

1 Problem set 1

predict the water level at Nawarat Bridge

1.1 problem

We need to predict the water level at Nawarat Bridge in the next 7 hours. using the Flood data set Station 1 and Station 2 at the current time and back up to 3 hours, So this data set has 8 input. this Flood data set has data of

Station 1 at time $t-3$, $t-2$, $t-1$, $t-0$.

Station 2 at time $t-3$, $t-2$, $t-1$, $t-0$.

and 1 output is a water level at Nawarat Bridge is a Desire Output

1.2 process

We separate 90 % to train and 10 % to blind-test for cross validation (selecting 1 line data every 10 lines of data to distribute the grouping of data)

then try changes hidden nodes including learning , momentum rates to test the impact. (different initialize weights between -1.0 to 1.0)

$$\varphi(x) = \begin{cases} x, & \text{if } x > 0, \\ 0.01, & \text{otherwise.} \end{cases} \quad (1)$$

The activation function used Leaky ReLU (1) ,Leaky ReLU solved the problem that the old ReLU had . for example:if nodes output is 0. that make neuron network lost the gradient for back-propagation and the weights were not updated or learn.

The 8-5-5-5-1 neural network is used because from many experiments It has a fast learning curve. and less error than others.

So we adjust parameters including hidden nodes , learning , momentum rates to see what happens. by following parameters table.

neural test				
Test	Learning Rate	Momentum Rate	Biases	Neural Type
base	0.9	0.9	0.9	8-5-5-5-1
A	0.01	0.0	0.0	8-5-1
B	0.1	0.5	0.5	8-5-4-3-2-1
C	0.01	0.1	0.1	8-16-16-1

1.3 training

All dataset was normalized using min-max normalization between 0.0 and 1.0. for friendly with activation function

$$x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Then data-set are used for the training neural network is as followed:

- Split the dataset into 10 groups (10% by 10 unique rounds).
- For each unique group, 90% train 10% test
 - 1. Fit model by randomly data from training set
 - 2. evaluate the model using the train set.
 - 3. get the error that neural network predict.
 - 4. Backpropagation to updated weights and learn
 - 5. measure neural network by test data-set

1.4 coding

Training process done by this code:

```
1 public void train(){
2     int epoch =0;
3     while (N < maxEpoch && avg_error_n > minError){
4
5         for(int data = 0; data < train_dataset.size() ; data++) {
6             //setup randomly input data
7             int ran_dataset_i = (int) (Math.random() * ((train_dataset.size()) ));
8
9             //set dataset value to input node
10            for(int input_i = 0 ; input_i < neural_type[0] ; input_i++){
11                node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
12            }
13
14            //cal sum_(input x weight) -> activation_Fn for each neuron_node
15            forward_pass();
16
17            get_error(ran_dataset_i);
18            backward_pass();
19        }
20
21        double sum = 0.0;
22        for (Double[] doubles : error_n) {
23            // sum E(n) = 1/2 sum e^2 : sum of squared error at iteration n (sse)
24            sum += 0.5*Math.pow(doubles[0], 2);
25        }
26        // avg_E(n) = 1/N sum E(n) : avg (sse)
27        avg_error_n = sum / (error_n.size());
28        epoch++; // next epoch
29    }
30 }
```

```

1 private void forward_pass(){
2     for(int layer = 0; layer < neural_type.length-1 ; layer++) {
3
4         double sum_input;
5         Double[] sum_inputnode = new Double[neural_type[layer+1]];
6
7         //mutiply matrix
8         for (int j = 0; j < neural_type[layer+1] ; j++){
9             double sum=0;
10            for(int k=0;k<node[layer].length;k++)
11            {
12                //w_ji : weight from input neuron j to neron i : in each layer
13                sum += layer_weight[layer].data[j][k] * activation_fn( node[layer][k])
14                ;
15            }
16            // V_j = sum all input*weight i->j + biases
17            sum_input = sum + biases;
18            sum_inputnode[j] = sum_input;
19        }
20        // O_k = output of neuron_node k in each layer
21        node[layer+1] = sum_inputnode;
22    }
23 }

```

- forward_pass() is just matrix multiply [line 8] by sum of the multiplication weight-line and output value of node. then add it in to next layer node.
- get_error() is function to get error of neural network at output node by desire Output in training set
- backward_pass() is function that use local gradient to calculate delta weight of all weight-line

Then neural network change all own weight by this code:

```

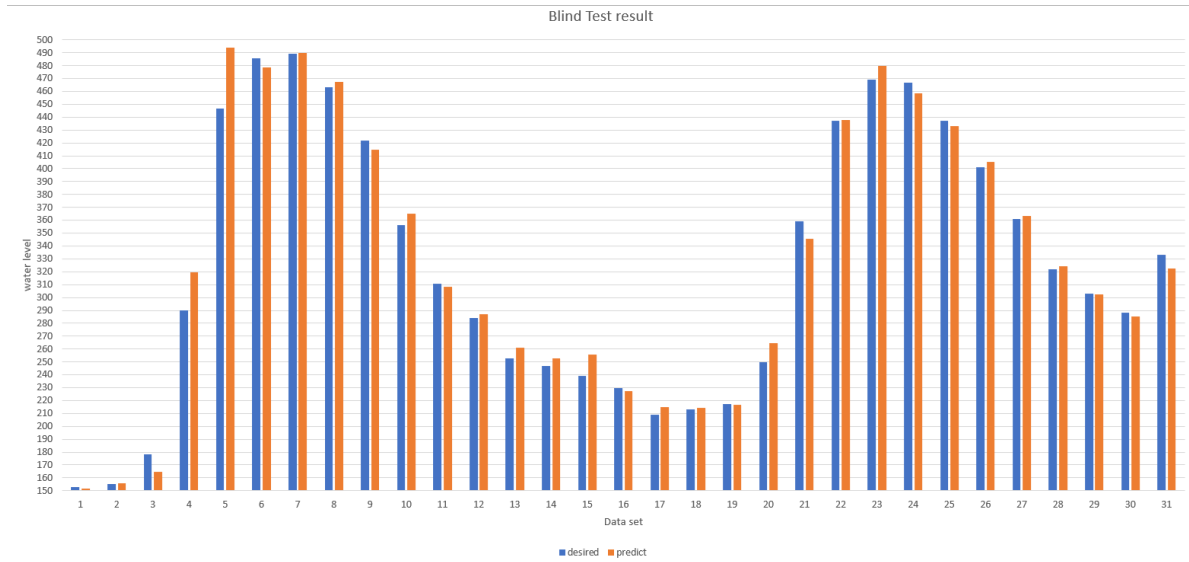
1 for (int weight_layer = layer_weight.length-1 ; weight_layer >= 0 ; weight_layer--)
2 {
3     layer_weight[weight_layer] = Matrix.plus_matrix(layer_weight[weight_layer],
4         change_weight[weight_layer]) ;
5 }

```

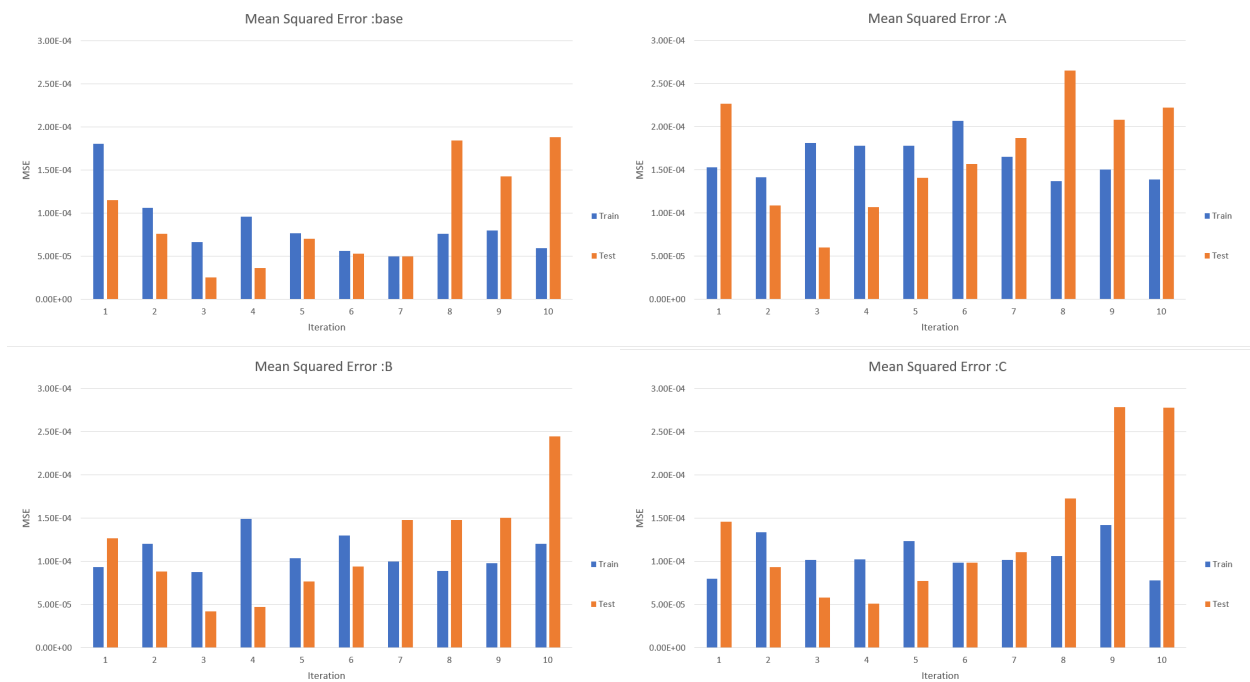
All the source code are in github: <https://github.com/min23asdw/neural-network>

