1. Specify all necessary details of your implementation (for example number of hidden nodes, gain function, additional post processing if this was necessary, …)

We choose Sigmoid function $g(x) = \frac{1}{1 + e^{(-\beta (x - x0))}}$ as gain function, beta =2, which determine the steepness of the sigmoid, x0=0.5, which determine the shift of the sigmoid function along the "input" axis.

Squashing function need not be sigmoidal, but it must be differentiable. The reason we choose Sigmoid function is it's differential function is sample.

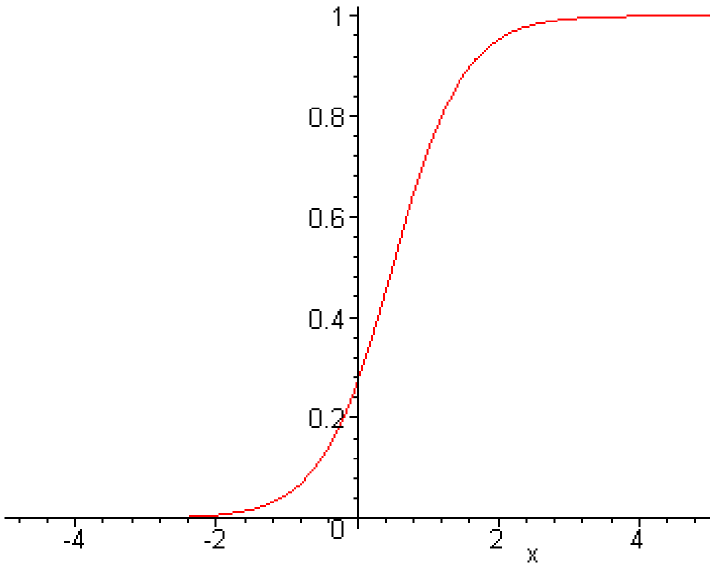

Figure 1: Curve of the gain function used on the network

We also use the following function to transfer final output value to 0 or 1 before compare them with the final desired output.

f(x)=0 if x<=0.5
f(x)=1 if x>0.5

The following shows the method that we used to generate noise input patterns.

The elements of each input pattern are randomly chosen, if it is not marked as "Selected", we change it from 0 to 1 and 1 to 0, then it is marked as "Selected". The number of elements that will be changed depends on the noise level. If the noise is 100%, all elements of the input pattern will be changed from 0 to 1 or 1 to 0.

The initial weights are set to be random value from 0 to 0.01, and learning rate is 0.1. We notice that there is a big difference of training steps for different initial weights values and different learning rate.

2. Plot the learning curve in which you show the performance of the network versus the training steps

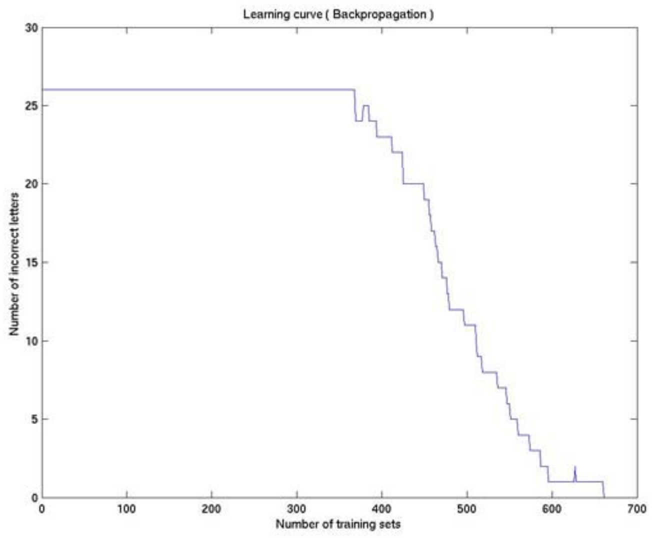

Figure 2 Learning curve showing the performance of the network versus the training steps for the 1 hidden layer with 10 hidden nodes.

