ECE 8720 SECTION 001/002

TAKEHOME #1

PARAMETERIZED SQUASHER – TRAINING AND EFFECTS

Submitted By:

Name: VIVEK KOODLI UDUPA

CU User Name: VKOODLI

PROPOSAL

Introduction:

An artificial neuron decides its "activation" status based on the weighed sum of its inputs and weights. An activation function is used in order to give the neural network a sense of nonlinearity, which will be helpful in making the artificial neural network be more general in nature.

One of the popular activation function/squasher is the logistic squasher. The logistic squasher is a fixed activation function and has no learnable parameters. Adding a learnable parameter may help the network learn faster and perform better.

Goals:

1. Select an appropriate data set for the ANN.
2. Build a MLFF ANN with standard logistic activation function.
3. Modify the MLFF ANN to have a learnable parameterised logistic activation function.
4. Compare the performance of the modified squasher network with that of the standard squasher network.

Outcomes/Deliverables:

1. MATLAB Source code for the MLFF ANN with parameterised squasher.
2. Performance analysis of modified MLFF ANN with parameterised logistic squasher with that of the standard MLFF ANN with standard logistic squasher.

References:

[1] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[2] https://theclevermachine.wordpress.com/tag/logistic-sigmoid/

[3] https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural- networks-9491262884e0

[4] http://mi.eng.cam.ac.uk/~cz277/doc/Poster-Interspeech2015-ACT.pdf

[5] Luke B. Godfrey, Michael S. Gashler *"A parameterized activation function for learning fuzzy logic operations in deep neural networks"* (Submitted on 28 Aug 2017 (v1), last revised 11 Sep 2017 (this version, v2))

[6] Forest Agostinelli, Matthew Hoffman, Peter Sadowski, Pierre Baldi *"Learning Activation Functions to Improve Deep Neural Networks"* (Submitted on 21 Dec 2014 (v1), last revised 21 Apr 2015 (this version, v3))

Introduction

Every Artificial Neural Network has a typically fixed non-linear activation function at each neuron. The type of activation function used can have a significant impact on learning. The space of possible activation functions is huge. One of the ways to explore this space is to learn the activation function during training.

The standard logistic sigmoid is a fixed non-linear activation function with no learnable parameters. Adding parameters to the logistic sigmoid and learning these parameters during training helps the network to converge to minimum error faster.

Initial Approach

The first dilemma faced was whether to select a data set and write a Matlab code to match the specifications of the dataset (i.e the units in the input and output layers.) or to write a generic network that can have any number of layers with any number of neurons in each layer and worry about the dataset after developing the network. Since I couldn't decide on a dataset immediately, I went with the option of developing a generic network.

First 3 weeks were spent on developing the generic network. This was a great learning experience. The concepts that I failed to understand for Quiz 1 became more clear while implementing the network. The number of weights required between two layers and the Student A - Student B problem asked during Quiz 1 seems fairly obvious now.

Development

For the purpose of ease, a simple 3 layer FeedForward network for learning the XOR truth table was considered in the beginning. Simple logistic sigmoid activation function without any learnable parameters were considered at this stage. Once the feedforward and backpropagation algorithms were implemented successfully, the network was then tested with A to D Dataset.

The results of the A to D reproduced by the network is shown in Figure 1. The learning rate was set to 0.5. epoch was set to 100000. The network had 4 layers with 2 units in input, 13 units in hidden1, 17 units in hidden2 and 4 units in output layer.

```
input =

    0  0.0625  0.1250  0.1875  0.2500  0.3125  0.3750  0.4375
    1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000

target =|

    0   0   0   0   0   0   0   0
    0   0   0   0   1   1   1   1
    0   0   1   1   0   0   1   1
    0   1   0   1   0   1   0   1

output =

    0.0000   0.0000   0.0000   0.0000   0.0002   0.0005   0.0029   0.0042
    0.0000   0.0000   0.0000   0.0098   0.9881   0.9996   1.0000   1.0000
    0.0000   0.0038   0.9968   0.9955   0.0035   0.0034   0.9985   0.9966
    0.0112   0.9843   0.0155   0.9825   0.0300   0.9692   0.0269   0.9737
```

Figure 1: output for A to D

The next step was adding learnable parameters to the standard logistic activation function.

A scaling(alpha1) and a shifting(alpha2) parameters were added to the logistic activation function.

