OKLAHOMA STATE UNIVERSITY

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

NEURAL NETWORK-ECEN 5733

Reducing the Dimensionality of Data with Neural Networks

Author:
Neda NOURSHAMSI

Supervisor:
Dr. Martin HAGAN

May 8, 2016

Contents

List of Figures

1 Introduction

Dimension reduction can be a useful way to visualize and save high dimensional data and make it more simple. One way to reduce dimensionality is using Principal Components Analysis (PCA). PCA is one of the simplest methods to reduce the dimension. This method finds the directions along which the data has maximum variance in addition to the relative importance of these directions [2]. There have been several studies which shows a multilayer neural network can be used to convert high-dimensional codes to low dimensional codes. In this report, we tried to understand and implement the paper "Reducing the Dimensionality of Data with Neural Network" by G. E. Hinton and R. R. Salakhutdinov [1]. The authors tried to explain a non linear generalization of PCA which uses a multilayered encoder network to convert high dimensional data to low dimension and the same decoder to regain the data. The required initial weights and biases are reached by applying a new algorithm.

2 Paper Review

Autoencoders are neural networks trained to reconstruct their original input. It has two parts: encoder and decoder. In encoder, we reduce the elements and in decoder we reach to the same original elements. The autoencoder needs random weights and biases in both parts (encoder and decoder) and the network tries to minimize the difference between original data and its reconstruction. The autoencoder works based on the back propagation algorithm which has the forward , sensitivities, weights and biases and all depends on the number of layers. However, it is not easy to optimize the initial weights in a nonlinear autoencoder with several hidden layers [2]. If we use large initial weights, the autoencoder finds the poor local minimum and by applying small initial weights, we will not be able to train the autoencoder. Therefore, the authors presented a "pretraining" procedure to find the initial values. In the pretraining procedure, the ensemble binary

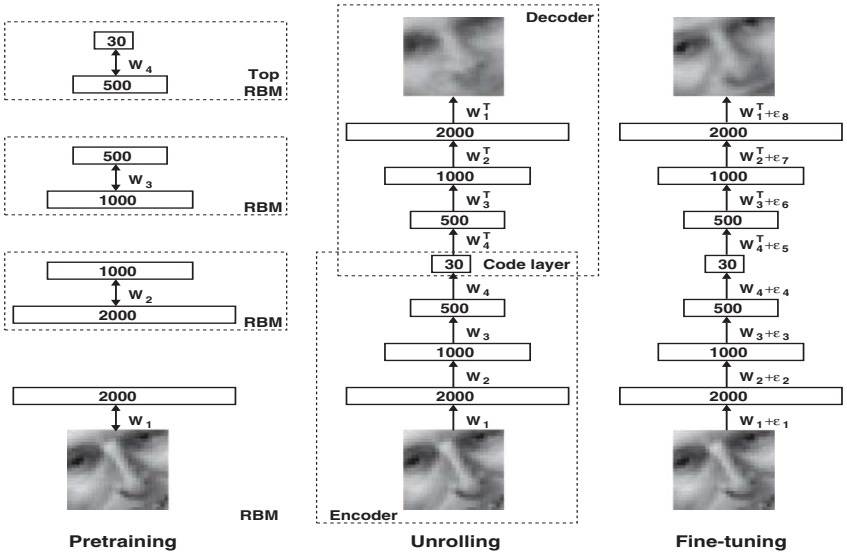


Figure 1: Pretraining and autoencoder explanation [1]

vectors can be modeled by a two layer network which is named Restricted Boltzman Machine (RBM). This network connects the stochastic binary pixels to the stochastic binary features by using symmetrical weight connections. Fig. 1 shows the pretraining and autoencoder to give us a general idea.

We consider an autoencoder with 400 - 200 - 100 - 200 - 400 layers. The layers 200 - 100 - 200 have the linear transfer function. At the beginning, we decided to train the network (autoencoder) with random initial weights. Figs. 2, 3 and 4 show the inputs and results of the autoencoder. As we can see, the autoencoder can not have the same input shape and SSE can't go down. Therefore, we need to change the initial weights and biases.

Therefore, the authors proposed a two layer RBM network to find these initial values. Hinton in 2002 [3] proposed a learning procedure to train the RBM which is called Contrastive Divergence (CD) [4]. The architect of this algorithm is shown in Fig. 5. In the RBM, we will have two layers: visible and hidden. The weights of the first RBM were initialized with small random values, sampled form normal distribution with zero mean and standard deviation of 0.1 [5]. In the first step to