1.5 result

This table is compare between desired and predict Water Level with blind test data-set using neural network :base in [neural test] table



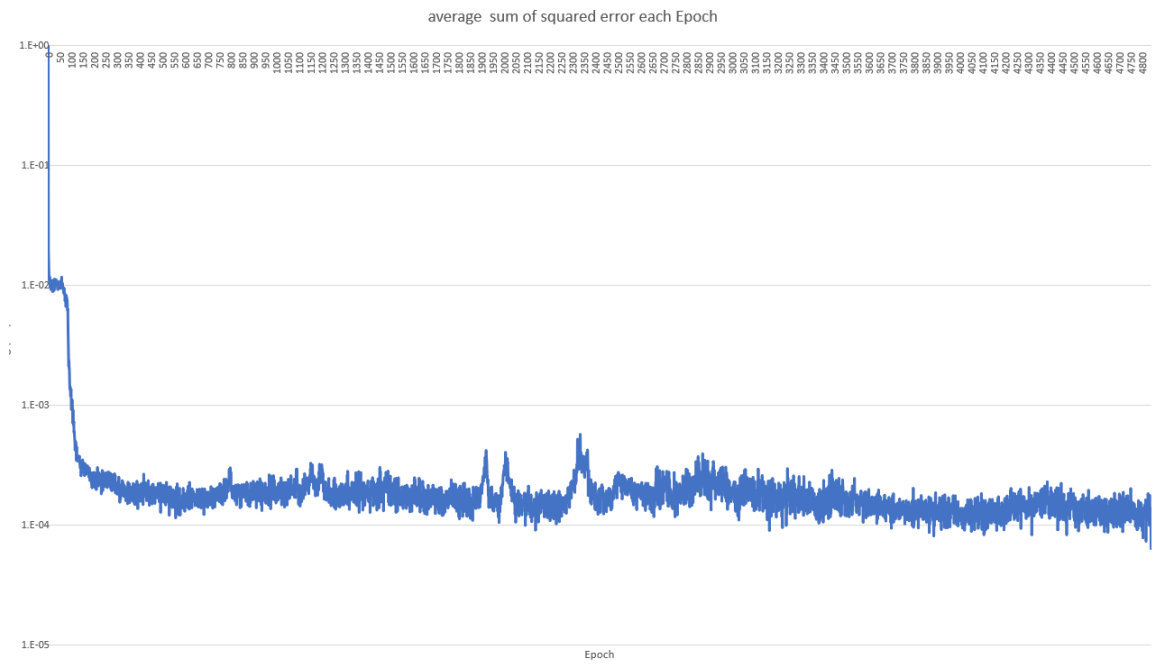
By this table is the result of 10%-cross validation by 10 Iteration show mean squared error of Train and Test form neural network



1.6 neural network analysis

Overall, the [base] test performed best, followed by B ,C, A . the C test, had largest layer of nodes. But not working as well like [base] This means more number of layer that doesn't always mean neural network more accurate.

By this training curve the learning curve are good at 1-100 epoch by can greatly reduce the error but the accuracy stuck at E-04 for a while until it breaks down to E-05 at more than 4000 epoch and there is a variance in around 1800-2400 epoch and we don't know how many epoch will use to decrease error until reaching the E-06 or there is no way to reach



However, number of layer and node have a huge impact If selected appropriately with the complexity of the data,

Finally, besides the number of layer and node Parameter momentum rate , learning rate and number of epochs, the training data set and test data set is important as well.

2 Problem set 2

cross.pat 2 classes and 2 features

2.1 problem

We need to predict 2 classes $[1, 0]$ or $[0, 1]$, by using the 2 input data that have values between 0 to 1, So this problem. neural network will have 2 input node and 2 output node. On $[1, 0]$ we call (**True**) and $[0, 1]$ we call (**False**)

2.2 process

We separate 90 % to train and 10 % to blind-test for cross validation (selecting 1 line data every 10 lines of data to distribute the grouping of data)
then try changes hidden nodes including learning , momentum rates to test the impact. (different initialize weights between -1.0 to 1.0)

$$\varphi(x) = \begin{cases} x, & \text{if } x > 0, \\ 0.01, & \text{otherwise.} \end{cases} \quad (2)$$

The activation function used Leaky ReLU (1) ,Leaky ReLU solved the problem that the old ReLU had . for example:if nodes output is 0. that make neuron network lost the gradient for back-propagation and the weights were not updated or learn.

Then we measure

-Precision as a comparison of the correct prediction is true and it true (TP) and the prediction is true but is not true (FP).

-Recall as the accuracy of the prediction "true" compared to the number of times the event both predicted and actual as "true".

-Accuracy as the accuracy of predict matches actually correctly

The 2-8-2 neural network is used because from many experiments It has a fast learning curve. and less error than others.

So we adjust parameters including hidden nodes , learning , momentum rates to see what happens. by following parameters table.

neural test				
Test	Learning Rate	Momentum Rate	Biases	Neural Type
base	0.01	0.1	1	2-8-2
A	0.01	0.0	0.0	2-5-2
B	0.01	0.0	0.5	2-5-4-3-2
C	0.1	0.1	1	2-16-16-2

2.3 training

All dataset was normalized using min-max normalization between 0.0 and 1.0. Even if the value is between 0 and 1, because we don't know for sure if these two inputs are the same type of data.

Then data-set are used for the training neural network is as followed:

- Split the dataset into 10 groups (10% by 10 unique rounds).
- For each unique group, 90% train 10% test
 - 1. Fit model by randomly data from training set
 - 2. evaluate the model using the train set.
 - 3. get the error that neural network predict.
 - 4. Backpropagation to updated weights and learn
 - 5. measure neural network by test data-set

2.4 coding

All training process same like [Problem set 1] but for easy classification we classify by $>$ and $<$ form 2 output node.

So we have 2 output node in neuron network (0 , 1) if node-0 $>$ node-1 set to class [1,0] else set to class [0,1]

```
1 // class set
2 if(output_node[0] > output_node[1]){
3     output_node[0] = 1.0;
4     output_node[1] = 0.0;
5 }else {
6     output_node[0] = 0.0;
7     output_node[1] = 1.0;
8 }
```

For confusion matrix classification if output_node[0] more than output_node[1] or (output_node[0] = 1.0 & output_node[1] = 0.0) call (**True**) predict else (output_node[0] = 1.0 & output_node[1] = 0.0) call (**False**) predict.