Now the activation function looks like this: $\frac{1}{1+e^{(-\alpha 1(net+\alpha 2))}}$

Figure 2 represents the strategy for updating the weights and alphas.

WEIGHT UPDATE

$$E^p = \frac{1}{2} \sum_j (t_j^p - o_j^p)^2 \qquad o_j = \frac{1}{1 + e^{-\alpha 1_j (net_j + \alpha 2_j)}} \qquad net_j = w_{ji} * o_i^p$$

WEIGHT CORRECTION: $\Delta W_{ji} = \delta_j \tilde{o}_i^p$

Output units : $\delta_j^p = (t_j - o_j) f_j'(net_j)$

Internal units : $\delta_j = \left[\sum_n \delta_n . W_{nj} \right] f_j'(net_j)$

$f_j'(net_j) = \alpha 1_j [o_j (1 - o_j)]$

$\alpha 1_j$ UPDATE (Scaling α)

Scaling correction $\rightarrow \Delta \alpha 1_j = \dfrac{\delta_j}{\alpha 1_j} [net_j + \alpha 2_j]$

Output unit : $\delta_j = (t_j - o_j) f_j'(net_j)$

Internal unit : $\delta_j = \left(\sum_n \delta_n . W_{nj} \right) [f_j'(net_j)]$

$f_j'(net_j) = \alpha 1_j [o_j (1 - o_j)]$

$\alpha 2_j$ update (Shifting α)

Shifting Correction : $\Delta \alpha 2_j = \delta_n$

Output unit $\delta_j = [t_j - o_j] [\alpha 1_j [o_j (1 - o_j)]$

Internal unit $\delta_j = \left(\sum_n \delta_n W_{nj} \right) f_j'(net_j)$

$f'(net_j) = \alpha 1_j [o_j (1 - o_j)]$

Figure 2: weights and alpha updation

The goal is to make the MLFF network learn alpha1 and alpha2 along with its weights. This way the network can slightly modify the shape and position of its squasher to better classify the target.

Initialization of alpha1 and alpha2 plays a prominent factor in faster convergence of error. Multiple ranges were selected and the error convergences for each range was recorded. Figure 3 represents the iteration vs range plot. It can be observed that the error converges to zero with minimum iterations when alphas are initialized between (1.4, 1.8).

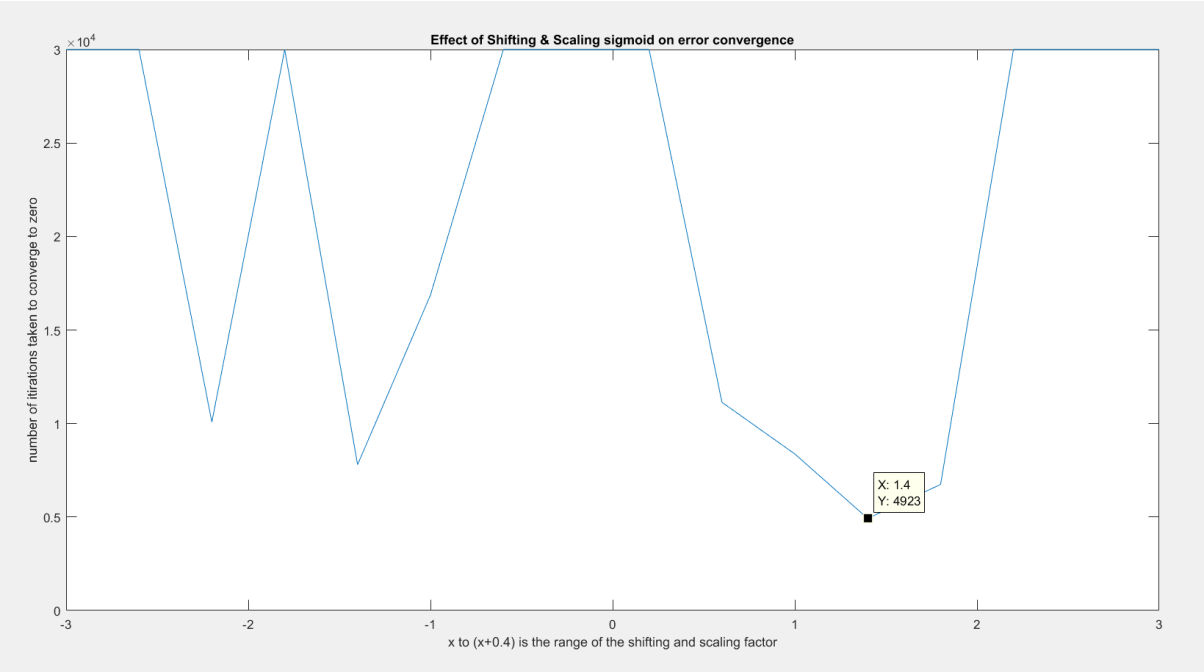

Figure 3: Error convergence graph for scaled and shifted logistic sigmoid.