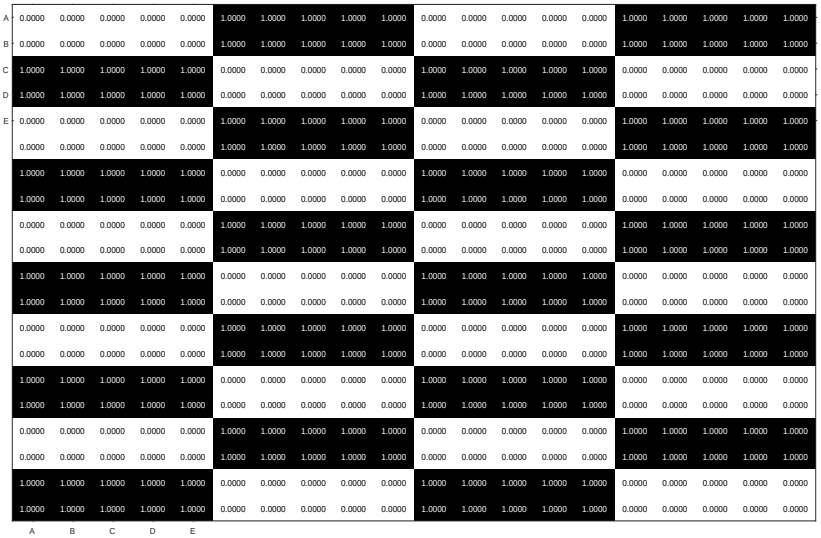


Figure 2: Input without using RBM

Figure 3: Output without using RBM

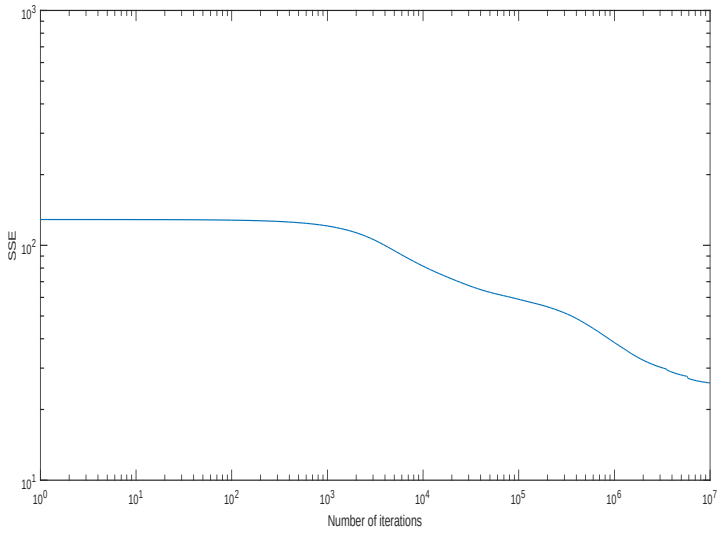


Figure 4: SSE without using RBM

train RBM, we will compute the output of network which is showing the hidden values (forward) by using the inputs (visible values) see Eq. 1. The Eq. 1 and the input can help us to calculate out_1 see Eq. 2. In the next step, we are using the output of previous step as the input (hidden values) and calculating the output (visible values) see Eq. 3. In the final step, we use the output of Eq. 3 (visible values) as the input. Then, we calculate the final output (reconstruction) see Eq. 4. Then, we calculate out_2 see Eq. 5. The weights and biases are updated as explained in Eqs. 6, 7 and 8. All of RBMs can use the same algorithm. But, the top one which has the real values in its hidden layers instead of binary values. [1]. Figs. 5 and 6 show the architecture for one RPM and the structure of RBMs for more details.

We tried to solve a simple problem. The autoencoder with following layers were chosen: 400 - 200 - 100 - 200 - 400. This autoencoder needs two RBM network.

Each RPM includes two parts: forward and backward. The equation for forward and backward are explained in Eqs.1, 2, 3, 4 and 5 .

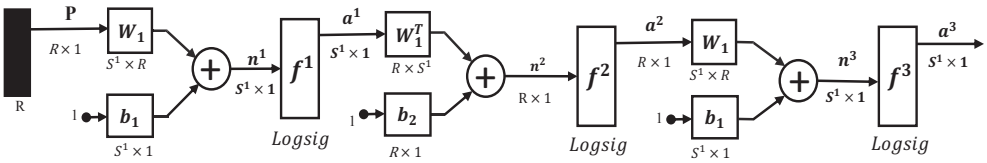


Figure 5: RBM Architecture by using CD

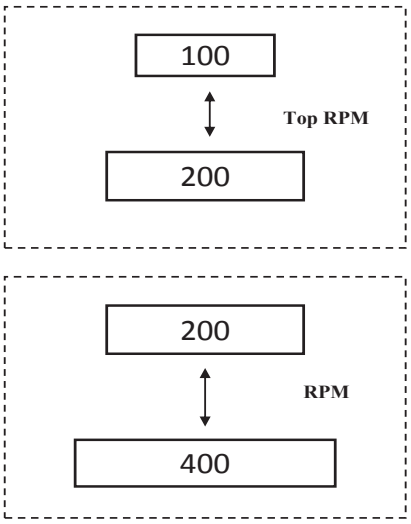


Figure 6: RPM layers

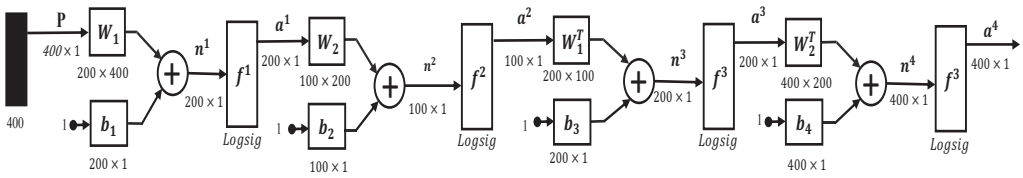


Figure 7: Autoencoder architecture

$$\mathbf{a}_1 = logsig(\mathbf{W} \times \mathbf{p} + \mathbf{b}_1) \tag{1}$$

$$\mathbf{out}_1 = (\mathbf{a}_1 \times \mathbf{p}^T) \tag{2}$$

$$\mathbf{a}_2 = logsig(\mathbf{W}^T \times \mathbf{a}_1 + \mathbf{b}_2) \tag{3}$$

$$\mathbf{a}_3 = logsig(\mathbf{W} \times \mathbf{a}_1 + \mathbf{b}_1) \tag{4}$$

$$\mathbf{out}_2 = (\mathbf{a}_3 \times \mathbf{a}_2^T) \tag{5}$$

Weights and biases can be updated as explained in Eqs. 6, 7 and 8

$$\mathbf{W}(k+1) = \mathbf{W}(k) + \frac{\alpha}{N}(\mathbf{out}_1 - \mathbf{out}_2) \tag{6}$$

$$\mathbf{b}_1(k+1) = \mathbf{b}_1(k) + \frac{\alpha}{N}\sum_{n=1}^{N}(\mathbf{a}_1 - \mathbf{a}_3) \tag{7}$$

$$\mathbf{b}_2(k+1) = \mathbf{b}_2(k) + \frac{\alpha}{N}\sum_{n=1}^{N}(\mathbf{p} - \mathbf{a}_2) \tag{8}$$

where n shows the number of columns.

The autoencoder architecture and structure are shown in Fig. 7 and Fig. 8.

The forward equations of autoencoder are shown in Eq. 9 and 10.