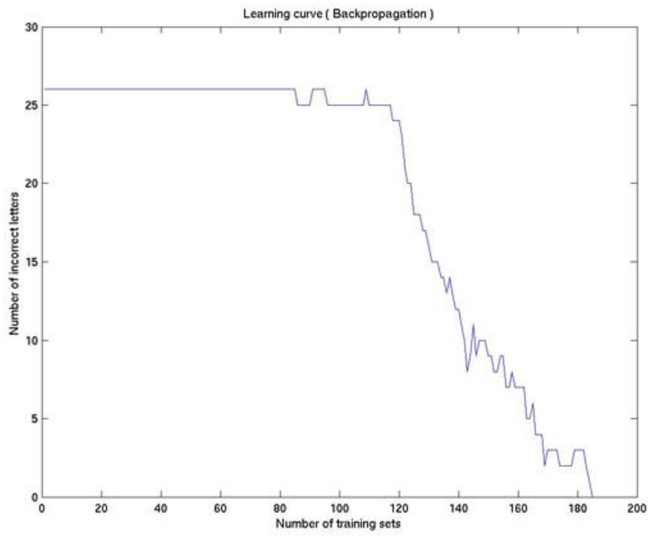

Figure 3 Learning curve showing the performance of the network versus the training steps for the 1 hidden layer and 50 hidden nodes.

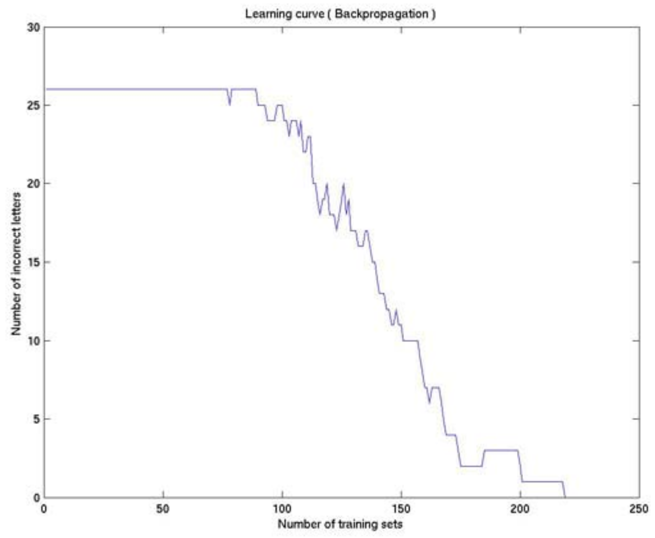

Figure 4 Learning curve showing the performance of the network versus the training steps for the 1 hidden layer and 100 hidden nodes.

We trained the network with different hidden layers and hidden nodes, as figure 5, 6, 7 showing the learning curves of the above networks. We can see that different number of nodes has different learning abilities. In some models, for this specific problem, they learn faster. For example, when we choose 1 hidden layer with 50 nodes, it takes almost the least steps to complete the learning; the learning process is slow with 10 nodes and is fast with 100 nodes, while it is not true that the network can learn faster with more nodes. When we choose one hidden layer with 200 nodes, the model was stuck, it spends hours to learn. We think that the learning ability of the networks is greatly affected by the initial weights value, learning rate, number of nodes, hidden layers in the model, and other variable parameters. We need to do more research on choosing appropriate parameters for Backpropagation algorithm in the future.

Actually, it takes a longer time to train the 2 hidden layers network. It means that an excessive number of layers and nodes numbers cause slower convergence in the back-propagation learning.

Although the networks take different steps for the learning process in this experiment, the trend is almost same:

1. At the beginning there will be a low recognition rate for certain time, i.e. the incorrect letters number is zero at the beginning.

2. At the point about 50% of the whole learning progress, the recognition ability of the network increases quickly.

3. It is different from the learning curve trained by Delta rule. The Delta rule learning process takes less time and plots a gradually declining curve.

3.Evaluate the robustness of the network in recognizing noisy versions of the letter patterns. Plot a curve that shows the average recognition rate versus the noise level. The plot should include errorbars.