Analysis

The below analysis is done primarily for the A to D dataset.

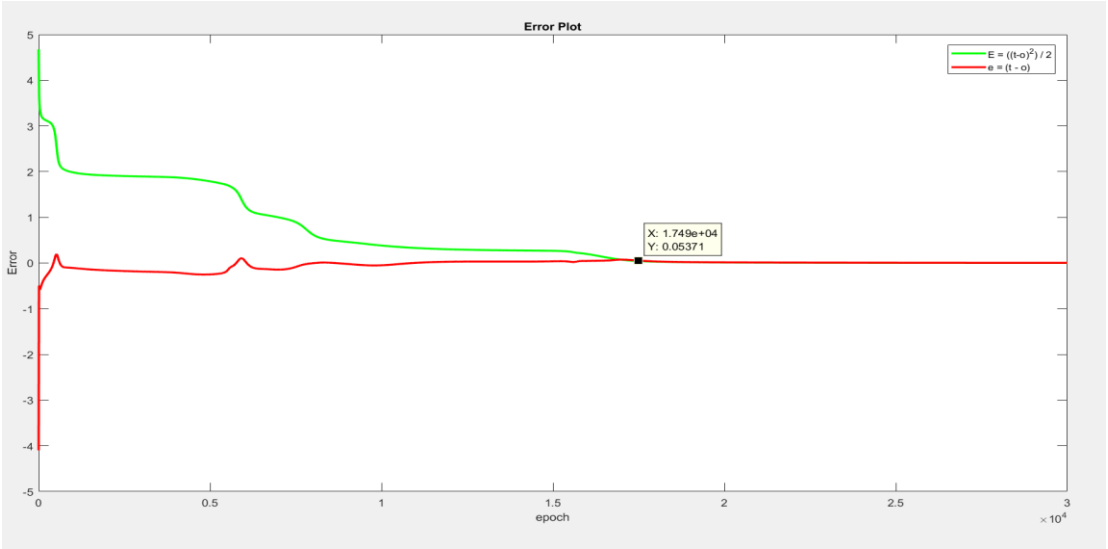

Figure 4: Error convergence plot for standard logistic Sigmoid

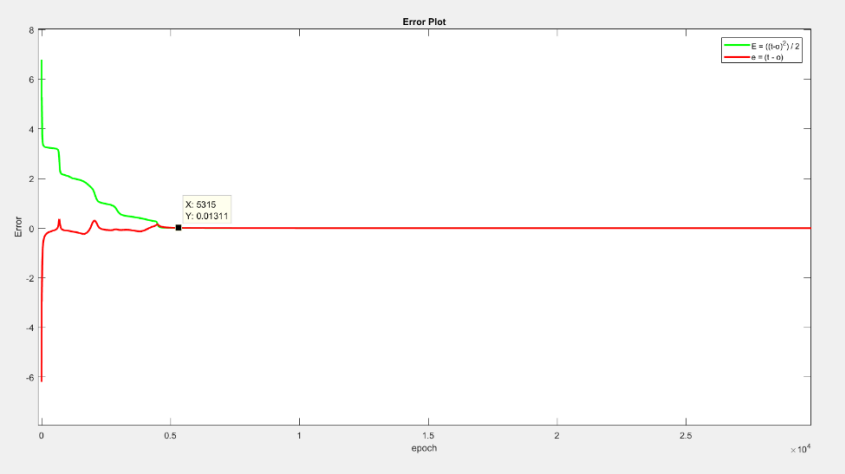

Figure 5: Error convergence plot for Shifted and scaled sigmoid

From Figure 4 and Figure 5, we can see that 17490 iterations are needed for the fixed standard logistic squasher where as the scaled and shifted squasher takes only 5315 iterations.

```
input =

    0   0.0625   0.1250   0.1875   0.2500   0.3125   0.3750   0.4375
    1.0000   1.0000   1.0000   1.0000   1.0000   1.0000   1.0000   1.0000


target =

     0     0     0     0     0     0     0     0
     0     0     0     0     1     1     1     1
     0     0     1     1     0     0     1     1
     0     1     0     1     0     1     0     1


output =

    0.0000    0.0000    0.0000    0.0000    0.0002    0.0005    0.0029    0.0042
    0.0000    0.0000    0.0000    0.0098    0.9881    0.9996    1.0000    1.0000
    0.0000    0.0038    0.9968    0.9955    0.0035    0.0034    0.9985    0.9966
    0.0112    0.9843    0.0155    0.9825    0.0300    0.9692    0.0269    0.9737
```

Figure 6: Output of MLFF with standard Logistic squasher for A to D data

```
Initial alpha_scale =

  1.7259   1.7654   1.5114   1.7860   1.7829   1.4568   1.7169   1.4143   1.6715   1.5569   1.6824   1.4185   1.6779        0        0        0
  1.7623   1.6529   1.6188   1.4630   1.5942   1.5687   1.7838   1.7397   1.7031   1.6622   1.4127   1.4389   1.5268   1.5755   1.7181   1.5782   1.7019
  1.4508   1.4390   1.7830   1.7882        0        0        0        0        0        0        0        0        0        0        0        0        0


Initial alpha_shift =

  1.6719   1.4476   1.5362   1.7005   1.6796   1.6189   1.5030   1.7257   1.5400   1.6464   1.7323   1.7669   1.7015        0        0        0        0
  1.6620   1.5993   1.6341   1.5020   1.7564   1.4554   1.7363   1.4974   1.4786   1.5893   1.6341   1.5143   1.5522   1.4216   1.7736   1.5878   1.4649
  1.4650   1.7839   1.4895   1.6024        0        0        0        0        0        0        0        0        0        0        0        0        0


input =

  input :        0   0.0625   0.1250   0.1875   0.2500   0.3125   0.3750   0.4375
