plot_result( net,input,target,network_output,error)
% DISPLAY - Create displays of function surface and level curves
%
%    Arguments:
%              net                - Neural network
%              input              - Neural network input in the form of [x y]'
```

```
%                   target                 - A row vector of desired output.
%                    network_output - A row vector of network output.
%                    error                   - error
%

X    = reshape(input(1,:),9,9);
Y    = reshape(input(2,:),9,9);
Z    = reshape(target,9,9);
No = reshape(network_output,9,9);
E =    reshape(error,9,9);

% plot function surface
figure,
subplot(1,2,1);
surfc(X,Y,Z);
xlabel('X');
ylabel('Y');
zlabel('Z');
title('Target Function Surface');

subplot(1,2,2);
surfc(X,Y,No);
xlabel('X');
ylabel('Y');
zlabel('Z');
title('Approximated Function Surface');

% plof level curves...

% create level curves of error
figure,
[C,h] = contour(X, Y, E);
clabel(C,h);
xlabel('x');
ylabel('y');
title('level courve of the error')

figure,
[C,h1] = contour(X, Y, Z,'k'); % create level curve of target
set(h1,'LineWidth',2);
% clabel(C,h);
hold on
[C,h2] = contour(X, Y, No,'m'); % create level curve of approximation
% clabel(C,h);
set(h2,'LineWidth',2);
hold off
legend([h1(1);h2(1)],'target','approximation');
xlabel('x');
ylabel('y');
title('level courves of the target and approximation functions')
```

```
% M - Slope of the best linear regression.M=1 means perfect fit.
% B - Y intercept of the best linear regression.B=0 means perfect fit.
% R - Regression R-value. R=1 means perfect correlation.

figure, %create a new figure for displaying the performance
[M,B,R] = postreg(network_output,target); % check the quality of the network training

fprintf('\n\tThe slope of the best linear regression[1]: %6.5f\n',M);
fprintf('\tThe Y intercept of the best linear regression[0]: %6.5f\n',B);
fprintf('\tThe coorelation between the network output and the target[1]: %6.5f\n',R);
```

A.2 Exercise 2a

A.2.1 main_exe2a.m

```
% Main script for problem_2a: Character Recognition using MLP

% clear the memory of the workspace
clear all
% clear the command window
clc
% close all the figures
close all

fprintf ('\t----------------------------------------------------\n');
fprintf ('\t- Problem 2a: Character Recognition (MLP) -\n');
fprintf ('\t-----------------------------------------------------\n\n');

% >>>>> STEP 1: Generate alphabet image <<<<<
fprintf ('Step 1: Generate alphabet matrix...\n');
fprintf ('===================================\n');
[alphabet,targets] = generate_chars;
fprintf ('Data generation is finished ! \n\n');

% >>>>> STEP 2: Create the network <<<<<
fprintf ('Step 2: Create the network...\n');
fprintf ('=============================\n');
net = create_network(alphabet,targets);
fprintf ('Network creation is finished ! \n\n');

% >>>>>STEP 3: Train the networks using either original data or noisy data<<<<<
fprintf ('Step 3: Train the networks using \both ideal data and noisy data...\n');
fprintf ('=================================\n');
[net, netn] = train_network( net,alphabet,targets); % net: no noise, netn: with noise
fprintf ('Network training is finished ! \n\n');

% >>>>>STEP 4: Test the network <<<<<
fprintf ('Step 4: Test these two networks...\n');
fprintf ('=================================\n');
[error,errorn,noise_range,noisy_input,outputn] = test_network(net,netn,alphabet,targets);
```

```
fprintf ('Testing is finished ! \n\n');

% >>>>>FINAL step: Plot the result... <<<<<
fprintf ('FINAL step: Plot the result...\n');
fprintf ('=============================\n');
plot_result(error,errorn,noise_range,noisy_input,outputn);
fprintf ('Hope the training result is good : )');
```

A.2.2 generate_chars.m

```
function [alphabet,targets] = generate_chars()
% GENERATE_CHARS – Create target patterns
%
%    Returns:
%            alphabet        - 35x31 matrix of 5x7 bit maps for each letter.
%            targets         - 31x31 target vectors.