Code to count True Positive (**TP**) , True Negative (**TN**), False Positive (**FP**), False Negative (**FN**)

```
1 if(output_node == train_desired && output_node[0].equals(1.0) )
2     true_positive++;
3 if(output_node == train_desired && output_node[0].equals(0.0) )
4     true_negative++;
5 if(output_node != train_desired && output_node[0].equals(1.0) )
6     false_positive++;
7 if(output_node != train_desired && output_node[0].equals(0.0) )
8     false_negative++;
```

All the source code are in github: <https://github.com/min23asdw/neural-network>

Then calculate Precision, Recall, Accuracy by this formula:

$$\text{Precision} = \text{TPs} / (\text{TPs} + \text{FPs})$$

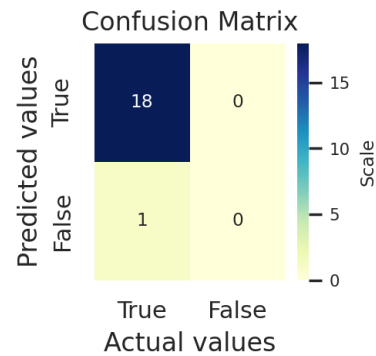
$$\text{Recall} = \text{TPs} / (\text{TPs} + \text{FNs})$$

$$\text{Accuracy} = (\text{TPs} + \text{TNs}) / (\text{TPs} + \text{TNs} + \text{FPs} + \text{FNs})$$

2.5 result

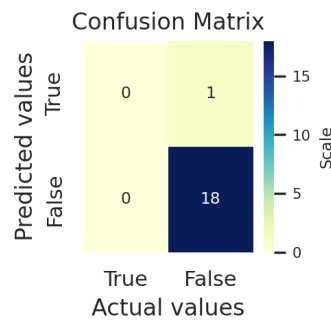
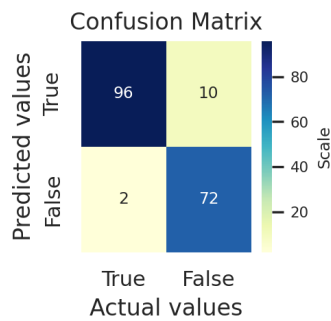
This table show Confusion Matrix with blind test data-set using neural network :base in [neural test] table

By this table is the result of 10%-cross validation by 10 Iteration show mean squared error of Train and Test form neural network

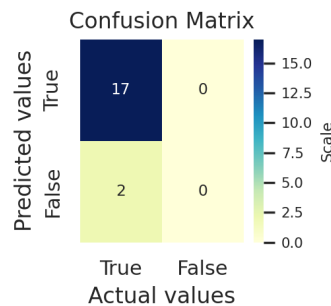
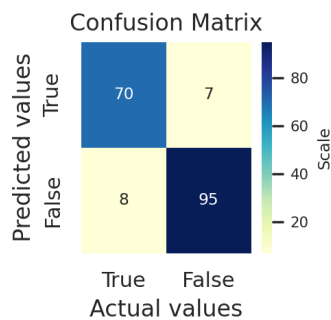


On the left side is training , right side is blind test.

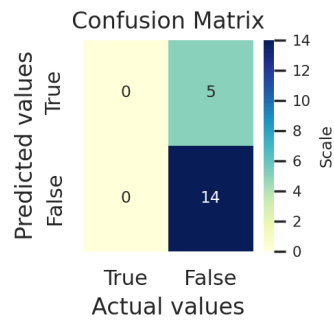
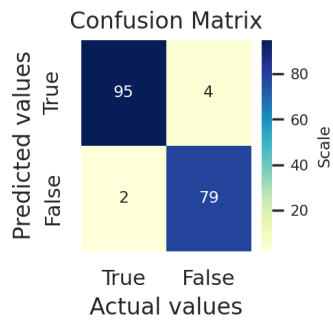
Iteration 0



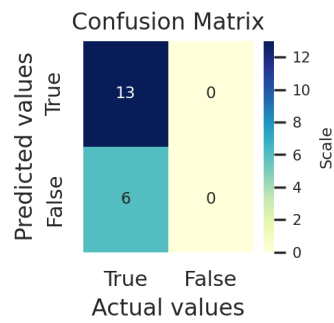
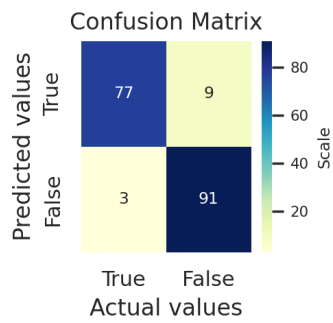
Iteration 1



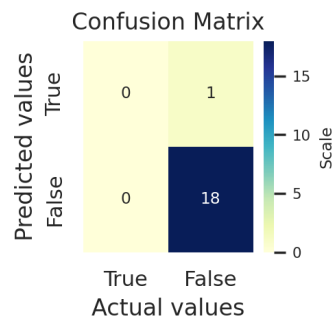
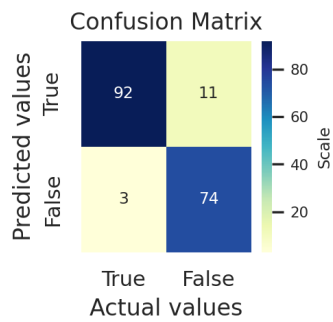
Iteration 2



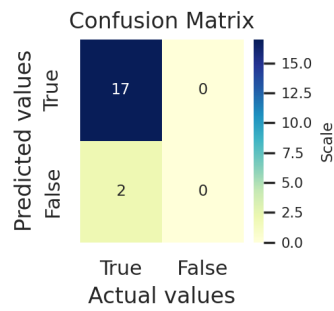
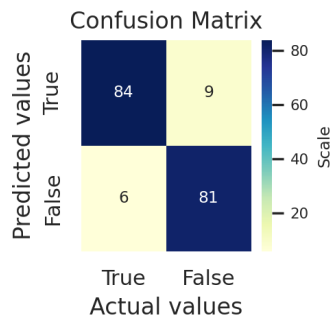
Iteration 3



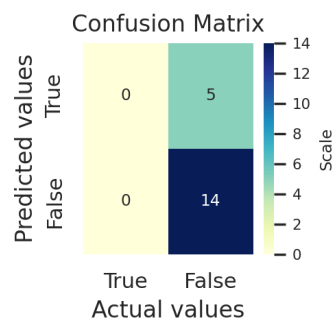
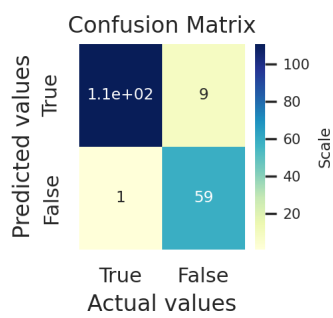
Iteration 4



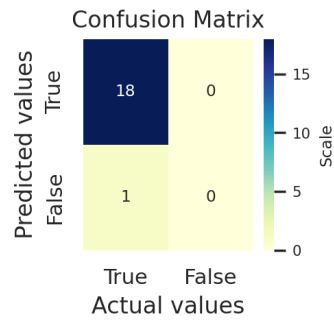
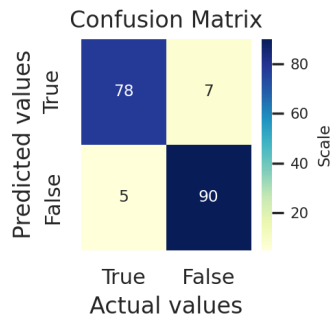
Iteration 5



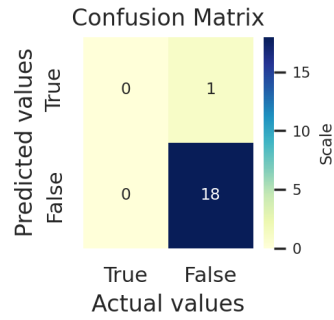
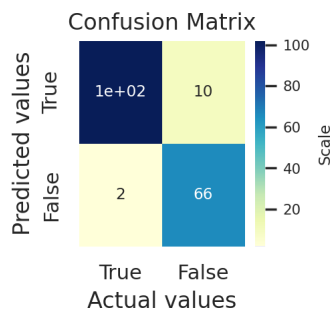
Iteration 6