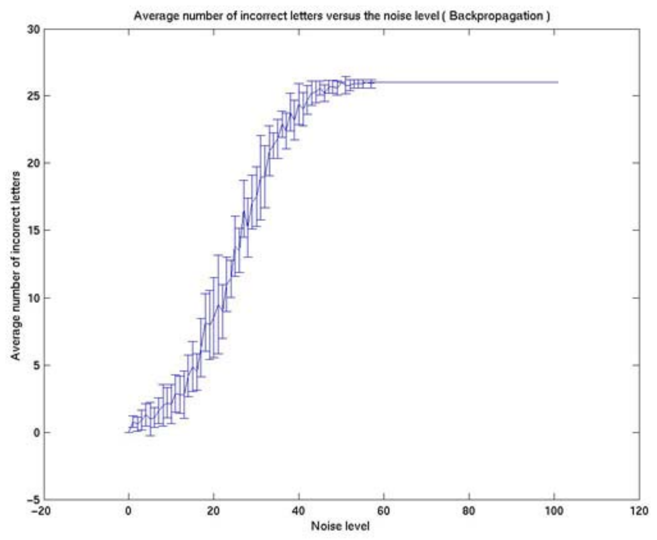

Figure 5 Average number of incorrect recognized letters versus the noise level for 1 hidden layer and 10 hidden nodes.

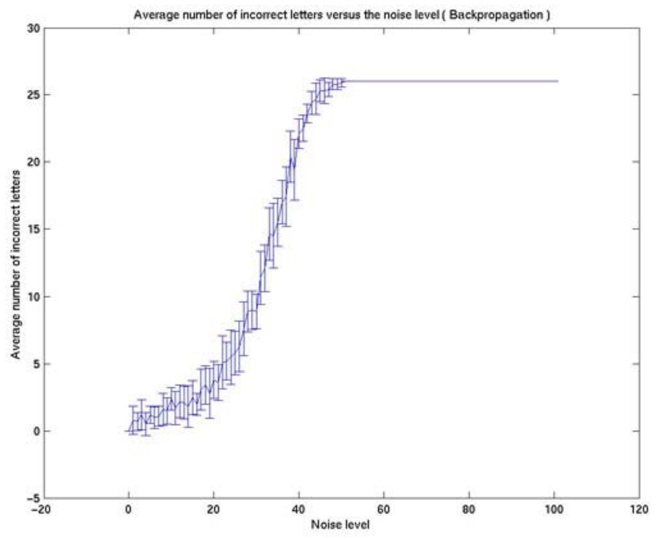

Figure 6 Average number of incorrectly recognized letters versus the noise level for 1 hidden layer and 50 hidden nodes.

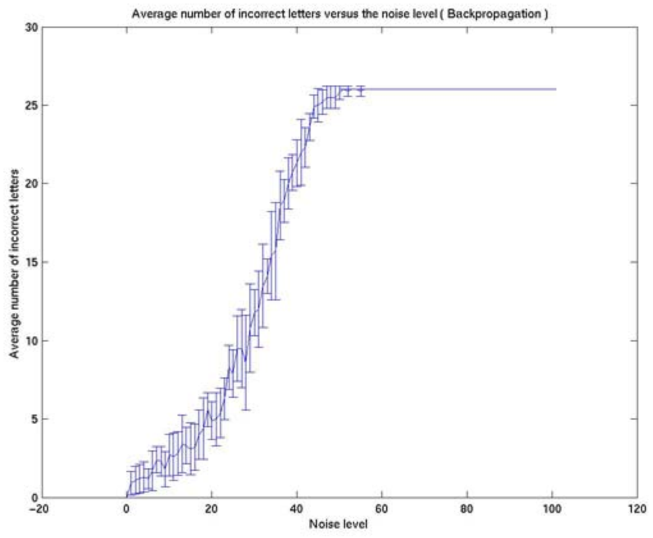

Figure 7 Average number of incorrectly recognized letters versus the noise level for 1 hidden layer and 100 hidden nodes.