[alphabetC,targets] = prprob; % capital characters

% add 5 lower case characters of my name: jin yu
letter_j = [0 0 0 1 0 ...
            0 0 0 0 0 ...
            0 0 0 1 0 ...
            0 0 0 1 0 ...
            0 1 0 1 0 ...
            0 1 0 1 0 ...
            0 0 1 0 0 ]';

letter_i = [0 0 1 0 0 ...
            0 0 0 0 0 ...
            0 0 1 0 0 ...
            0 0 1 0 0 ...
            0 0 1 0 0 ...
            0 0 1 0 0 ...
            0 0 1 0 0 ]';

letter_n = [0 1 1 1 0 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ]';

letter_y = [1 0 0 0 1 ...
            1 0 0 0 1 ...
            1 0 0 0 1 ...
            0 1 0 1 0 ...
            0 0 1 0 0 ...
            0 1 0 0 0 ...
            1 0 0 0 0]';
```

```
letter_u = [1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            1 0 0 1 0 ...
            0 1 1 0 1]';


name = [letter_j,letter_i,letter_n,letter_y,letter_u];

alphabet = [alphabetC,name];
targets    = eye(31);

% show the image of alphabet
figure;
for i=1:size(alphabet,2)
    subplot(4,8,i);
    colormap('summer');
    imagesc(reshape(alphabet(:,i),5,7)',[0,1]);
    axis off;
end
```

A.2.3 generate_charsn.m

```
function noisy_alphabet = generate_charsn(alphabet,noise_level)
% GENERATE_CHARSN – Create distorted patterns
%
%     Arguments:
%            alphabet          - 35x31 matrix of 5x7 bit maps for each letter.
%            noise_level       - Number of bits which will be changed .
%     Returns:
%            noisy_alphabet - Alphabet with noise

% add noise to the original alphabet
noisy_alphabet = alphabet;

if noise_level~=0
    size_image = length(alphabet(:,1));

    % choose noise_level amount of random positions for each letter matrix
    for i=1:size(alphabet,2)

        R(i,:) = round(rand(1,noise_level)*(size_image-1)+1)+(i-1)*(size_image);
        while length(unique(R(1,:)))< noise_level % prevent same random numbers to be generated
            R(i,:) = round(rand(1,noise_level)*(size_image-1)+1)+(i-1)*(size_image);
        end

    end
```

```
    % randomly change noise_level number of bits in each letter image : 0->1 and 1->0
    noisy_alphabet(R) = imcomplement(alphabet(R));
end
```

A.2.4 create_network.m

```
function net = create_network(input,target)
% CREATE_NETWORK - Create a feed-forward backpropagation network with one
% hidden layer.
%
%     Arguments:
%                input      - Network inputs.
%               target    - Target value.
%
%     Returns:
%               net -    Network object created
%

[S2,Q] = size(target);

% ask the user for the network parameters
S1    = getInput('Size of the hidden layer[15] -> ',15);
S2    = getInput(['Size of the output layer[',num2str(S2),'] -> '],S2);
TF1 = getInput('Transfer function of the hidden layer[logsig]-> ','logsig','s');
TF2 = getInput('Transfer function of the output layer[logsig]-> ','logsig','s');

% create the network based on the user input

net = newff(minmax(input),[S1 S2],{TF1 TF2});

% scale down weights and bias
net.LW{2,1} = net.LW{2,1}*0.01;
net.b{2} = net.b{2}*0.01;
```

A.2.5 train_network.m

```
function [net,netn] = train_network( net,input,target)
% TRAIN_NETWORK - Train the network
%
%     Arguments:
%                net                   - Neural network.
%               input               - Input matrix.
%                target             - Desired output matrix.
%     Returns:
%               net - New network trained by input
%                netn - New network trained by noisy_input

% ask the user for the training parameters
epoch           = round( getInput('Number of epochs to train [5000]: ',5000)); % maximum number of
epochs to train
trainFcn      = getInput('Training function [traingdx]-> ','traingdx','s'); % training function
```

```
net.trainFcn = trainFcn;
net.trainParam.epochs = epoch;
net.trainParam.show      = 40;      % Epochs between displays
net.trainParam.goal = 0;            % Mean-squared error goal
net.trainParam.mc = 0.95;           % Momentum constant.

% Training...