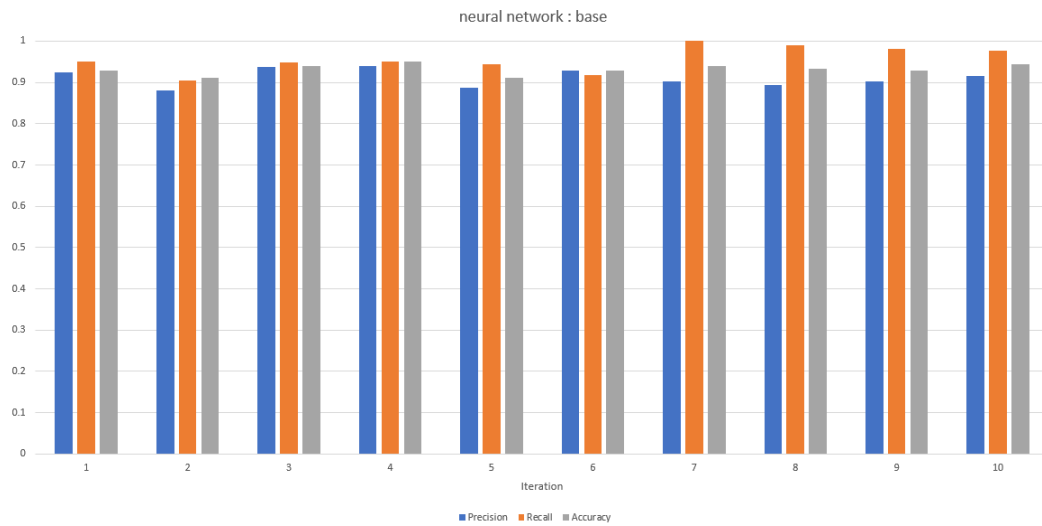
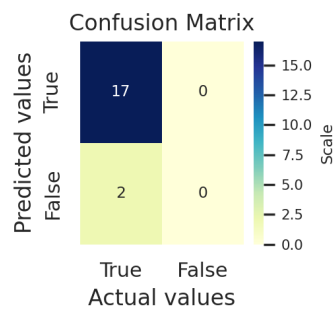
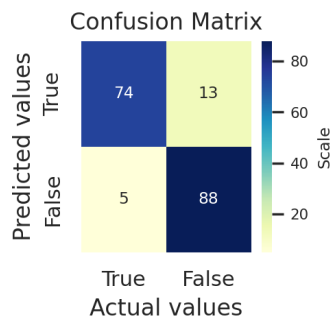
Iteration 7

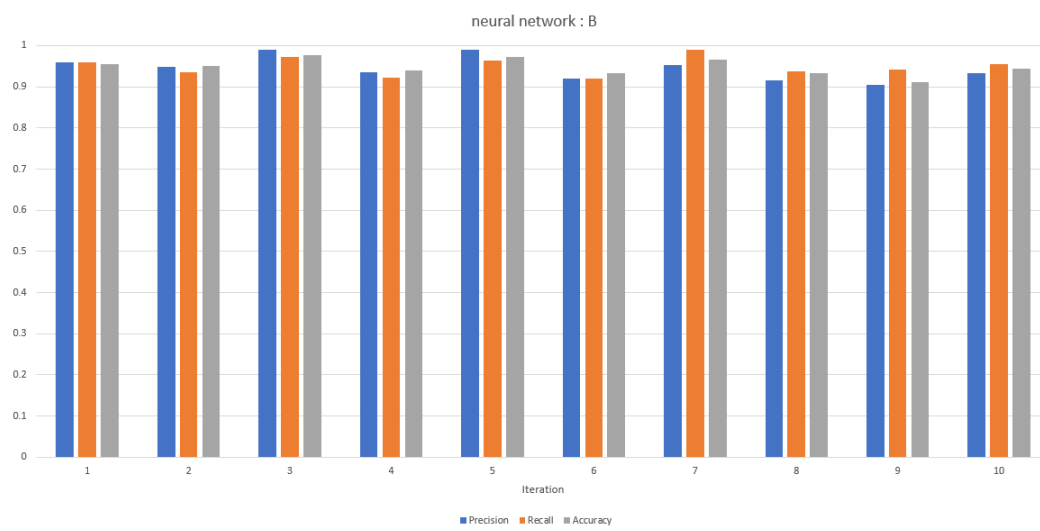
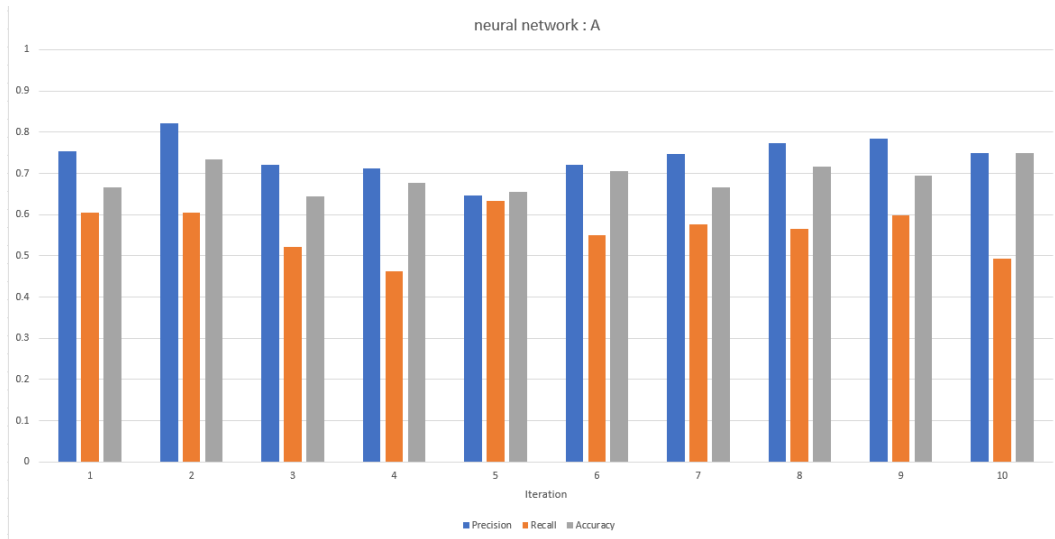


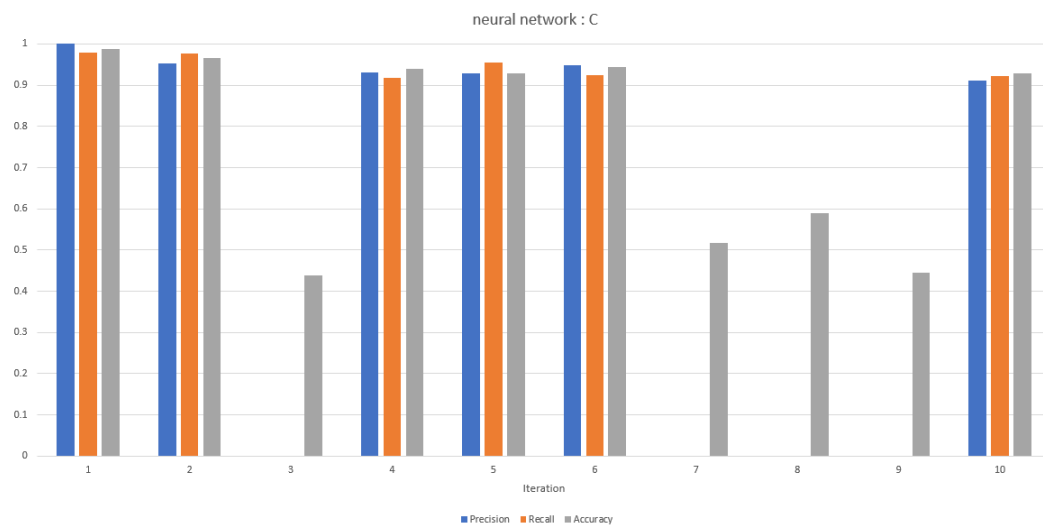
Iteration 8



Iteration 9







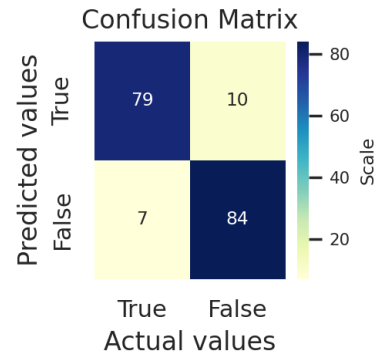
2.6 neural network analysis

Overall, the [base] test performed best, followed by B ,A, C . the C test, had largest nodes for each layer. But not working as well like [base] This means more number of node in each layer that doesn't always mean neural network more accurate.