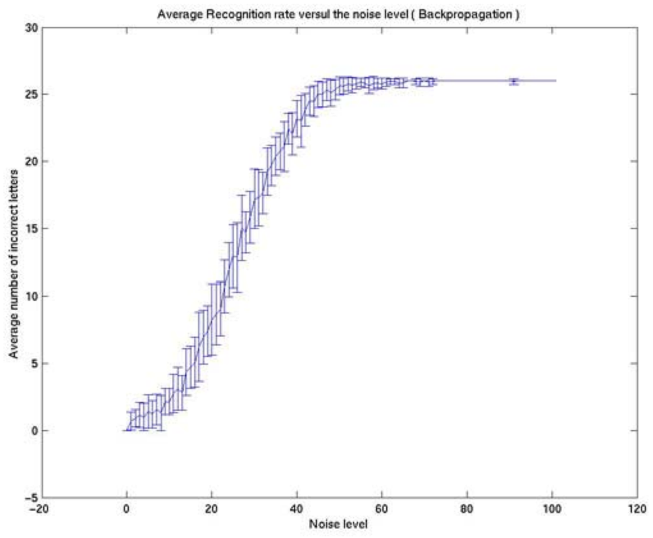

Figure 8 Average number of incorrectly recognized letters versus the noise level for 2 hidden layers with 20 hidden nodes each.

The curve shows the trend that incorrect recognition letters increase with the increase of input noise level. We use errorbar to show the deviation for the experiments.

1. At the beginning, at the noise level of 0, the correctness of recognition is 100%.

2. When the input noise value increased from 0 to about 50%, the number of incorrect letters goes up.

3. The network can not recognize the input when the noise level is greater than about 60%.

4. We tested the networks with different numbers of hidden layers and hidden nodes.

   We test the network by 1 hidden layer with different number of hidden nodes (1 hidden layer with 10,50 and 100 nodes), 2 layers with 20 hidden nodes on each hidden layer. As we can see from figure 5, 6, 7 and 8, there is no significant improvement or changing in the recognition rate versus noise input between these networks. The experiment shows that an excessive number of layers and nodes does not mean the network posses a greater recognition ability.

   Another way we test the network is to make noise while it is training. That is, prior to training, add some small random noise on the input data, then implement the learning. We trained the networks with noise input patterns with noise level from 30 to 0 (See the attached program 6. BackpropagationNoise2.m). Figure 9 shows the test result.

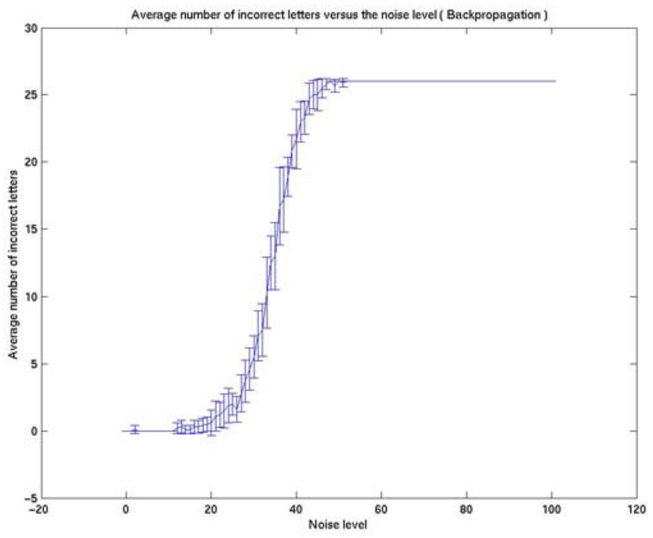

Figure 9. Average number of incorrectly recognized letters versus the noise level for 1 hidden layer and 50 hidden nodes of a network trained with noise input patterns.

As we can see from figure 9, the network has an almost 100% recognition ability if the noise level of input patterns is less than 10. At the noise level of 30, the number of incorrectly recognized letters is only about 3 or 4 compare to 7 at the figure 8. This result shows that training with noise input patterns helps to account for noise and natural variability in real data, and tends to produce a more reliable network.