  bias:     1.0000   1.0000   1.0000   1.0000   1.0000   1.0000   1.0000   1.0000


target =

     0     0     0     0     0     0     0     0
     0     0     0     0     1     1     1     1
     0     0     1     1     0     0     1     1
     0     1     0     1     0     1     0     1


output =

    0.0000    0.0000    0.0000    0.0000    0.0000    0.0002    0.0011    0.0034
    0.0000    0.0000    0.0000    0.0039    0.9956    0.9998    1.0000    1.0000
    0.0000    0.0022    0.9998    0.9967    0.0000    0.0032    0.9975    0.9993
    0.0045    0.9940    0.0053    0.9947    0.0055    0.9945    0.0063    0.9945


Final alpha_scale =

  1.7679   1.9714   1.7021   1.7920   1.7913   1.5879   1.9341   1.4633   1.8587   1.6636   1.6890   1.4276   1.6779        0        0        0        0
  1.8892   1.6734   1.6328   1.5966   1.7922   1.5819   2.1249   1.8112   1.7476   1.6896   1.7491   1.6537   1.5360   1.5891   2.1392   1.5899   1.7019
  1.5233   1.7109   1.9276   2.0086        0        0        0        0        0        0        0        0        0        0        0        0        0


Fnal alpha_shift =

  1.6438   1.4229   1.4950   1.7049   1.6596   1.5833   1.4555   1.6973   1.4963   1.6119   1.7359   1.7463   1.7015        0        0        0        0
  1.6700   1.6002   1.6341   1.5311   1.7808   1.4568   1.7600   1.5020   1.4811   1.5900   1.6740   1.5423   1.5526   1.4232   1.8171   1.5883   1.4649
  1.4148   1.8352   1.5127   1.6215        0        0        0        0        0        0        0        0        0        0        0        0        0
```

Figure 7: output of MLFF with modified squasher for A to D dataset

In Figure 7 alpha1 is represented as alpha_scale and alpha2 is represented as alpha_shift.

Observing the alpha values in Figure 7, we can see that the network makes minor adjustments to alpha1 and alpha 2 during the training process. This helps the network to converge to minimum error faster.

Note: The learning rates for the alpha update was set to 0.005 and the learning rates for the weight update was set to 0.5.

From the above observations, I conclude that the network converges to zero faster when learnable parameters are added to the fixed logistic sigmoid.

