Computer Assignment 1 Report by Pongsakorn Rattanapan. Student ID: 630610749

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)

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

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:

```java
public void train(){
    int epoch =0;
    while (N < maxEpoch && avg_error_n > minError){

        for(int data = 0; data < train_dataset.size() ; data++) {
          //setup randomly input data
            int ran_dataset_i = (int) (Math.random() * ((train_dataset.size()) ));

        //set dataset value to input node
        for(int input_i = 0 ; input_i < neural_type[0] ; input_i ++){
            node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
        }

        //cal sum_(input x weight) -> activation_Fn  for each neuron_node
        forward_pass();

        get_error(ran_dataset_i);
        backward_pass();
        }

        double sum = 0.0;
        for (Double[] doubles : error_n) {
            // sum E(n) = 1/2 sum e^2   : sum of squared error at iteration n (sse)
            sum += 0.5*Math.pow(doubles[0], 2);
        }
        // avg_E(n) = 1/N sum E(n)   : avg (sse)
        avg_error_n =  sum / (error_n.size());
        epoch++; // next epoch
        }
    }
```

```
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            // V_j = sum all input*weight i->j + biases
16            sum_input = sum + biases;
17            sum_inputnode[j] = sum_input;
18        }
19        // O_k  =  output of neuron_node k in each layer
20        node[layer+1] = sum_inputnode;
21    }
22 }
```

- 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

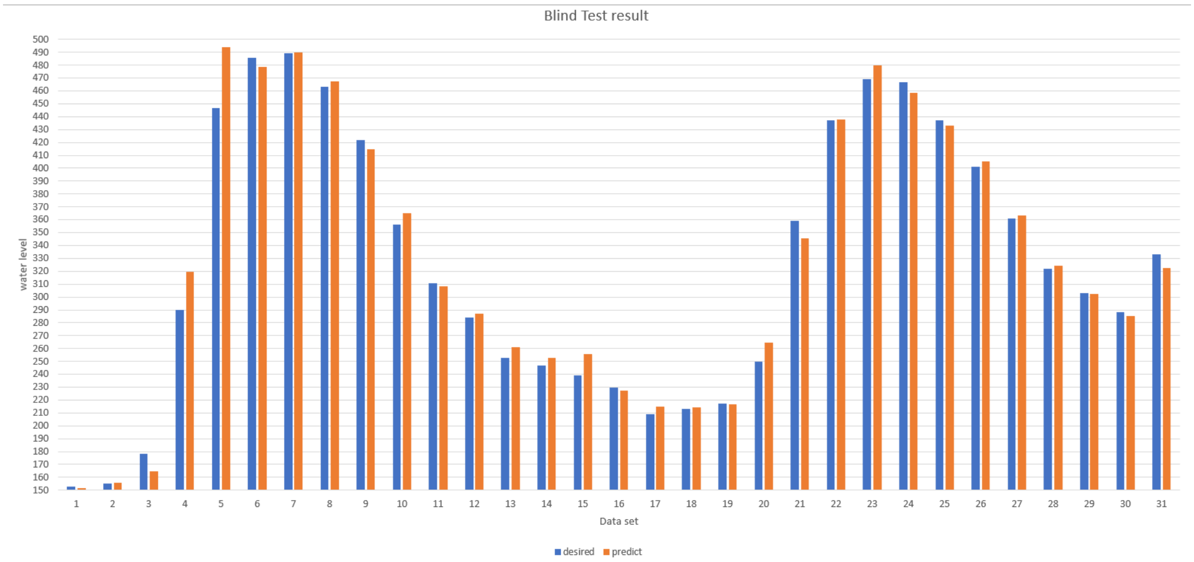
Then neural network change all own weight by this code:

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

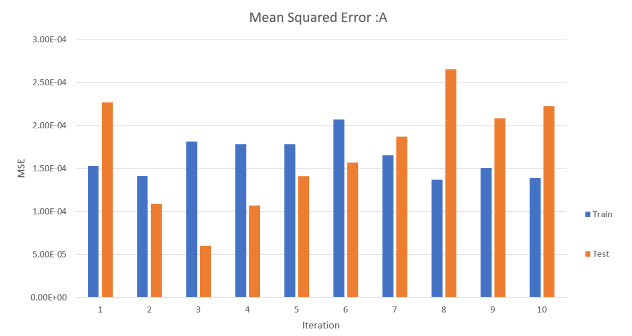
All the source code are in github: https://github.com/min23asdw/neural-network

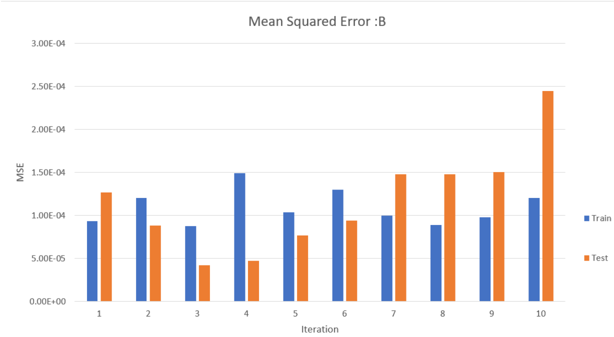
1.5 result

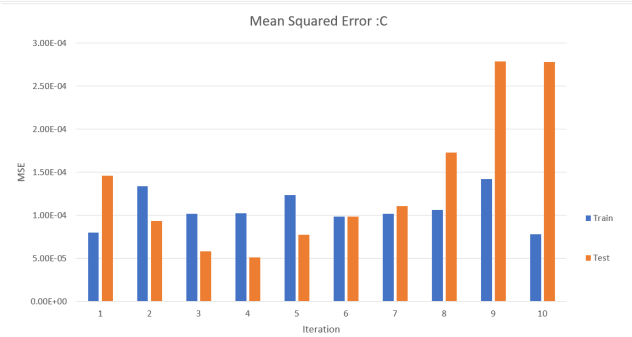
This table is compare between desired and predict Water Level with blind test data-set using neural network :base in [neural test] table