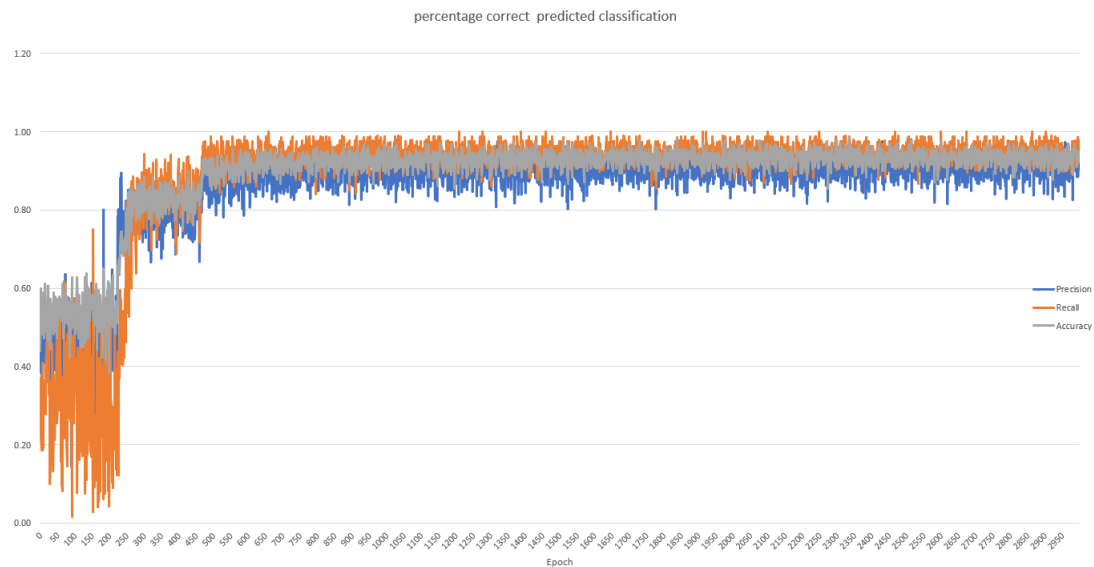
This table show Confusion Matrix when training with training data-set on neural network :base in [neural test] table

On training neural network are good to predict True Positive (TP) , True Negative (TN)

but some time False Positive (FP), False Negative (FN) appeared.



By this training table the learning curve was a significant increase in accuracy. at 1-500 epoch by can greatly reduce the error. the accuracy around 90% - 96% and accuracy are 100% some time



Finally, number of node in each layer have a huge impact If selected appropriately

with the complexity of the data, besides the number of layer and node Parameter momentum rate , learning rate and number of epochs, the training data set and test data set is important as well.

3 Source code

3.1 Main code for Problem set 1

```
1 import java.io.*;
2 import java.util.ArrayList;
3
4 public class main {
5
6     private static ArrayList<ArrayList<Double[]>> test_dataset = new ArrayList<>();
7     private static ArrayList<ArrayList<Double[]>> test_desired_data = new ArrayList<>();
8     private static ArrayList<ArrayList<Double[]>> train_dataset = new ArrayList<>();
9     private static ArrayList<ArrayList<Double[]>> train_desired_data = new ArrayList<>();
10
11     private static int NumberOftest = 10;
12
13     public static void main(String[] args) throws IOException {
14
15         for(int tain_i = 0 ; tain_i < NumberOftest ; tain_i ++){
16
17             ArrayList<Double[]> test_dataset_i = new ArrayList<>();
18             ArrayList<Double[]> test_desired_data_i = new ArrayList<>();
19             ArrayList<Double[]> train_dataset_i = new ArrayList<>();
20             ArrayList<Double[]> train_desired_data_i = new ArrayList<>();
21
22             FileInputStream fstream = new FileInputStream("src/Flood_dataset.txt");
23             DataInputStream in = new DataInputStream(fstream);
24             BufferedReader br = new BufferedReader(new InputStreamReader(in));
25             String data;
26
27             int line_i = 0;
28             while ((data = br.readLine()) != null) { // each line
29                 String[] tmp = data.split("\t"); //Split space
30
31                 Double[] tmp_dataset = new Double[tmp.length-1];
32                 Double[] tmp_desired_data = new Double[1];
33
34                 int word_i = 0;
35                 for (String t : tmp) { // each word
36                     double tmp_val = (Double.parseDouble(t)/700.0); // norm
37
38                     if (word_i == tmp.length - 1) { // desired_data
39                         tmp_desired_data[0] = (tmp_val) ;
40                     } else {
41                         tmp_dataset[word_i] = (tmp_val) ;
42                     }
43
44                     word_i++;
45                 }
46
47                 if (line_i % (10) == tain_i) { // test 10% data
48                     test_dataset_i.add(tmp_dataset);
49                     test_desired_data_i.add(tmp_desired_data);
50                 } else { //train 90%
51                     train_dataset_i.add(tmp_dataset);
52                     train_desired_data_i.add(tmp_desired_data);
53                 }
54             }
55         }
56     }
57 }
```



```

54         line_i++;
55     }
56     test_dataset.add(test_dataset_i);
57     test_desired_data.add(test_desired_data_i);
58     train_dataset.add(train_dataset_i);
59     train_desired_data.add(train_desired_data_i);
60
61 }
62
63 for(int test_i = 1 ; test_i < NumberOftest ; test_i ++ ) {
64
65     brain b1 = new brain("8,5,5,5,1", 5000, 0.00001, 1, 0.05, 0.9);
66     System.out.println("train: " + test_i);
67     b1.train( train_dataset.get(test_i), train_desired_data.get(test_i));
68     System.out.println("test: " + test_i);
69     b1.test(test_dataset.get(test_i), test_desired_data.get(test_i));
70
71 }
72 }
73
74 }

```

3.1.1 brain code for Problem set 1

```
1 import java.util.ArrayList;
2
3 public class brain {
4     private final int[] neural_type;
5     private ArrayList<Double[]> train_dataset;
6     private ArrayList<Double[]> train_desired_data;
7
8     private final int maxEpoch;
9     private final double minError;
10    private final double learning_rate;
11    private final double moment_rate;
12
13    //sum of squared error at iteration n (sse)
14    private final ArrayList<Double[]> error_n = new ArrayList<>();
15    private double avg_error_n = 1000000000 ; // average sse of all epoch
16    private double biases; // threshold connected : biases
17
18    private Matrix[] layer_weight ;
19    private Matrix[] change_weight;
20
21    //private Matrix[] old_change_weight;
22    private Double[][] node ;
23    private Double[][] local_gradient_node ;
24
25    public brain(String _neural_type , int _maxEpoch , double _minError ,double
        _biases, double _learning_rate , double _moment_rate){
26
27        String[] splitArray = _neural_type.split(",");
28        int[] array = new int[splitArray.length];
29        for (int i = 0; i < splitArray.length; i++) array[i] = Integer.parseInt(
            splitArray[i]);
30
31        this.neural_type = array;
32
33        init_Structor();
34
35        this.maxEpoch = _maxEpoch;
36        this.minError = _minError;
37        this.biases = _biases;
38        this.learning_rate = _learning_rate;
39        this.moment_rate = _moment_rate;
40    }
41
42    private void init_Structor(){
43        node = new Double[neural_type.length][];
44        local_gradient_node = new Double[neural_type.length][];
45        for (int i = 0; i < neural_type.length; i++) {
46            node[i] = new Double[neural_type[i]];
47            local_gradient_node[i] = new Double[neural_type[i]];
48        }
49
50        layer_weight = new Matrix[neural_type.length-1];
51        change_weight = new Matrix[neural_type.length-1];
52        for (int layer = 0; layer < layer_weight.length; layer++) {
53            Matrix weight = new Matrix(neural_type[layer+1],neural_type[layer] ,true)
                    ;
54            Matrix change = new Matrix(neural_type[layer+1],neural_type[layer] ,false
                    );

```

```

55         layer_weight[layer] = weight;
56         change_weight[layer] = change;
57     }
58
59 }
60
61 public void train(ArrayList<Double[]> _train_dataset, ArrayList<Double[]>
    _train_desired_data ){
62     this.train_dataset = _train_dataset;
63     this.train_desired_data = _train_desired_data;
64
65     int N =0;
66     while (N < maxEpoch && avg_error_n > minError){ //
67
68         unique_random uq = new unique_random(train_dataset.size());
69
70         for(int data = 0; data < train_dataset.size() ; data++) {
71             //random one dataset
72             int ran_dataset_i = uq.get_line();
73             //setup dataset value to input node
74             for(int input_i = 0 ; input_i < neural_type[0] ; input_i++){
75                 node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
76             }
77
78             //cal sum_(input x weight) -> activation_Fn for each neuron_node
79             forward_pass();
80
81             get_error(ran_dataset_i );
82             backward_pass();
83
84             double d = train_desired_data.get(ran_dataset_i)[0]*700 ;
85             double g = activation_fn( node[node.length-1][0]*700 );
86 // System.out.println("desired:" +(int)d+"get:"+g+"\t error_n:"+Math.abs(d-g));
87         }
88
89         double sum = 0.0;
90         for (Double[] doubles : error_n) {
91 // sum E(n) = 1/2 sum of e^2 : sum of squared error at iteration n (sse)
92             sum += 0.5*Math.pow(doubles[0], 2);
93         }
94         // avg_E(n) = 1/N sum_ E(n) : avg (sse)
95         avg_error_n = sum / (error_n.size());
96
97         error_n.clear();
98
99         System.out.println( N + " \t " + avg_error_n);
100         N++; // next epoch
101     }
102
103     System.out.println("avg_error_n final : " + avg_error_n);
104 }
105
106 private void forward_pass(){
107     for(int layer = 0; layer < neural_type.length-1 ; layer++) {
108
109         // W r_c X N r_1 = N+1 r_1
110         if( layer_weight[layer].cols != node[layer].length){
111             System.out.println("invalid matrix");
112             return;
113         }