% 1: train a network without noise
tic, % start a stopwatch timer.
[net,tr] = train(net,input,target);
toc; % prints the elapsed time since tic was used

fprintf ('Strike any key to train the network with noise...\n');
pause
%       A copy of the network will now be made.    This copy will
%       be trained with noisy examples of letters of the alphabet.
netn = net;

% 2: train another network with noise
tic,
% netn will be trained on all sets of noisy letters
netn.trainParam.goal = 0.01;
for pass = 1:20
% create noisy inpupt by distording 3 bits
% of every original character matrix
noisy_input = generate_charsn(input,3);
[netn,tr] = train(netn,noisy_input,target);
end

% netn is now retrained without noise to
% insure that it correctly categorizes non-noizy letters.
netn.trainParam.goal = 0;
[netn,tr] = train(netn,input,target);
toc;
```

A.2.6 test_network.m

```
function [error,errorn,noise_range,noisy_input,outputn] = test_network(net,netn,alphabet,targets)
% TEST_NETWORK - Evaluate the performance of the trained network by
% average errors.
%
%     Arguments:
%               alphabet - 35x31 matrix of 5x7 bit maps for each letter.
%               targets    - Target value
%     Returns:
%                error             - Average error of the network trained without noise
%                errorn            - Average error of the network trained with noise
%                noise_range - Noise levels
%                noisy_input - Distorded patterns with the highest noise level
```

```
%                    outputn         - Output given by netn
%

% SET TESTING PARAMETERS
noise_range = 0:3;
max_test = 100;
error = [];
errorn = [];
T = targets;

% PERFORM THE TEST
for noise_level = noise_range
    fprintf('Testing networks with %d bits of noise\n',noise_level);
    e = 0;
    en = 0;

    for i=1:max_test

        P = generate_charsn(alphabet,noise_level);
        noisy_input = P;

        % TEST NETWORK WITHOUT NOISE
        A = sim(net,P);
        AA = compet(A);
        e = e + sum(sum(abs(AA-T)))/2;

        % TEST NETWORK WITH NOISE
        An = sim(netn,P);
        AAn = compet(An);
        en = en + sum(sum(abs(AAn-T)))/2;
    end

    % AVERAGE ERRORS FOR max_test SETS OF ALL TARGET VECTORS.
    error    = [error e/size(T,2)/max_test]
    errorn = [errorn en/size(T,2)/max_test]
end

% output of netn when input patterns are distorted with the highest noise_level
result = full(AAn);
outputn = alphabet;
for i = 1:size(result,2)
    index = find(result(:,i));
    outputn(:,i) = alphabet(:,index);
end
```

A.2.7 plot_result.m

```
function plot_result( error,errorn,noise_range,noisy_input,outputn )
% PLOT_RESULT - Create displays of recognition error rate
%
%      Arguments:
```

```
%                error           - Average error of the network trained without noise
%                errorn          - Average error of the network trained with noise
%                noise_range - Noise levels
%                noisy_input - Distorded patterns with the highest noise level
%                outputn         - Output given by netn


%       Here is a plot showing the percentage of errors for
%       the two networks for varying levels of noise.
figure,
plot(noise_range,error*100,'--k',noise_range,errorn*100,'r','LineWidth',2);
xlabel('Noise Level');
ylabel('Percentage of Recognition Errors');
legend('trained without noise','trained with noise');

% give a plot of noisy inputs and outputs
% given by the network trained on noisy data
figure,
for i=1:size(noisy_input,2)
       subplot(4,8,i);
       colormap('summer');
       imagesc(reshape(noisy_input(:,i),5,7)',[0,1]);
       axis off;
end
figure
for i=1:size(outputn,2)
       subplot(4,8,i);
       colormap('summer');
       imagesc(reshape(outputn(:,i),5,7)',[0,1]);
       axis off;
end
```

A.3 Exercise 2b

A.3.1 main_exe2b.m

```
% Main script for problem_2b: Character Recognition using Hopfield network

% clear the memory of the workspace
clear all
% clear the command window
clc
% close all the figures
close all

fprintf ('\t---------------------------------------------------------\n');
fprintf ('\t- Problem 2b: Character Recognition (Hopfield)-\n');
fprintf ('\t---------------------------------------------------------\n\n');

% >>>>> STEP 1: Generate alphabet image <<<<<
fprintf ('Step 1: Generate alphabet matrix...\n');
```

```
fprintf ('===================================\n');
[alphabet,targets] = generate_chars;
s    = getInput('Store how many patterns out of 31 alphabet[8] -> ',8, [],'(t>0) & (t<32)');

alphabet = alphabet(:,(31-s+1):31); %    part of alphabet

fprintf ('Data generation is finished ! \n\n');