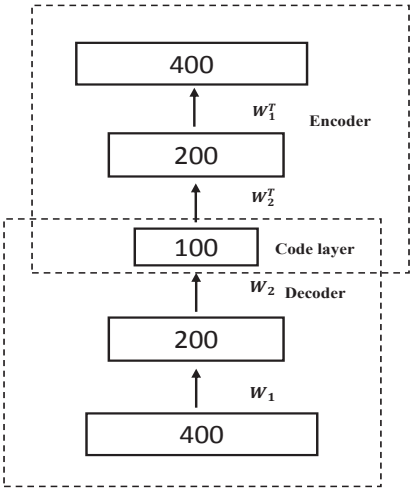


Figure 8: Autoencoder

$$\mathbf{a}^0 = \mathbf{p} \tag{9}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1} \times (\mathbf{a}^{m+1} + \mathbf{b}^{m+1})) \tag{10}$$

Where $m = 0, 1, ...M - 1$.

The first and last layer have the log-sigmoid transfer function and the second and third layers have the linear transfer functions.

Sensitivities equations of autoencoder are shown in Eqs. 11 and 12. The gradient is calculated by applying Steepest decent.

$$\mathbf{s}^M = -2 \times \dot{\mathbf{F}}^M \times (\mathbf{t} - \mathbf{a}) \tag{11}$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m \times ((\mathbf{W}^{m+1})^T \times \mathbf{s}^{m+1}) \tag{12}$$

Where $m = M - 1, ..., 0$ and M is final layer. Updated weight and biases equations are shown in Eqs. 13, 14:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}(\mathbf{s}_q)^m \times ((\mathbf{a}_q)^{m-1})^T \tag{13}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}(\mathbf{s}_q)^m \tag{14}$$

3 Program Description

The autoencoder has the following layers: 400 - 200 - 100 - 200 - 400. The proportional following code shows the first RBM.

```
function [w_1,b_1,b_2,a_20,a_21] = CD11(w_1,p_1,b_1,b_2)
alpha = 0.001;
itr=50;
for i=1:itr
            % FIRST-RPM-FW:
        a_0   = logsig(w_1 * p_1 + b_1*ones(1,2));
        a_00  = sum(a_0,2);
        a_1   = a_0>rand(200,2);
        out_1 = a_1*p_1';
        p_11  = sum(p_1,2);

        % FIRST-RPM-BCK:
        a_2   = logsig(w_1'* a_1 + b_2*ones(1,2));
        a_21  = logsig(w_1*  a_2 + b_1*ones(1,2));
        a_20  = sum(a_21,2);
        out_2 = a_21*a_2' ;
        a_22  = sum(a_2,2);

        %UPDATE WIGHT AND BIASES

        w_1  = w_1 + (alpha)*(out_1 - out_2);
```

```
        b_1   = b_1 + (alpha)*(a_00   - a_20);
        b_2   = b_2 + (alpha)*(p_11   - a_22);

  end
```

The proportional following code shows the top RBM which will give the initial weights and biases for the autoencoder.

```
function [w_2,b_3,b_4,a_41] = CD12(w_2,a_20,a_21,b_3,b_4)
alpha = 0.001;
itr=50;
for i=1:itr
     % RPM-FW:
    a_3    = logsig(w_2 * a_21 + b_3*ones(1,2));
    a_30   = sum(a_3,2);
    out_1 = a_3*a_21';

    % RPM-BCK:
    a_4    = logsig(w_2'* a_3 + b_4*ones(1,2));
    a_41   = logsig(w_2*a_4 + b_3*ones(1,2));
    a_42=sum(a_41,2);
    a_43=sum(a_4,2);
    out_2 = a_41*a_4' ;

    %UPDATE WIGHT AND BIASES

    w_2   = w_2 + (alpha)*(out_1-out_2);
    b_3 = b_3 + (alpha)*(a_30 - a_42);
    b_4 = b_4 + (alpha)*(a_20-a_43);


end
```

The following code is the back propagation algorithm for the autoencoder.

```
1
2  % random values for weights
3  w_1 = randn(200,400) * 0.1;
4  % random values for biases
5  b_1 = rand(200,1) - 0.5;
6  b_2 = rand(400,1) - 0.5;
7
8  % random values for weights
9  w_2 = randn(100,200) * 0.1;
10 % random values for biases
11 b_3 = rand(100,1) - 0.5;
12 b_4 = rand(200,1) - 0.5;
13 [R,Q]=size(p_1);
14
15 [w_1,b_1,b_2,a_20,a_21] = CD11(w_1,p_1,b_1,b_2);
16 [w_2,b_3,b_4,a_41] = CD12(w_2,a_20,a_21,b_3,b_4);
17
18 iteration=1e7;
19 beta=0.001;
20
21 for i=1:iteration
22 %FORWARD
23
24 w_3 = w_2';
25 w_4 = w_1';
26 a_10=logsig(w_1*p_1 + b_1*ones(1,2));
27 a_20=(w_2*a_10 + b_3*ones(1,2));
28 a_30=(w_3*a_20 + b_4*ones(1,2));
29 a_40=logsig(w_4*a_30 + b_2*ones(1,2));
30 e = p_1 - a_40;
31
32 %SENSITIVITIES
```

```
1
2  Fdot4=a_40.*(1-a_40);
3  Fdot1=a_10.*(1-a_10);
4  s_4=-2.*Fdot4.*(e);
5  s_3=w_1*s_4;
6  s_2=w_2*s_3;
7  s_1=Fdot1.*(w_2'*s_2);
8
9  %SUMMES
10
11 sum1_1=(s_1*p_1');
12 sum1_2=sum(s_1,2);
13 sum2_1=(s_2*a_10');
14 sum2_2=sum(s_2,2);
15 sum3_1 = (s_3*a_20');
16 sum3_2 = sum(s_3,2);
17 sum4_1 = (s_4*a_30');
18 sum4_2 = sum(s_4 , 2);
19 w_1=w_1 - (beta*sum1_1)/Q;
20 b_1=b_1 - (beta*sum1_2)/Q;
21 w_2=w_2 - (beta*sum2_1)/Q;
22 b_3=b_3 - (beta*sum2_2)/Q;
23 w_3 = w_3 - (beta*sum3_1)/Q;
24 b_4 = b_4 - (beta*sum3_2)/Q;
25 w_4 = w_4 - (beta*sum4_1)/Q;
26 b_2 = b_2 - (beta*sum4_2)/Q;
27
28 sum_e(i)=trace(e'*e);
29 end
30
31 figure;                        %shows SSE
32 loglog(sum_e)
```

We used three different inputs separately in the first step and two of them at the same time at the second step to see the results of training.