```

```

114
115     double sum_input;
116     Double[] sum_inputnode = new Double[neural_type[layer+1]];
117
118     //mutiply matrix
119     for (int j = 0; j < neural_type[layer+1] ; j++){
120         double sum=0;
121         for(int k=0;k<node[layer].length;k++)
122         {
123             //w_ji : weight from input neuron j to neron i : in each layer
124             sum += layer_weight[layer].data[j][k] * activation_fn( node[
                layer][k]) ;
125         }
126         // V_j = sum all input*weight i->j + biases
127         sum_input = sum + biases;
128         sum_inputnode[j] = sum_input;
129     }
130     // O_k = output of neuron_node k in each layer
131     node[layer+1] = sum_inputnode;
132 }
133 }
134
135 private void get_error(int ran_dataset_i ) {
136
137     int number_outputn_node = node[node.length-1].length;
138     Double[] errors = new Double[number_outputn_node];
139     for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++) {
140         //train_desired_data => d_j desired output for neuron_node j at iteration N
141         // it have "one data"
142         //e_j = error at neuron j at iteration N
143         double desired = train_desired_data.get(ran_dataset_i)[0];
144         errors[outnode_j] = desired - activation_fn( node[node.length-1][
            outnode_j] ) ;
145
146         double diff_fn = diff_activation_fn(node[node.length-1 ][outnode_j]);
147         node[node.length-1 ][outnode_j]
148         local_gradient_node[node.length-1][outnode_j] = errors[outnode_j] *
            diff_fn;
149     }
150     error_n.add(errors);
151 }
152
153 private void backward_pass() {
154     //diff_sse/w_j = diff_sum_(n) / diff_e_j * diff_e_j / diff_Y_j *
155     // diff_Y_j / diff_V_j * diff_V_j / diff_w_ji
156     //diff_sse/w_j = (e_j(n)) * -1 * diff_Y_j(sum_input) * Y_i
157
158     // delta_weight_ji = - learning rate ( diff_sse/w_j )
159     // delta_weight_ji = learning rate [ (e_j(n)) * diff_Y_j(sum_input) * Y_i ]
160     // add Momentum
161     // delta_weight_ji = delta_weight_ji(old) + delta_weight_ji
162     // wji_next = wji_now + delta_weight_ji
163
164     // output change_weight
165     int output_layer = node.length-1;
166     delta_weight_outputnode(output_layer);
167
168     //local gradient output_k= e_k * diff Y_k :: local gradient hidden_j =
169     diff Y_j * sum_ ( W_kj * l_g k)

```

```

168     local_gradient();
169     for (int layer = node.length-3 ; layer >= 0 ; layer--) {
170         // hidden layer change_weight
171         // delta_weight_ji = learning_rate * local_gradient_j * Y_i
172         delta_weight_hiddennode(layer);
173     }
174
175     for (int weight_layer = layer_weight.length-1 ; weight_layer >= 0 ;
176         weight_layer--) {
177         layer_weight[weight_layer] = Matrix.plus_matrix(layer_weight[weight_layer]
178             , change_weight[weight_layer]) ;
179     }
180
181     public void delta_weight_outputnode(int layer){
182         int weight_layer = layer-1;
183
184         //mutiply matrix
185         for (int j = 0; j < error_n.get(error_n.size()-1).length ; j++){
186
187             double diff_fn = diff_activation_fn(node[layer][j]);
188             for(int i=0;i< node[layer-1].length ; i++)
189             {
190                 double old_weight = moment_rate * change_weight[weight_layer].data[j][i];
191                 double delta_weight = learning_rate * (error_n.get(error_n.size()-1)[j] *
192                     diff_fn * activation_fn(node[layer-1][i])) ;
193                 double delta = old_weight + delta_weight;
194                 change_weight[weight_layer].set(j,i,delta);
195             }
196         }
197     }
198
199     private void local_gradient() {
200         for (int layer = layer_weight.length-1 ; layer >= 0 ; layer--) {
201             for (int j = 0; j < node[layer].length ; j++){
202                 double sum_j = 0;
203                 for(int k=0;k< node[layer+1].length ; k++)
204                 {
205                     sum_j += ( local_gradient_node[layer+1][k]) * layer_weight[layer].
206                         data[k][j] ;
207                 }
208                 // node[layer][j]
209                 double diff_fn = diff_activation_fn(node[layer][j]);
210                 local_gradient_node[layer][j] = sum_j * diff_fn;
211             }
212         }
213     }
214
215     public void delta_weight_hiddennode(int weight_layer){
216         int node_layer = weight_layer+1;
217
218         //mutiply matrix
219         for (int j = 0; j < node[node_layer].length ; j++){
220             for(int i=0;i< node[node_layer-1].length ; i++)
221             {
222                 double old_weight = moment_rate * change_weight[weight_layer].data[j]
223                     [i];
224                 double delta_weight = learning_rate * ( local_gradient_node[

```

```

223         node_layer][j] * activation_fn(node[node_layer-1][i])) ;
224         double delta = old_weight + delta_weight;
225
226         change_weight[weight_layer].set(j,i,delta);
227     }
228 }
229
230 public void test(ArrayList<Double[]> _test_dataset, ArrayList<Double[]>
231     _test_desired_data){
232
233     //setup input data
234     for(int test_i = 0; test_i < _test_dataset.size()-1 ; test_i++) {
235
236         //set dataset value to input node
237         for (int input_i = 0; input_i < neural_type[0]; input_i++) {
238             node[0][input_i] = _test_dataset.get(test_i)[input_i];
239         }
240
241         forward_pass();
242
243         int number_outputn_node = node[node.length-1].length;
244         Double[] errors = new Double[number_outputn_node];
245         for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++)
246         {
247             double desired = _test_desired_data.get(test_i)[0];
248             errors[outnode_j] = desired -node[node.length-1][outnode_j];
249         }
250         error_n.add(errors);
251
252         double d = _test_desired_data.get(test_i)[0]*700 ;
253         double g = node[node.length-1][0]*700;
254         System.out.println("desired:" + (int)d + " get: " + g + "\t error_n: " +
255             Math.abs(d-g));
256     }
257
258     double sum = 0.0;
259     for (Double[] doubles : error_n) {
260         // sum_E(n) = 1/2 sum_e^2 : sum of squared error at iteration n (sse)
261         sum += 0.5*Math.pow(doubles[0], 2);
262     }
263     // avg_E(n) = 1/N sum_E(n) : avg (sse)
264     avg_error_n = sum / (error_n.size());
265
266     System.out.println("avg_error_n final : " + avg_error_n);
267     error_n.clear();
268 }
269
270 public double activation_fn(Double x){
271     return Math.max(0.01,x);
272 }
273
274 public double diff_activation_fn(Double v ){
275     if(v<=0){
276         return 0.01;
277     }else{
278         return 1;
279     }
280 }
281 }