Summary

A generic MLFF network was developed with logistic sigmoid as the activation function.
Learnable parameters were added to the activation function.
It was observed that the network converged to zero much faster with the shifting and scaling parameters.
These parameters were learnt by the network.

Future work

1. Study the effects of varying the number of layers.
2. Study the effects of different learning rates for the shifting and scaling parameters.
3. Application of learnable parameters for other activation functions like the ReLU.

APPENDIX

Feedforward

```
function [net,y] = Feedforeward( input,dim,layers,W,y,net,alpha_scale,alpha_shift)


   for i=1:layers-1

      %input neurons
      cur_cnt = dim(i)+1; %neurons in current layer +1 bias

      %output neurons
      nxt_cnt = dim(i+1); %neurons in next layer

      %to section out weight matrix
      si = 1; %starting index
      ei = cur_cnt;   %ending index

      for m = 1:nxt_cnt

         %weight segment selection
         ws = W(i, si:ei);

         %input segment selection
         if(i == 1)
            ips = input;
         else
            ips = y(i-1, 1:cur_cnt)';
         end

         %selecting alpha
         alpha_scale_i = alpha_scale(i,m); %selecting corrosponding alpha_scale
         alpha_shift_i = alpha_shift(i,m); %selecting corrosponding alpha_shift


         %calculate output and net activation
         [ net(i,m) , y(i,m) ] = neuron(ws,ips,alpha_scale_i,alpha_shift_i);

         %slide indices to select next set of weights
         si = si + cur_cnt;
         ei = ei + cur_cnt;

      end % m forloop
   end %i forloop end


end %function end
```

Backpropogation

```
function [dw,dalpha_scale,dalpha_shift,error,err_single] =
backpropogation(ip,target,dim,layers,W,y,dy,j,dw,a,net,da,alpha_scale,dalpha_scale,ds,alpha_shift,dalpha_shift)

   dim_bias = dim_bias_calc(dim); %dim with bias units

   for i = layers:-1:2

      cur_cnt = dim(i); %neurons count in current layer without bias

      if(i ~= layers) %there is no next layer for output layer
         %bias is not considered as neuron,thus dim_bias(i+1) not
         %used here
         nxt_cnt = dim(i+1); %neuron count in previous layer
      end
      %starting and ending indiecs
         si = 1;
         ei = cur_cnt; %current count without bias
         eib = dim_bias(i); %current count with bias
         %no bias in final layer. thus neuron count doesnt include bias
         if( i == layers)
```

```
            eib = ei;
        end

    %%calculating dy(sensitivity)
    for m = 1:eib %current layer

        %sigmoid derivative wrt W and alpha respectively
        %fnet are single values.
        fnet = alpha_scale(i-1,m) * ( y(i-1,m) * (1 - y(i-1,m)) );
%           fnet_delta = ( y(i-1,m) * (1 - y(i-1,m)) );

        if( i == layers ) %output layer

            %output layer
            err = target(j,m) - y(i-1,m); %single value

            %for plotting error ( sum of errors of all o/p neurons)
            if( m == 1)
                error = (err*err)/2;
                err_single = err;
            else
                error = error + (err*err)/2;
                err_single = err_single + err;
            end

            %sensitivity for weights updation
            dy(i-1,m) = fnet * err; %normal single value multiplication

            %da = sensitivity for alpha1(scaling)
            da(i-1,m) = fnet * err;

            %da = sensitivity for alpha2(shifting)
            ds(i-1,m) = fnet * err;

        else %hidden layers

            si = m;
            for k = 1:nxt_cnt
                Wseg(:,k) = W(i,si);
                si = si + eib;
            end %k for

            %weight sensitivity
            %inner product of all next layersensitivity with weights
            %connecting them to current neuron
            err_hidden  = Wseg * dy(i,1:nxt_cnt)' ;
            dy(i-1,m) = fnet * err_hidden ;

            %no need of delta function. this is just direct
            %multiplication of fnet and err_hidden

            %same procedure as above
            %alpha1(scaling) calcuations
            err_hidden  = Wseg * da(i,1:nxt_cnt)' ;
            da(i-1,m) = fnet * err_hidden ;

            %alpha2(shifting) calcuations
            err_hidden  = Wseg * ds(i,1:nxt_cnt)' ;
            ds(i-1,m) = fnet * err_hidden ;

        end %if
    end %m for
end% i for

%%calculating dw and dalpha
for i = 1:layers-1
    cur_cnt = dim_bias(i);%to select input/previous neuron output
    nxt_cnt = dim(i+1); %bias in not considered as a neuron.
    si = 1;
    ei = cur_cnt;
    for m = 1:nxt_cnt %m = 1 points at y11
        if( i == 1)
            opt = ip; %input layer
        else
            %all outputs of hidden layer/inputs to next layer
```

```
            opt = y(i-1,1:cur_cnt);
        end %if end

        dyseg = dy(i,m); %individual neurons in next layer

        %weight delta matrix

        %no need of weight_corr(dyseg,opt) function below.
        %direct multiplication is sufficient
        dw(i,si:ei) = opt * dyseg;

        %alpha1(scaling) delta matrix
        mul_factor = ( ( net(i,m) + alpha_shift(i,m) ) / alpha_scale(i,m) );
        dalpha_scale(i,m) = da(i,m) * mul_factor ;

        %alpha2(shifting) delta matrix
        dalpha_shift(i,m) = ds(i,m);


        %update starting and ending index
        si = si+cur_cnt;
        ei = ei+cur_cnt;

      end %m loop
   end %i loop

end %function
```