4 Result

We used three different inputs and trained each of them individually to see SSE and the results of output. It can be seen that the network converges for each of them. The Figs. 9, 10 and 11 show the input, output and SSE of rectangular shape.

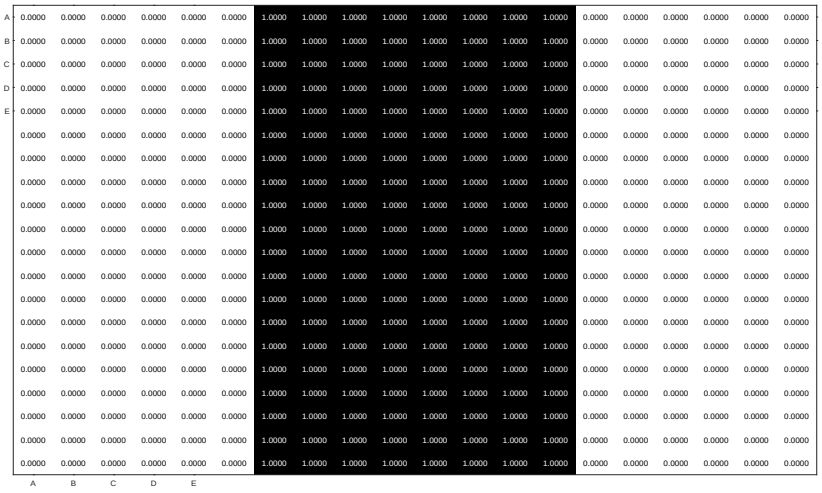
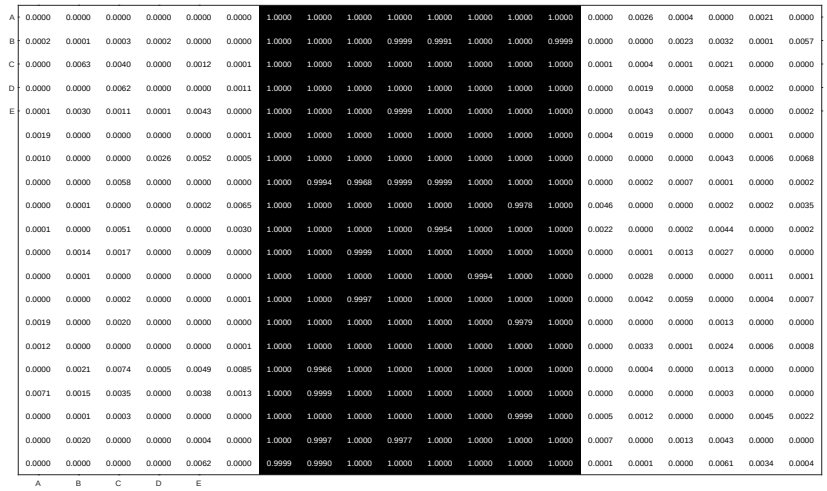


Figure 9: Input of rectangular shape by using RBM



	A	B	C	D	E
A	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0026	0.0004	0.0000	0.0021	0.0000
B	0.0002	0.0001	0.0003	0.0002	0.0000	0.0000	1.0000	1.0000	1.0000	0.9999	0.9991	1.0000	1.0000	0.9999	0.0000	0.0000	0.0023	0.0032	0.0001	0.0057
C	0.0000	0.0063	0.0040	0.0000	0.0012	0.0001	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0001	0.0004	0.0001	0.0021	0.0000	0.0000
D	0.0000	0.0000	0.0062	0.0000	0.0000	0.0011	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0019	0.0000	0.0058	0.0002	0.0000
E	0.0001	0.0030	0.0011	0.0001	0.0043	0.0000	1.0000	1.0000	1.0000	0.9999	1.0000	1.0000	1.0000	1.0000	0.0000	0.0043	0.0007	0.0043	0.0000	0.0002
	0.0019	0.0000	0.0000	0.0000	0.0000	0.0001	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0004	0.0019	0.0000	0.0000	0.0001	0.0000
	0.0010	0.0000	0.0000	0.0026	0.0052	0.0005	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0043	0.0006	0.0068
	0.0000	0.0000	0.0058	0.0000	0.0000	0.0000	1.0000	0.9994	0.9968	0.9999	0.9999	1.0000	1.0000	1.0000	0.0000	0.0002	0.0007	0.0001	0.0000	0.0002
	0.0000	0.0001	0.0000	0.0000	0.0002	0.0065	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9978	1.0000	0.0046	0.0000	0.0000	0.0002	0.0002	0.0035
	0.0001	0.0000	0.0051	0.0000	0.0000	0.0030	1.0000	1.0000	1.0000	1.0000	0.9954	1.0000	1.0000	1.0000	0.0022	0.0000	0.0002	0.0044	0.0000	0.0002
	0.0000	0.0014	0.0017	0.0000	0.0009	0.0000	1.0000	1.0000	0.9999	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0001	0.0013	0.0027	0.0000	0.0000
	0.0000	0.0001	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9994	1.0000	1.0000	0.0000	0.0028	0.0000	0.0000	0.0011	0.0001
	0.0000	0.0000	0.0002	0.0000	0.0000	0.0001	1.0000	1.0000	0.9997	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0042	0.0059	0.0000	0.0004	0.0007
	0.0019	0.0000	0.0020	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9979	1.0000	0.0000	0.0000	0.0000	0.0013	0.0000	0.0000
	0.0012	0.0000	0.0000	0.0000	0.0000	0.0001	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0033	0.0001	0.0024	0.0006	0.0008
	0.0000	0.0021	0.0074	0.0005	0.0049	0.0085	1.0000	0.9966	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0004	0.0000	0.0013	0.0000	0.0000
	0.0071	0.0015	0.0035	0.0000	0.0038	0.0013	1.0000	0.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0003	0.0000	0.0000
	0.0000	0.0001	0.0003	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999	1.0000	0.0005	0.0012	0.0000	0.0000	0.0045	0.0022
	0.0000	0.0020	0.0000	0.0000	0.0004	0.0000	1.0000	0.9997	1.0000	0.9977	1.0000	1.0000	1.0000	1.0000	0.0007	0.0000	0.0013	0.0043	0.0000	0.0000
	0.0000	0.0000	0.0000	0.0000	0.0062	0.0000	0.9999	0.9990	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0001	0.0001	0.0000	0.0061	0.0034	0.0004