4. Compare the results to the results in assignment 1.

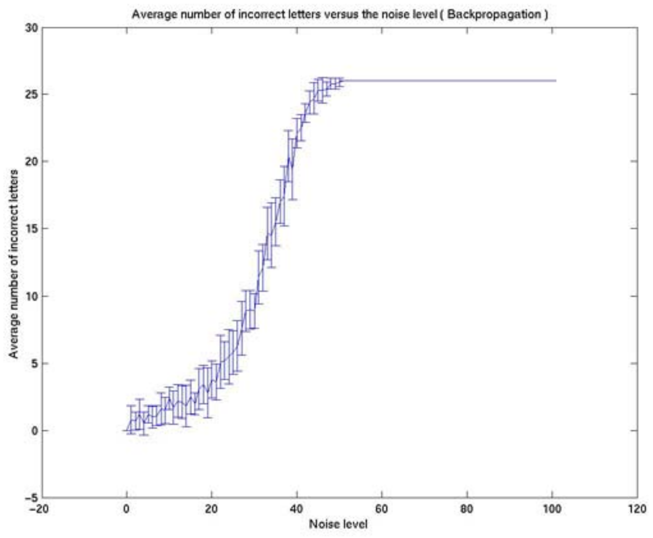

Figure 10. Average number of incorrectly recognized letters versus the noise level for 1 hidden layer and 50 hidden nodes.

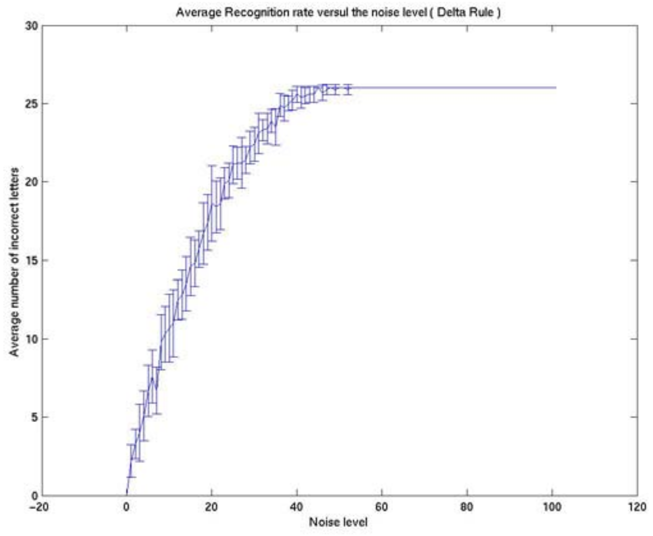

Figure 11. Average number of incorrectly recognized letters versus the noise level for Delta rule.

Compare figure 10 and figure 11, there is some difference on the recognition rate between the experiment of Delta rule and Backpropagation algorithm.

1. Before the noise level is up to 30%, there are no more than 10 error letters in the recognition for Backpropagation algorithm while in the Delta rule the error letters is more than 20.

2. When the noise level is greater than 40%, they are almost the same on recognition rate. Both have a very low recognition rate, almost zero.

Obviously, Backpropagation algorithm is a little more robust in recognizing noise versions of the letter patterns than Delta rule.

However, training of Delta rule requires no hidden layer back-propagation of error and can be done very efficiently.

On the other hand, training of Backpropagation rule is difficult.

- Knowing how to choose the parameters is not easy. There are too many works on selecting initial settings, which include beta, initial weights, learning rate, etc.

- The convergence tends to be extremely slow.

Program 1: Training program (BackPropagtion.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Backpropagation pattern recognition %
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;

% Reading pattern from pattern file

load pattern1;

% Reading desired output matrix file
load desiredOutputMatrix;

% Reshaping pattern to 156*26
letterVectors=reshape(pattern1', 12*13, 26);
trainingNumber=1;
Parameters;

disp('Begin training, please wait!');
for i=1:layerNumber-1
   weights{i}=rand(node(i+1),node(i))*.01;
end



errorNumber=26;
tmp=26;

while errorNumber>0

   for letterNumber=1:26

      % Apply sample patterns to the input nodes
      inputLetterVector=letterVectors(:,letterNumber); % 156*1
      outputVector{1}=inputLetterVector;
      for i=2:layerNumber
         % Cauculate the rate of output nodes
         h{i}=weights{i-1}*outputVector{i-1}; % 26*1
         outputVector{i}=gainFunction(h{i},beta,x0);
      end

      % Compute the delta term for the output layer
```

```
        deltaVector{layerNumber}=(desiredOutputMatrix(:,letterNumber)-
outputVector{layerNumber}); % 26*1

deltaVector{layerNumber}=gainDifferentiableFunction(h{layerNumber},beta,x0).*delta
Vector{layerNumber};

        % Backpropagation
        if layerNumber>2
            for i=layerNumber-1:-1:2
                deltaVector{i}=weights{i}'*deltaVector{i+1};
                deltaVector{i}=gainDifferentiableFunction(h{i},beta,x0).*deltaVector{i};
            end
        end