Training

```
function [W,alpha_scale,alpha_shift] = train(a,ep,layers,input,target,dim,y,net,a1_lrn, alpha_scale,alpha_shift,a2_lrn)

   dim_bias = dim_bias_calc(dim);
   %dimention weights are taken into account in weight_count function!

   %to find the total number of weights
   weight_cnt = 0;
   for i = 1:(layers-1)
      weight_cnt = weight_cnt + (dim(i)*dim(i+1));
   end


   %initializing weight matrix
   max_weight = weight_count(dim,layers);

   %weight matrix W
%     W = 0.1*randn( layers-1, max_weight );

   W = 0.01*rand(layers-1, max_weight); %A to D

   %code to set unwanted weights to 0
   W = set_zero(dim,W,layers,max_weight);

   %feedforeward and backprop
   pattern = length(target); %number of pattern

   %initializing a vector to plot error
   e_plot = zeros(1,ep);
   error_single_plot = zeros(1,ep);


   si = 1;
   ei = pattern;


   for i = 1:ep

      dw = zeros( layers-1, max_weight );% initialize weight update matrix
      dy = zeros(layers-1, max(dim_bias)); %initialize delta matrix(sensetivity)
      da = zeros(layers-1, max(dim_bias)); %initialize sensitivity for alpha scale
      ds = zeros(layers-1, max(dim_bias)); %initialize sensitivity for alpha shift
      dalpha_scale = zeros(layers-1, max(dim_bias)); %initialize delta(alpha scale) matrix
      dalpha_shift = zeros(layers-1, max(dim_bias)); %initialize delta(alpha shift) matrix
     error_pat = 0;
```

```
        error_single_pat = 0;
         for j = 1:pattern

            ip = input(:,j); %each pattern of input
            [net,y] =  Feedforeward( ip,dim,layers,W,y,net,alpha_scale,alpha_shift );
            [dw,dalpha_scale,dalpha_shift,error,err_single] = backpropogation(
ip,target,dim,layers,W,y,dy,j,dw,a,net,da,alpha_scale,dalpha_scale,ds,alpha_shift,dalpha_shift );

            %sum of errors of all pattern
            error_pat = error_pat + error;
            error_single_pat = error_single_pat + err_single;

            %on-line update
            W = W + a*(dw);
            alpha_scale = alpha_scale + a1_lrn * ( dalpha_scale );
            alpha_shift = alpha_shift + a2_lrn * ( dalpha_shift );


        end % pattern
        %final error for each epoch
        error_plot(1,i) = error_pat;
        error_single_plot(1,i) = error_single_pat;

    end % epoch

    %plotting error
    X = 1:ep;
    Y = error_plot;
    Z = error_single_plot;

    figure
    plot(X,Y,'g','LineWidth',2);
    hold on;
    plot(X,Z,'r','LineWidth',2);
    xlabel("epoch");
    ylabel("Error");
    title("Error Plot");
    legend("E = ((t-o)^2) / 2 ","e = (t - o)");


    First = [ 'First error = ', num2str(error_plot(1,1)) ];
    Final = [ 'Final error = ', num2str(error_plot(1,end)) ];
    disp(First);
    disp(Final);



end %function
```