% >>>>> STEP 2: Create the network <<<<<
fprintf ('Step 2: Create the network...\n');
fprintf ('=============================\n');
net = create_network(alphabet);
fprintf ('Network creation is finished ! \n\n');

% >>>>>STEP 3: Test the network <<<<<
fprintf ('Step 4: Test the network...\n');
fprintf ('===========================\n');
[error,errorn,noise_range,recordn] = test_network(net,alphabet);
fprintf ('Testing is finished ! \n\n');

% >>>>>FINAL step: Plot the result... <<<<<
fprintf ('FINAL step: Plot the result...\n');
fprintf ('==============================\n');
plot_result(error,errorn,noise_range,recordn);
fprintf ('Hope the training result is good : )');
```

A.3.2 create_network.m

```
function net = create_network(target)
% CREATE_NETWORK - Create a Hopfield network
%
%     Arguments:
%                   target - Target vectors used to define stable points.
%
%     Returns:
%                  net -    Network object created
%

target = imcomplement(imcomplement(target)*2); % target -> [-1 1]

net = newhop(target);
```

A.3.3 test_network.m

```
function [error,errorn,noise_range,recordn] = test_network(net,target)
% TEST_NETWORK - Evaluate the performance of the trained network by
% average errors.
%
%     Arguments:
%                   net           - Hopfield network
%                  target    - Target vectors (35x31 matrix of 5x7 bit maps)
```

```
%      Returns:
%                 error           - Average error of the network tested without noise
%                 errorn          - Average error of the network tested with noise
%                 noise_range - Noise levels
%                 recordn         - Stable points evolving from the noisy start points
%


% check whether the target vectors are indeed stable
% [Y,Pf,Af] = sim(net,size(targets,2),[],targets);
% Y = (Y+1)/2; % transfor back -1 -> 0
%
% figure,
% % image of stored alphabet
% for i=1:size(Y,2)
%       subplot(5,7,i);
%       colormap('summer');
%       imagesc(reshape(Y(:,i),5,7)',[0,1]);
%       axis off;
% end


% SET TESTING PARAMETERS
T = imcomplement(imcomplement(target)*2);; % target -> {-1 1}
noise_range = 0:3;
max_test = 5;
error = [];
errorn = [];


steps    = getInput('Simulate the Hopfield network for how many steps[20] -> ',20);

% PERFORM THE TEST
for noise_level = noise_range
     fprintf('Testing the network with( %d bits of noise ) or without noise, please wait...\n',noise_level);

     e = 0;
     en = 0;

     for i=1:max_test

          Tn = generate_charsn(T,noise_level);

          record    = [];
          recordn = [];

          for i=1:size(T,2);

               % TEST NETWORK WITHOUT NOISE
               [y,Pf,Af] = sim(net,{1 steps},{},{T(:,i)});
               result = (y{steps}+1)/2; % transfor back to [0 1]
```

```
            record = [record result];

            % TEST NETWORK WITH NOISE
            [yn,Pf,Af] = sim(net,{1 steps},{},{Tn(:,i)});
            resultn = (yn{steps}+1)/2; % transfor back to [0 1]
            recordn = [recordn resultn];
        end

        % accumutive recognition error
        e = e + sum((any(record - target)));
        en = en + sum((any(recordn - target)));

    end

    % AVERAGE ERRORS FOR max_test SETS OF TARGET VECTORS.
    error    = [error e/size(T,2)/max_test]
    errorn = [errorn en/size(T,2)/max_test]
end
```

A.3.4 plot_result.m

```
function plot_result( error,errorn,noise_range,recordn)
% PLOT_RESULT - Create displays of recognition error rate
%
%     Arguments:
%                 error           - Average error of the network tested without noise
%                 errorn          - Average error of the network tested with noise
%                 noise_range - Noise levels
%                 recordn         - Stable points evolving from the noisy start
%                                   points with the range [0 1]


%       Here is a plot showing the percentage of errors for the
%       network testedwith no noise or with varying levels of noise.
figure,
plot(noise_range,error*100,'r',noise_range,errorn*100,'b','LineWidth',2);
xlabel('Noise Level');
ylabel('Percentage of Recognition Errors');
legend('tested without noise','tested with noise',2);


%       Here is a plot showing the result given by the Hopfield
%       network whichevolves from noisy start points.

% image of recognized alphabet
figure;
for i=1:size(recordn,2)
    subplot(4,8,i);
    colormap('jet');
    imagesc(reshape(recordn(:,i),5,7)',[0,1]);
    axis off;
```

end