        % Update weights matrix by adding the term
        for i=2:layerNumber
            detaWeights{i-1}=learningRate*deltaVector{i}*outputVector{i-1}';
            weights{i-1}=weights{i-1}+detaWeights{i-1};
        end
    end


    % Test output with trained weight matrix
    errorNumber=0;
    for letterNumber=1:26

        % Apply sample patterns to the input nodes
        inputLetterVector=letterVectors(:,letterNumber); % 156*1
        outputVector{1}=inputLetterVector;
        for i=2:layerNumber
            % Cauculate the rate of output nodes
            h{i}=weights{i-1}*outputVector{i-1}; % 26*1
            outputVector{i}=gainFunction(h{i},beta,x0);
            g=zeros(size(outputVector{i}));
            g(outputVector{i}>0.5)=1;
                    outputVector{i}=g;
        end

        % Compare the output with the desired output, update error number variable
        if ~isequal(desiredOutputMatrix(:,letterNumber),outputVector{layerNumber})
            % error output
            errorNumber=errorNumber+1;
        end
    end

    % Update the learning curve X Y set
    X(trainingNumber)=trainingNumber;
    Y(trainingNumber)=errorNumber;

    % Add the number of training set
    trainingNumber=trainingNumber+1;
    if errorNumber<tmp
       tmp=errorNumber;
       errorNumber
    end
end
plot(X,Y);

% Plot the learning curve
plot(X,Y);
title('Learning curve ( Backpropagation )');
xlabel('Number of training sets');
ylabel('Number of incorrect letters');

disp('End of training, enjoy it.');

% Save weight matrix  to file 'w'
save('weights','weights');
```

Program 2 Parameter Setting file (Parameters.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Parameters.m                          %
% Allow user to set up variour parameters  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%

% Set learning constant and number of training
learningRate=0.1;
beta=2;
x0=0.5;
% Set layer and nodes of networks
layerNumber=3;
node(1)=156;
node(2)=50;
node(3)=26;
```

Program 3 Gain function (gainFunction.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%
% Game Function                %
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function g=gameFunction(x,beta,x0)
% Gain fuction of back propagation

g=1./(1+exp(-beta*(x-x0)));
```

Program 4 Differentiable gain function (gainDifferentiableFunction.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Differentiable Game Function                %
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function g=gameDifferentiableFunction(x,beta,x0)
% Game fuction of delta rule
% return 1 if x>0.45
% return 0 if x<=0.45
f=1./(1+exp(-beta*(x-x0)));
g=beta.*((ones(size(f))-f).*f);
```

program 5 Evaluation program (EvaluationBackPropagation.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
%
% Evaluation of Backpropagation pattern recognition %
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%

clear;

% Reading pattern from pattern file
load pattern1;

% Reading desired output matrix file
load desiredOutputMatrix;

% Reading weight matrix
load weights;
```

```
% Reshaping pattern to 156*26
letterVectors=reshape(pattern1', 12*13, 26);
Parameters;

disp('Begin testing input with noise!');

% Set the repeat test number of certain noise level
totalLoopNumber=10;

timeStart=now;
for noise=0:100
noise
   X(noise+1)=noise;
   for loopIndex=1:totalLoopNumber

      % Initialize the error number
      errorNumber=0;

      % Apply sample patterns to the input nodes
      inputLetterVectors=letterVectors;

      % Make nosie version
      for i=1:26
         noiseVec=ranBinVec(156,noise*156/100);
         inputLetterVectors(noiseVec==1,i)=inputLetterVectors(noiseVec==1,i)==0;
      end


      for letterNumber=1:26

         % Apply sample patterns to the input nodes
         inputLetterVector=inputLetterVectors(:,letterNumber); % 156*1
         outputVector{1}=inputLetterVector;
         for i=2:layerNumber
            % Cauculate the rate of output nodes
            h{i}=weights{i-1}*outputVector{i-1}; % 26*1
            outputVector{i}=gainFunction(h{i},beta,x0);
            g=zeros(size(outputVector{i}));
            g(outputVector{i}>0.5)=1;
            outputVector{i}=g;
         end