Figure 10: Output of rectangular shape by using RBM

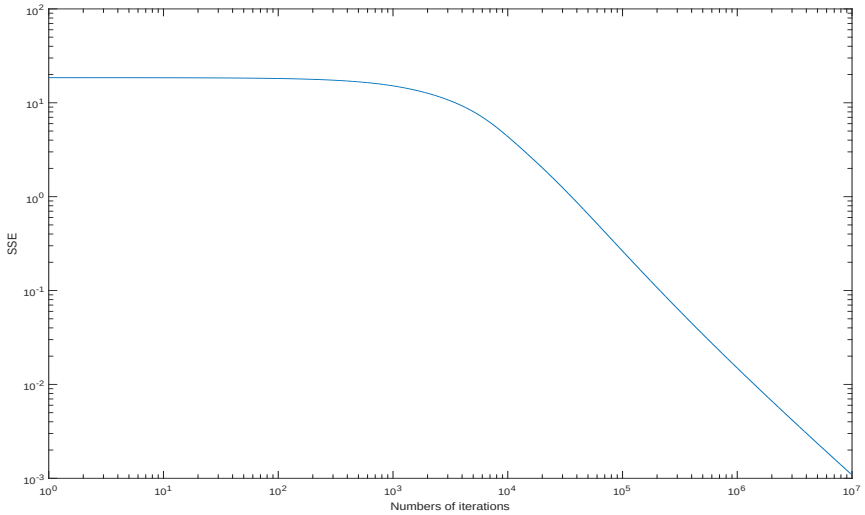


Figure 11: SSE

The Figs. 12, 13 and 14 show the input, output and SSE of triangle shape.

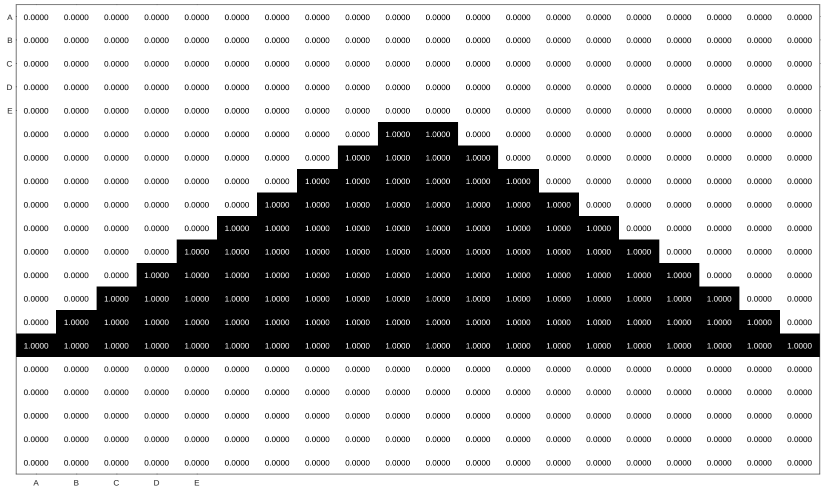


Figure 12: Input of triangle shape by using RBM

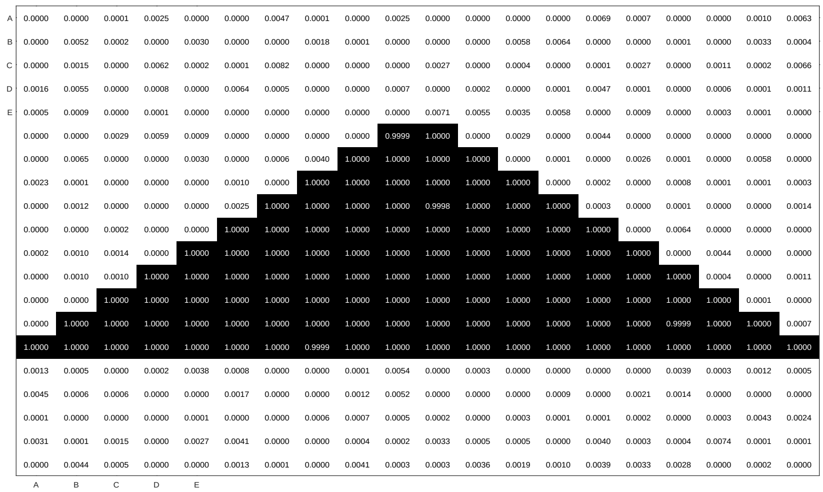


Figure 13: Output of triangle shape by using RBM

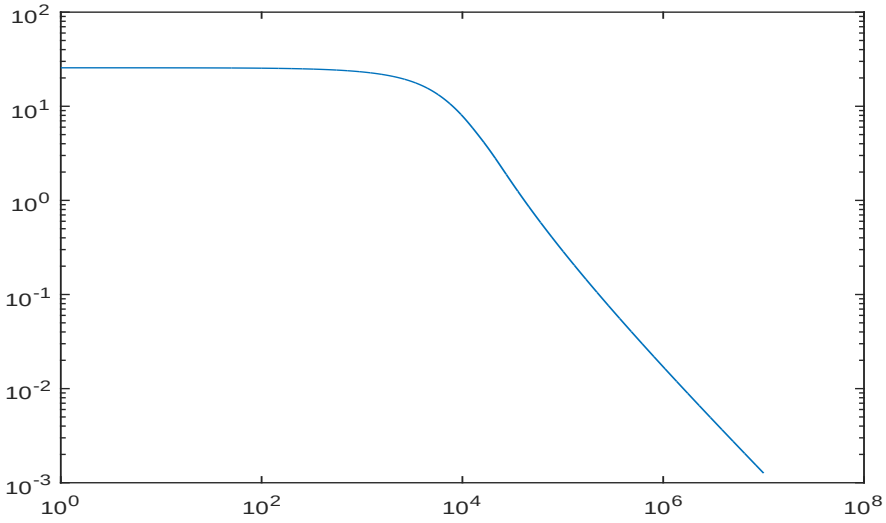


Figure 14: SSE

The Figs. 15, 16 and 17 shows the input, output and SSE of checkered shape.