```

3.2 Main code for Problem set 2

```
1 import java.io.*;
2 import java.util.ArrayList;
3
4 public class main_2 {
5
6     private static ArrayList<ArrayList<Double[]>> test_dataset = new ArrayList<>();
7     private static ArrayList<ArrayList<Double[]>> test_desired_data = new ArrayList
8         <>();
9
10    private static ArrayList<ArrayList<Double[]>> train_dataset = new ArrayList<>();
11    private static ArrayList<ArrayList<Double[]>> train_desired_data = new ArrayList
12        <>();
13
14    private static int NumberOftest = 10;
15
16    public static void main(String[] args) throws IOException {
17        for(int tain_i = 0 ; tain_i < NumberOftest ; tain_i ++){
18            ArrayList<Double[]> test_dataset_i = new ArrayList<>();
19            ArrayList<Double[]> test_desired_data_i = new ArrayList<>();
20
21            ArrayList<Double[]> train_dataset_i = new ArrayList<>();
22            ArrayList<Double[]> train_desired_data_i = new ArrayList<>();
23
24            FileInputStream fstream = new FileInputStream("src/cross.pat");
25            DataInputStream in = new DataInputStream(fstream);
26            BufferedReader br = new BufferedReader(new InputStreamReader(in));
27            String data;
28
29            int line_i = 1;
30            while ((data = br.readLine()) != null) { // each line
31                if((line_i%3 == 0 || (line_i+1)%3 == 0) { // not p line
32                    String[] eachLine = data.split("\\s+");
33                    Double[] temp = new Double[eachLine.length];
34
35                    for(int i =0 ; i<eachLine.length;i++){
36                        Double dataNum = Double.parseDouble(eachLine[i]);
37                        temp[i] = dataNum ;
38                    }
39
40                    if (line_i % 3 == 0) { // line desired
41                        if (line_i % 10 == tain_i) { // 10% for test
42                            test_desired_data_i.add(temp);
43                        } else
44                            train_desired_data_i.add(temp);
45                    } else if ((line_i + 1) % 3 == 0) { // line input
46                        if ((line_i + 1) % 10 == tain_i) { // 10% for test
47                            test_dataset_i.add(temp);
48                        } else
49                            train_dataset_i.add(temp);
50                    }
51                }
52                line_i++;
53            }
54            test_desired_data.add(test_desired_data_i);
55            train_desired_data.add(train_desired_data_i);
56            test_dataset.add(test_dataset_i);
57            train_dataset.add(train_dataset_i);
58        }
59    }
60 }
```

```

57
58
59     for(int test_i = 0 ; test_i < NumberOftest ; test_i ++ ) {
60         brain_2 b2 = new brain_2("2,8,2", 3000, 0.00007, 1, 0.01, 0.1);
61         System.out.println("train: " + test_i);
62         b2.train( train_dataset.get(test_i), train_desired_data.get(test_i));
63         System.out.println("test: " + test_i);
64         b2.test(test_dataset.get(test_i), test_desired_data.get(test_i));
65     }
66 }
67 }

```


3.2.1 brain code for Problem set 2

```
1 import java.util.ArrayList;
2
3 public class brain_2 {
4     private final int[] neural_type;
5     private ArrayList<Double[]> train_dataset;
6     private ArrayList<Double[]> train_desired_data;
7
8     private final int maxEpoch;
9     private final double minError;
10    private final double learning_rate;
11    private final double moment_rate;
12
13    private final ArrayList<Double[]> error_n = new ArrayList<>(); //sum of squared
        error at iteration n (sse)
14    private double avg_error_n = 1000000000 ; // average sse of all epoch
15    private double biases; // threshold connected : biases
16
17    private Matrix[] layer_weight ;
18    private Matrix[] change_weight;
19
20    private Double[][] node ;
21    private Double[][] local_gradient_node ;
22
23    public brain_2(String _neural_type , int _maxEpoch , double _minError ,double
        _biases, double _learning_rate , double _moment_rate){
24
25        String[] splitArray = _neural_type.split(",");
26        int[] array = new int[splitArray.length];
27        for (int i = 0; i < splitArray.length; i++) array[i] = Integer.parseInt(
            splitArray[i]);
28        this.neural_type = array;
29
30        init_Structor();
31
32        this.maxEpoch = _maxEpoch;
33        this.minError = _minError;
34        this.biases = _biases;
35        this.learning_rate = _learning_rate;
36        this.moment_rate = _moment_rate;
37    }
38    private void init_Structor(){
39        node = new Double[neural_type.length][];
40        local_gradient_node = new Double[neural_type.length][];
41        for (int i = 0; i < neural_type.length; i++) {
42            node[i] = new Double[neural_type[i]];
43            local_gradient_node[i] = new Double[neural_type[i]];
44        }
45
46        layer_weight = new Matrix[neural_type.length-1];
47        change_weight = new Matrix[neural_type.length-1];
48        for (int layer = 0; layer < layer_weight.length; layer++) {
49            Matrix weight = new Matrix(neural_type[layer+1],neural_type[layer] ,true)
                ;
50            Matrix change = new Matrix(neural_type[layer+1],neural_type[layer] ,false
                );
51            layer_weight[layer] = weight;
52            change_weight[layer] = change;
53        }
54    }
```

```

54 }
55
56
57 public void train(ArrayList<Double[]> _train_dataset, ArrayList<Double[]>
    _train_desired_data ){
58     this.train_dataset = _train_dataset;
59     this.train_desired_data = _train_desired_data;
60
61     int N =0;
62     while (N < maxEpoch && avg_error_n > minError){ //
63         double true_positive =0;
64         double true_negative =0;
65         double false_positive =0;
66         double false_negative =0;
67
68         unique_random uq = new unique_random(train_dataset.size());
69
70         for(int data = 0; data < train_dataset.size() ; data++) {
71             //random one dataset
72             int ran_dataset_i = uq.get_line();
73             //setup dataset value to input node
74             for(int input_i = 0 ; input_i < neural_type[0] ; input_i++){
75                 node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
76             }
77
78             //cal sum_(input x weight) -> activation_Fn for each neuron_node
79             forward_pass();
80
81             get_error(ran_dataset_i );
82             backward_pass();
83
84             //         double d = train_desired_data.get(ran_dataset_i)[0]*700 ;
85             //         double g = activation_fn( node[node.length-1][0]*700 );
86             //         System.out.println("desired:" + (int)d + " get: " + g + "\t error_n:
" + Math.abs(d-g));
87
88             // class set
89             if(node[node.length-1][0] > node[node.length-1][1]){
90                 node[node.length-1][0] = 1.0;
91                 node[node.length-1][1] = 0.0;
92             }else {
93                 node[node.length-1][0] = 0.0;
94                 node[node.length-1][1] = 1.0;
95             }
96
97             //         System.out.println("get");
98             //         for (Double val : node[node.length-1]) {
99             //             System.out.println(val);
100             //         }
101             //         System.out.println("desired");
102             //         for (Double val : train_desired_data.get(ran_dataset_i)) {
103             //             System.out.println(val);
104             //         }
105             //         System.out.println("++");
106
107             if(node[node.length-1][0].equals(train_desired_data.get(ran_dataset_i)
                [0]) && node[node.length-1][0].equals(1.0) ) true_positive++;
108             if(node[node.length-1][0].equals(train_desired_data.get(ran_dataset_i)
                [0]) && node[node.length-1][0].equals(0.0) ) true_negative++;
109

```

```

110         if(!node[node.length-1][0].equals(train_desired_data.get(
            ran_dataset_i)[0]) && node[node.length-1][0].equals(1.0) )
            false_positive++;
111         if(!node[node.length-1][0].equals(train_desired_data.get(
            ran_dataset_i)[0]) && node[node.length-1][0].equals(0.0) )
            false_negative++;
112
113     }
114     //Precision = true_positive/true_positive+false_positive
115     // Recall = true_positive/(true_positive+false_negative)
116     // Accuracy = (true_positive+true_negative)/(true_positive+true_negative+
        false_positive+false_negative)
117     if(N==maxEpoch-1){
118         // true_positive      true_negative      false_positive
            false_negative
119         System.out.println(true_positive+"\t"+true_negative+"\t"+
            false_positive+"\t"+false_negative);
120         System.out.println(true_positive/(true_positive+false_positive) +"\t"
            + true_positive/(true_positive+false_negative) +"\t"+ (
            true_positive+true_negative)/(true_positive+true_negative+
            false_positive+false_negative));
121     }
122     error_n.clear();
123     N++; // next epoch
124 }
125 }
126
127 private void forward_pass(){
128     for(int layer = 0; layer < neural_type.length-1 ; layer++) {
129
130         // W r_c X N r_1 = N+1 r_1
131         if( layer_weight[layer].cols != node[layer].length){
132             System.out.println("invalid matrix");
133             return;
134         }
135
136         double sum_input;
137         Double[] sum_inputnode = new Double[neural_type[layer+1]];
138
139         //mutiply matrix
140         for (int j = 0; j < neural_type[layer+1] ; j++){
141             double sum=0;
142             for(int k=0;k<node[layer].length;k++)
143             {
144                 //w_ji : weight from input neuron j to neron i : in each layer
145                 sum += layer_weight[layer].data[j][k] * activation_fn( node[
                    layer][k]) ;
146             }
147             // V_j = sum all input*weight i->j + biases
148             sum_input = sum + biases;
149             sum_inputnode[j] = sum_input;
150         }
151         // O_k = output of neuron_node k in each layer
152         node[layer+1] = sum_inputnode;
153     }
154 }
155 }
156
157 private void get_error(int ran_dataset_i ) {
158