         % Compare the output with the desired output, update error number variable
         if ~isequal(desiredOutputMatrix(:,letterNumber),outputVector{layerNumber})
            % error output
            errorNumber=errorNumber+1;
```

```
          end
        end

        % Update the recognition rate versus noisie level curve X Y set

        Y(loopIndex,noise+1)=errorNumber;
    end
end;

timeEnd=now;
disp('Total time used for recognitions: ');
disp(datestr(timeEnd-timeStart,'HH:MM:SS'));

% Compute the average error number
AverageY=mean(Y);

% Comput the standard deviation
S=std(Y);

% Plot the error bar
errorbar(X,AverageY,S);
title(' Average number of incorrect letters versus the noise level ( Backpropagation )');
xlabel('Noise level');
ylabel('Average number of incorrect letters');

disp('End of testing input with noise.');
```

Program 6 Training with noise program (BackpropagationNoise2.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%
%
% Backpropagation pattern recognition Training with noise%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%

clear;

% Reading pattern from pattern file

load pattern1;

% Reading desired output matrix file
load desiredOutputMatrix;
```

```
% Reshaping pattern to 156*26
letterVectors=reshape(pattern1', 12*13, 26);
trainingNumber=1;

Parameters;

disp('Begin training, please wait!');
for i=1:layerNumber-1
   weights{i}=rand(node(i+1),node(i))*.001;
end


tmp=26;

for noise=30:-1:0
errorNumber=26;
while errorNumber>0

      noiseLetterVectors=letterVectors;

      % Make nosie version
      for i=1:26
         noiseVec=ranBinVec(156,noise*156/100);
         noiseLetterVectors(noiseVec==1,i)=noiseLetterVectors(noiseVec==1,i)==0;
      end

   for letterNumber=1:26

      % Apply sample patterns to the input nodes
      inputLetterVector=noiseLetterVectors(:,letterNumber); % 156*1

      outputVector{1}=inputLetterVector;
      for i=2:layerNumber
         % Cauculate the rate of output nodes
         h{i}=weights{i-1}*outputVector{i-1}; % 26*1
         outputVector{i}=gainFunction(h{i},beta,x0);
      end

      % Compute the delta term for the output layer
      deltaVector{layerNumber}=(desiredOutputMatrix(:,letterNumber)-
outputVector{layerNumber}); % 26*1

deltaVector{layerNumber}=gainDifferentiableFunction(h{layerNumber},beta,x0).*delta
Vector{layerNumber};

      % Backpropagation
```

```
        if layerNumber>2
            for i=layerNumber-1:-1:2
                deltaVector{i}=weights{i}'*deltaVector{i+1};
                deltaVector{i}=gainDifferentiableFunction(h{i},beta,x0).*deltaVector{i};
            end
        end

        % Update weights matrix by adding the term
        for i=2:layerNumber
            detaWeights{i-1}=learningRate*deltaVector{i}*outputVector{i-1}';
            weights{i-1}=weights{i-1}+detaWeights{i-1};
        end
    end


    % Test output with trained weight matrix
    errorNumber=0;
    for letterNumber=1:26

        % Apply sample patterns to the input nodes
        inputLetterVector=letterVectors(:,letterNumber); % 156*1
        outputVector{1}=inputLetterVector;
        for i=2:layerNumber
            % Cauculate the rate of output nodes
            h{i}=weights{i-1}*outputVector{i-1}; % 26*1
            outputVector{i}=gainFunction(h{i},beta,x0);
            g=zeros(size(outputVector{i}));
            g(outputVector{i}>0.5)=1;
                    outputVector{i}=g;
        end

        % Compare the output with the desired output, update error number variable
        if ~isequal(desiredOutputMatrix(:,letterNumber),outputVector{layerNumber})
            % error output
            errorNumber=errorNumber+1;
        end
    end

    % Update the learning curve X Y set
%    X(trainingNumber)=trainingNumber;
%    Y(trainingNumber)=errorNumber;

    % Add the number of training set
    trainingNumber=trainingNumber+1;
    if errorNumber<tmp
        tmp=errorNumber;
        errorNumber
    end
end
noise
end


disp('End of training, enjoy it.');

% Save weight matrix  to file 'w'
save('weights','weights');
```