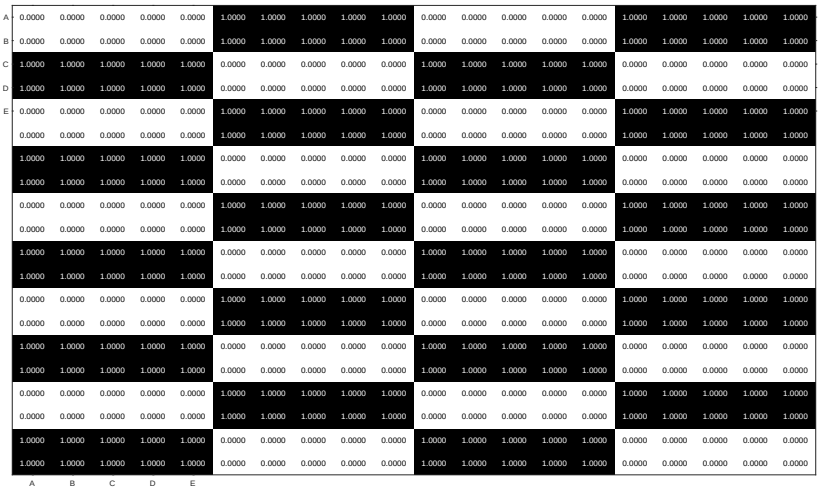


Figure 15: Input of checkered shape by using RBM

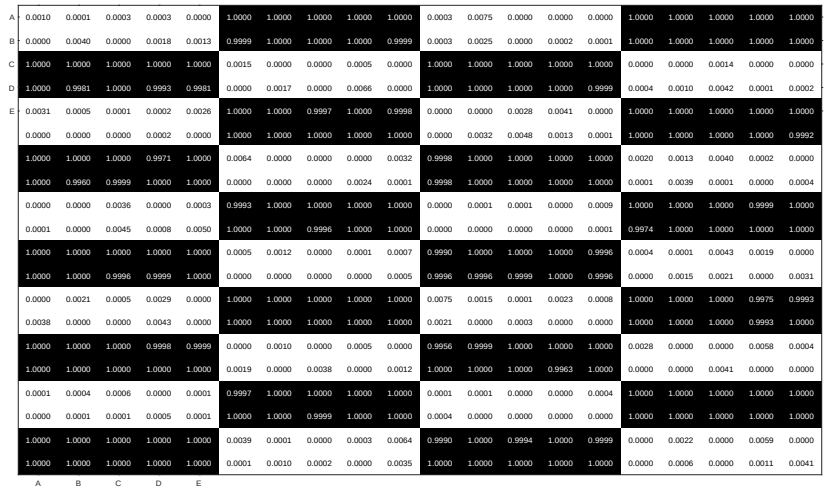


Figure 16: Output of checkered shape by using RBM

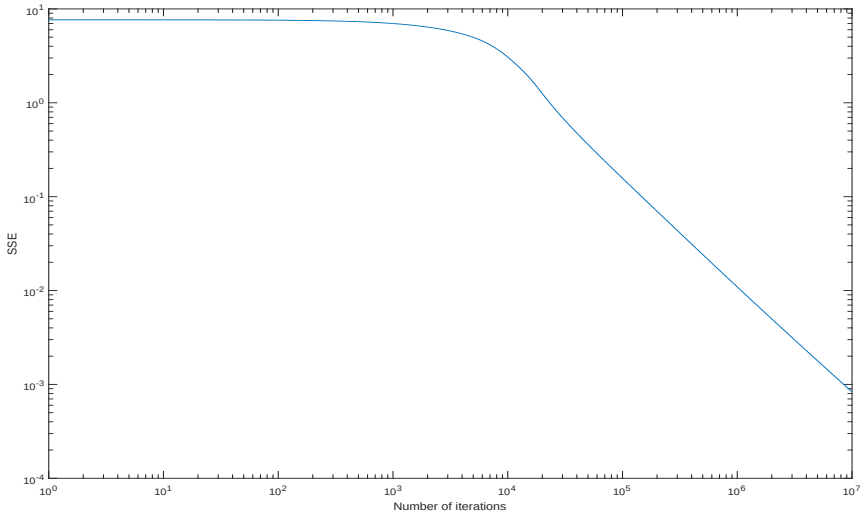


Figure 17: SSE

In the final step, we trained two inputs in Figs. 9 and 15 at the same time to

see the results. The Figs. 18, 19 and 20 shows the outputs and SSE of checkered and rectangular shapes.