```

```

159     int number_outputn_node = node[node.length-1].length;
160     Double[] errors = new Double[number_outputn_node];
161     for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++) {
162         //train_desired_data => d_j desired output for neuron_node j at iteration
163         N // it have "one data"
164         //e_j = error at neuron j at iteration N
165         double desired = train_desired_data.get(ran_dataset_i)[outnode_j];
166         errors[outnode_j] = desired - activation_fn( node[node.length-1][
167             outnode_j] ) ;
168
169         double diff_fn = diff_activation_fn(node[node.length-1 ][outnode_j]);
170         local_gradient_node[node.length-1][outnode_j] = errors[outnode_j] *
171             diff_fn;
172     }
173     error_n.add(errors);
174 }
175
176 private void backward_pass() {
177     //diff_sse/w_j = diff_sum(n) / diff_e_j * diff_e_j / diff_Y_j *
178     //diff_Y_j / diff_V_j * diff_V_j / diff_w_ji
179     //diff_sse/w_j = (e_j(n)) * -1 * diff_Y_j(sum_input) *
180     Y_i
181
182     // Y_j is linear_fn
183     //diff_Y_j = linear_fn -> 1
184
185     // delta_weight_ji = -learning rate ( diff_sse/w_j )
186     // delta_weight_ji = learning rate [ (e_j(n)) * diff_Y_j(sum_input) * Y_i ]
187     // delta_weight_ji = delta_weight_ji(old) + delta_weight_ji
188     // wji_next = wji_now + delta_weight_ji
189
190     // output change_weight
191     int output_layer = node.length-1;
192     delta_weight_outputnode(output_layer);
193
194     //local gradient output_k = e_k * diff_Y_k :: local gradient hidden_j =
195     diff_Y_j * sum_ ( W_kj * l_g k)
196     local_gradient();
197     for (int layer = node.length-3 ; layer >= 0 ; layer--) {
198         // hidden layer change_weight
199         // delta_weight_ji = learning rate * local_gradient_j * Y_i
200         delta_weight_hiddennode(layer);
201     }
202
203     for (int weight_layer = layer_weight.length-1 ; weight_layer >= 0 ;
204         weight_layer--) {
205         layer_weight[weight_layer] = Matrix.plus_matrix(layer_weight[weight_layer]
206             ], change_weight[weight_layer]) ;
207     }
208 }
209
210 public void delta_weight_outputnode(int layer){
211     int weight_layer = layer-1;
212     //multiply matrix
213     for (int j = 0; j < error_n.get(error_n.size()-1).length ; j++){
214         double diff_fn = diff_activation_fn(node[layer][j]);

```

```

211         for(int i=0;i< node[layer-1].length ; i++)
212         {
213             double old_weight = moment_rate * change_weight[weight_layer].data[j
214             ][i];
215             double delta_weight = learning_rate * (error_n.get(error_n.size()-1)
216             [j] * diff_fn * activation_fn(node[layer-1][i])) ;
217             double delta = old_weight + delta_weight;
218             change_weight[weight_layer].set(j,i,delta);
219         }
220     }
221     private void local_gradient() {
222         for (int layer = layer_weight.length-1 ; layer >= 0 ; layer--) {
223             for (int j = 0; j < node[layer].length ; j++){
224                 double sum_j = 0;
225                 for(int k=0;k< node[layer+1].length ; k++)
226                 {
227                     sum_j += ( local_gradient_node[layer+1][k]) * layer_weight[
228                     layer].data[k][j] ;
229                 }
230                 // node[layer][j]
231                 double diff_fn = diff_activation_fn(node[layer][j]);
232                 local_gradient_node[layer][j] = sum_j * diff_fn;
233             }
234         }
235     }
236     public void delta_weight_hiddennode(int weight_layer){
237         int node_layer = weight_layer+1;
238         //mutiply matrix
239         for (int j = 0; j < node[node_layer].length ; j++){
240             for(int i=0;i< node[node_layer-1].length ; i++)
241             {
242                 double old_weight = moment_rate * change_weight[weight_layer].data[j
243                 ][i];
244                 double delta_weight = learning_rate * ( local_gradient_node[
245                 node_layer][j] * activation_fn(node[node_layer-1][i])) ;
246                 double delta = old_weight + delta_weight;
247                 change_weight[weight_layer].set(j,i,delta);
248             }
249         }
250     }
251     public void test(ArrayList<Double []> _test_dataset ,ArrayList<Double []>
252     _test_desired_data){
253         //setup input data
254         double t_p =0;
255         double t_n =0;
256         double f_p =0;
257         double f_n =0;
258         for(int test_i = 0; test_i < _test_dataset.size()-1 ; test_i++) {
259             //set dataset value to input node
260             for (int input_i = 0; input_i < neural_type[0]; input_i++) {
261                 node[0][input_i] = _test_dataset.get(test_i)[input_i];
262             }
263             forward_pass();

```

```

265 //         double d = _test_desired_data.get(test_i)[0]*700 ;
266 //         double g = node[node.length-1][0]*700;
267 // System.out.println("desired:" + (int)d + " get: " + g + "\t error_n: " + Math.abs(d
    -g));
268 //         // class set
269 //         if(node[node.length-1][0] > node[node.length-1][1]){
270 //             node[node.length-1][0] = 1.0;
271 //             node[node.length-1][1] = 0.0;
272 //         }else {
273 //             node[node.length-1][0] = 0.0;
274 //             node[node.length-1][1] = 1.0;
275 //         }
276 //         System.out.println("get");
277 //         for (Double val : node[node.length-1]) { System.out.println(val);
    }
278 //         System.out.println("desired");
279 //         for (Double val : _test_desired_data.get(test_i)) { System.out.
    println(val); }
280
281 //         if(node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) &&
    node[node.length-1][0].equals(1.0) ) t_p++;
282 //         if(node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) &&
    node[node.length-1][0].equals(0.0) ) t_n++;
283
284 //         if(!node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) &&
    node[node.length-1][0].equals(1.0) ) f_p++;
285 //         if(!node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) &&
    node[node.length-1][0].equals(0.0) ) f_n++;
286 //     }
287 //     // t_p      t_n      f_p      f_n
288 //     System.out.println(t_p+"\t"+t_n+"\t"+f_p+"\t"+f_n);
289 //     System.out.println( t_p/(t_p+f_p) +"\t"+ t_p/(t_p+f_n) +"\t"+ (t_p+t_n)/(
    t_p+t_n+f_p+f_n)) ;
290 //     error_n.clear();
291 // }
292
293 // public double activation_fn(Double x){
294 //     return Math.max(0.01,x);
295 // }
296
297 // public double diff_activation_fn(Double v ){
298 //     if(v<=0){
299 //         return 0.01;
300 //     }else{
301 //         return 1;
302 //     }
303 // }
304 }

```

3.3 My library

3.3.1 Code of unique_random.java

```
1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.List;
4 public class unique_random {
5     int size;
6     List<Integer> list_number;
7     public unique_random(int size){
8         list_number = new ArrayList<>();
9         this.size = size;
10        for (int i = 0; i < size; i++) {
11            list_number.add(i);
12        }
13        Collections.shuffle(list_number);
14    }
15    public int get_line(){
16        int temp = list_number.get(0);
17        list_number.remove(0);
18        return temp;
19    }
20 }
21 }
```

3.3.2 Code of Matrix.java

```
1 import java.util.Random;
2
3 public class Matrix {
4     double[][] data;
5     int rows,cols;
6
7     /**
8      * Wji weight form input neuron i to j
9      * @param rows j node
10     * @param cols i node
11     */
12
13     public Matrix(int rows, int cols , boolean random){
14         data = new double[rows][cols];
15         this.rows=rows;
16         this.cols=cols;
17         Random generator = new Random(10);
18
19         if(random){
20             for(int j=0;j<rows;j++)
21             {
22                 for(int i=0;i<cols;i++)
23                 {
24                     double ran = 0;
25                     while(ran == 0){
26                         ran = generator.nextDouble(-1,1);
27                         data[j][i]=ran;
28                     }
29                 }
30             }
31         }
32     }
33
34     public static Matrix plus_matrix(Matrix a, Matrix b) {
35         Matrix temp=new Matrix(a.rows,a.cols , false);
36         for(int j=0;j<a.rows;j++)
37         {
38             for(int i=0;i<a.cols;i++)
39             {
40                 temp.data[j][i]=a.data[j][i]+b.data[j][i];
41             }
42         }
43         return temp;
44     }
45
46     public void set(int row, int col, double value) {
47         this.data[row][col] = value;
48     }
49 }
```