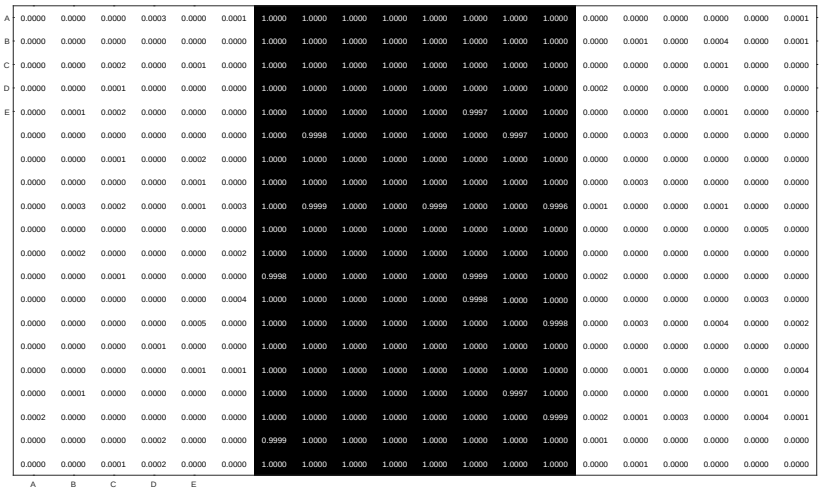
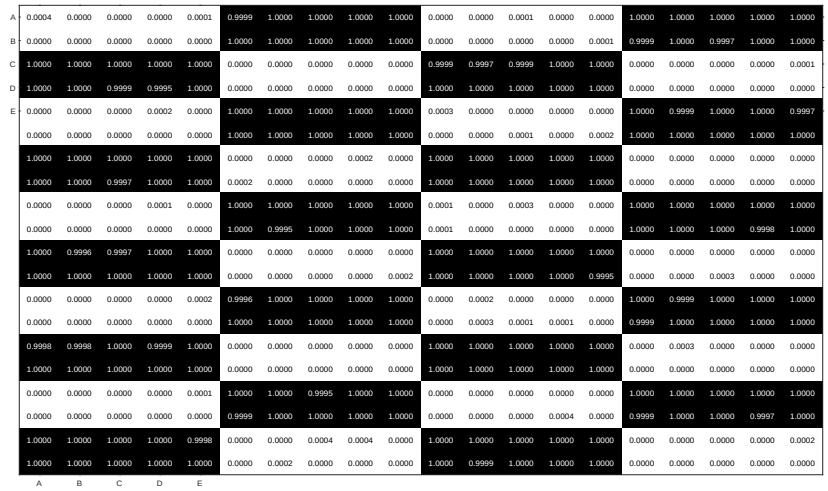


Figure 18: First output

It can be clearly seen that the autoencoder works well for two inputs at the same time by applying RBM network for finding the appropriate initial weights and biases. The two outputs are almost the same as their inputs and SSE is minimized.



	A	B	C	D	E															
A	0.0004	0.0000	0.0000	0.0000	0.0001	0.9999	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0001	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000
B	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.9999	1.0000	0.9997	1.0000	1.0000
C	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.9999	0.9997	0.9999	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0001
D	1.0000	1.0000	0.9999	0.9995	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
E	0.0000	0.0000	0.0000	0.0002	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0003	0.0000	0.0000	0.0000	0.0000	1.0000	0.9999	1.0000	1.0000	0.9997
	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0001	0.0000	0.0002	1.0000	1.0000	1.0000	1.0000	1.0000
	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0002	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	1.0000	1.0000	0.9997	1.0000	1.0000	0.0002	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	0.0000	0.0000	0.0000	0.0001	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0001	0.0000	0.0003	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	0.9995	1.0000	1.0000	1.0000	0.0001	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	0.9998	1.0000
	1.0000	0.9996	0.9997	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0002	1.0000	1.0000	1.0000	1.0000	0.9995	0.0000	0.0000	0.0003	0.0000	0.0000
	0.0000	0.0000	0.0000	0.0000	0.0002	0.9996	1.0000	1.0000	1.0000	1.0000	0.0000	0.0002	0.0000	0.0000	0.0000	1.0000	0.9999	1.0000	1.0000	1.0000
	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0003	0.0001	0.0001	0.0000	0.9999	1.0000	1.0000	1.0000	1.0000
	0.9998	0.9998	1.0000	0.9999	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0003	0.0000	0.0000	0.0000
	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	0.0000	0.0000	0.0000	0.0000	0.0001	1.0000	1.0000	0.9995	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	0.0000	0.0000	0.0000	0.0000	0.0000	0.9999	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0004	0.0000	0.9999	1.0000	1.0000	0.9997	1.0000
	1.0000	1.0000	1.0000	1.0000	0.9998	0.0000	0.0000	0.0004	0.0004	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0002
	1.0000	1.0000	1.0000	1.0000	1.0000	0.0000	0.0002	0.0000	0.0000	0.0000	1.0000	0.9999	1.0000	1.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Figure 19: Second output

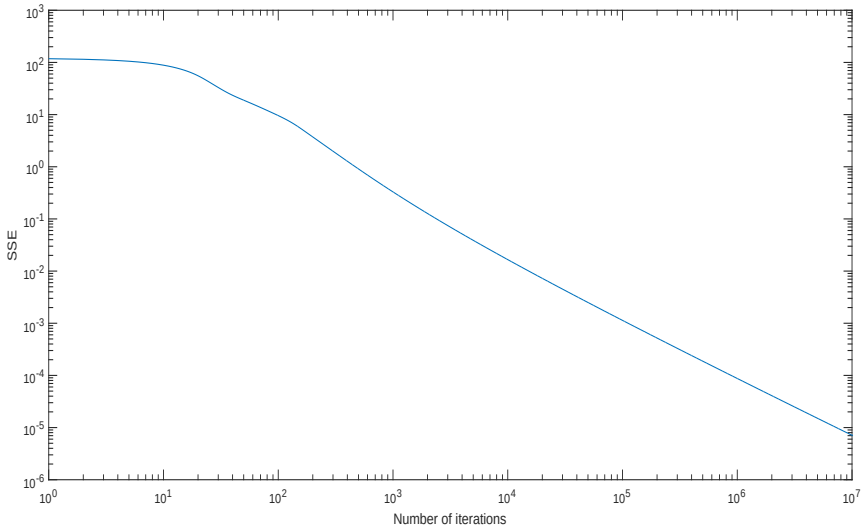


Figure 20: SSE for combination of two vectors

5 Conclusion

The encoder without the appropriate initials for weights and biases can't find the correct result. Ther pretraining procedure which used RBM can be a solution for this problem. The RBM can give us the the sufficient initial weights and biases which can be used in the autoencoder. In this report, we tested the network with binary pictures from chapter7 in [6]. We suggest to try a real picture with three layers with the code to see the results. In addition, the iteration number in each RBM layers is considered 50 based on support material of the paper [5]. We suggested to try RBM network with different iteration number to see the effects of the numbers as well.

A Matlab Code

```
close all
clc
clear

%%%%%%%%%%termproject%%%%%%%%%%%%
%%%%%%%%%%%%Fisrt Input %%%%%%%%%%%%%
p_10=[ zeros(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,9),ones(1,2),zeros(1,9);
    zeros(1,8),ones(1,4),zeros(1,8);
    zeros(1,7),ones(1,6),zeros(1,7);
    zeros(1,6),ones(1,8),zeros(1,6);
    zeros(1,5),ones(1,10),zeros(1,5);
    zeros(1,4),ones(1,12),zeros(1,4);
    zeros(1,3),ones(1,14),zeros(1,3);
```

```
    zeros(1,2),ones(1,16),zeros(1,2);
    zeros(1,1),ones(1,18),zeros(1,1);
    ones(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,20);
    zeros(1,20)];


%%%%%%%%%%%%Second Input%%%%%%%%%%%
p_20=[zeros(20,6),ones(20,8),zeros(20,6)];  %This is ...
   arbitarary input

%%%%%%%%%%%%%%%%%%%%Third input%%%%%%%%%%%%%%%%%%%%%%%%%%
p_30=[ zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    zeros(1,5),ones(1,5),zeros(1,5),ones(1,5);
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5);
```

```
    ones(1,5),zeros(1,5),ones(1,5),zeros(1,5)];


%make matrix like a vector
p_2=p_20(:);
p_3=p_30(:);

% merge two inputsd
p_1=[p_2,p_3];

% random values for weights
w_1 = randn(200,400) * 0.1;
% random values for biases
b_1 = rand(200,1) - 0.5;
b_2 = rand(400,1) - 0.5;

% random values for weights
w_2 = randn(100,200) * 0.1;
% random values for biases
b_3 = rand(100,1) - 0.5;
b_4 = rand(200,1) - 0.5;
[R,Q]=size(p_1);

[w_1,b_1,b_2,a_20,a_21] = CD11(w_1,p_1,b_1,b_2);
[w_2,b_3,b_4,a_41] = CD12(w_2,a_20,a_21,b_3,b_4);

iteration=1e7;
beta=0.001;

for i=1:iteration
%FORWARD

w_3 = w_2';
w_4 = w_1';
a_10=logsig(w_1*p_1 + b_1*ones(1,2));
```

```
a_20=(w_2*a_10 + b_3*ones(1,2));
a_30=(w_3*a_20 + b_4*ones(1,2));
a_40=logsig(w_4*a_30 + b_2*ones(1,2));
e = p_1 - a_40;

%SENSITIVITIES

Fdot4=a_40.*(1-a_40);
Fdot1=a_10.*(1-a_10);
s_4=-2.*Fdot4.*(e);
s_3=w_1*s_4;
s_2=w_2*s_3;
s_1=Fdot1.*(w_2'*s_2);

%SUMMES

sum1_1=(s_1*p_1');
sum1_2=sum(s_1,2);
sum2_1=(s_2*a_10');
sum2_2=sum(s_2,2);
sum3_1 = (s_3*a_20');
sum3_2 = sum(s_3,2);
sum4_1 = (s_4*a_30');
sum4_2 = sum(s_4 , 2);
w_1=w_1 - (beta*sum1_1)/Q;
b_1=b_1 - (beta*sum1_2)/Q;
w_2=w_2 - (beta*sum2_1)/Q;
b_3=b_3 - (beta*sum2_2)/Q;
w_3 = w_3 - (beta*sum3_1)/Q;
b_4 = b_4 - (beta*sum3_2)/Q;
w_4 = w_4 - (beta*sum4_1)/Q;
b_2 = b_2 - (beta*sum4_2)/Q;

sum_e(i)=trace(e'*e);
end
```

```
1
2 figure;                          %shows SSE
3 loglog(sum_e)
4
5
6 %%%%% This is a plotting part in matrix form to show the ...
     shape%%%%%%%
```

```
1  function [w_1,b_1,b_2,a_20,a_21] = CD11(w_1,p_1,b_1,b_2)
2  alpha = 0.001;
3  itr=50;
4  for i=1:itr
5              % FIRST-RPM-FW:
6          a_0   = logsig(w_1 * p_1 + b_1*ones(1,2));
7          a_00  = sum(a_0,2);
8          a_1   = a_0>rand(200,2);
9          out_1 = a_1*p_1';
10         p_11  = sum(p_1,2);
11
12         % FIRST-RPM-BCK:
13         a_2   = logsig(w_1'* a_1 + b_2*ones(1,2));
14         a_21  = logsig(w_1*  a_2 + b_1*ones(1,2));
15         a_20  = sum(a_21,2);
16         out_2 = a_21*a_2' ;
17         a_22  = sum(a_2,2);
18
19         %UPDATE WIGHT AND BIASES
20
21         w_1  = w_1 + (alpha)*(out_1 - out_2);
22         b_1  = b_1 + (alpha)*(a_00  - a_20);
23         b_2  = b_2 + (alpha)*(p_11  - a_22);
24
25   end
```

```
1
2  function [w_2,b_3,b_4,a_41] = CD12(w_2,a_20,a_21,b_3,b_4)
3  alpha = 0.001;
4  itr=50;
5  for i=1:itr
6
7      % RPM-FW:
8      a_3   = logsig(w_2 * a_21 + b_3*ones(1,2));
9      a_30  = sum(a_3,2);
10     out_1 = a_3*a_21';
11
12     % RPM-BCK:
13     a_4   = logsig(w_2'* a_3 + b_4*ones(1,2));
14     a_41  = logsig(w_2*a_4 + b_3*ones(1,2));
15     a_42=sum(a_41,2);
16     a_43=sum(a_4,2);
17     out_2 = a_41*a_4' ;
18
19     %UPDATE WIGHT AND BIASES
20
21     w_2  = w_2 + (alpha)*(out_1-out_2);
22     b_3 = b_3 + (alpha)*(a_30 - a_42);
23     b_4 = b_4 + (alpha)*(a_20-a_43);
24
25 end
```

References

[1] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, pp. 504 – 507, July 2006.

[2] P. Joshi, "Autoencoders in machine learning," October 2014.

[3] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," vol. 14, no. 8, pp. 1711–1800, Augest 2002.

[4] ——, "A practical guide to training restricted boltzmann machines," Augest 2010.

[5] G. E. Hinton and R. R. Salakhutdinov, "supporting material on science online," Tech. Rep.

[6] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. D. Joses, *Chapter 7,Neural Network Design*, second, Ed.