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



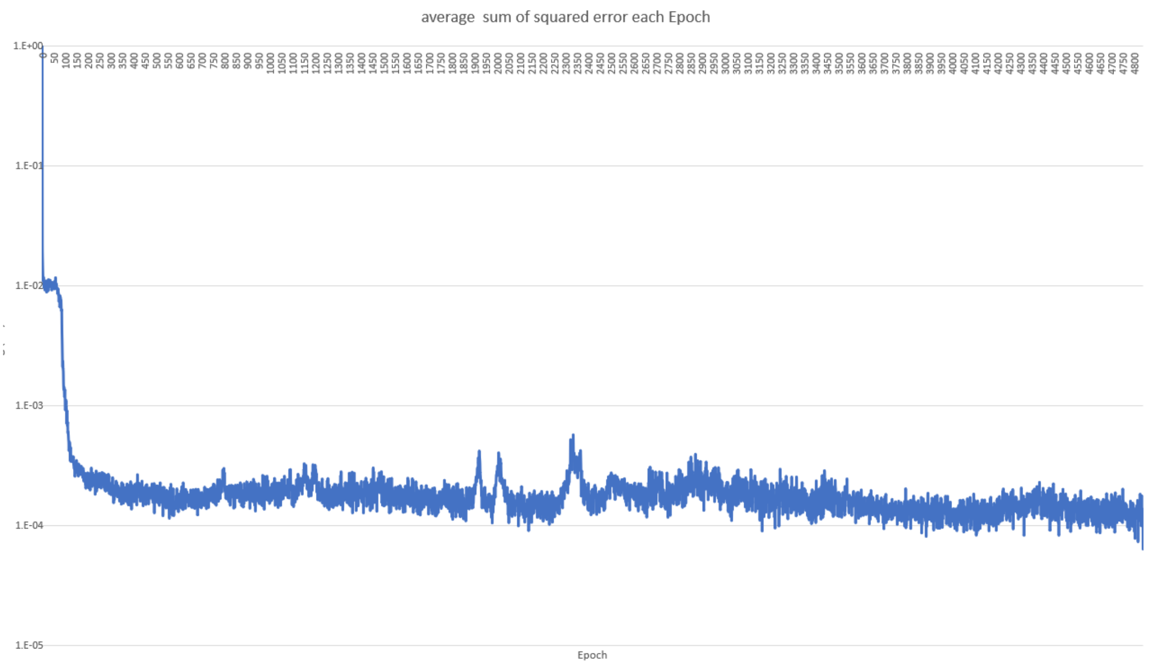




1.6 neural network analysis

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)

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

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 classs $[1, 0]$ else set to classs $[0, 1]$

```
 // class set
 if(output_node[0] > output_node[1]){
     output_node[0] = 1.0;
     output_node[1] = 0.0;
}else {
     output_node[0] = 0.0;
     output_node[1] = 1.0;
}
```

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**)

```
if(output_node == train_desired && output_node[0].equals(1.0) )
    true_positive++;
if(output_node == train_desired && output_node[0].equals(0.0) )
    true_negative++;

if(output_node != train_desired  && output_node[0].equals(1.0) )
    false_positive++;
if(output_node != train_desired  && output_node[0].equals(0.0) )
    false_negative++;
```

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

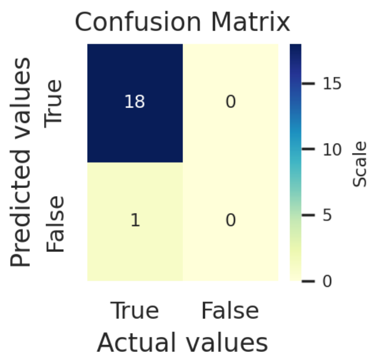
Then calculate Precision,Recall,Accuracy by this formula:

Precision = TPs / (TPs + FPs)
Recall = TPs/(TPs+FNs)
Accuracy = (TPs + TNs) / (TPs+TNs+FPs + FNs)

2.5 result

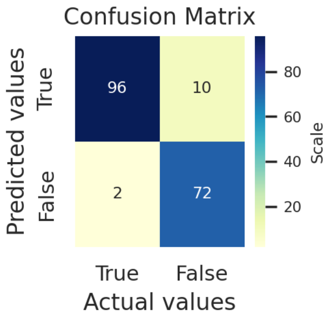
This table show Confusion Matrix with blind test data-set using neural network :base in [neural test] table

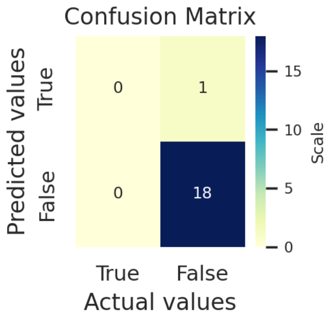
By this table is the result of 10%-cross validation by 10 Iteration show mean squared error of Train and Test form neural network



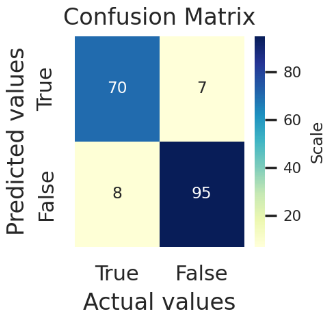
On the left side is training , right side is blind test.

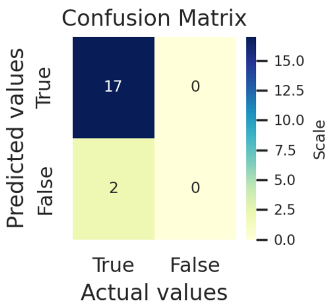
Iteration 0



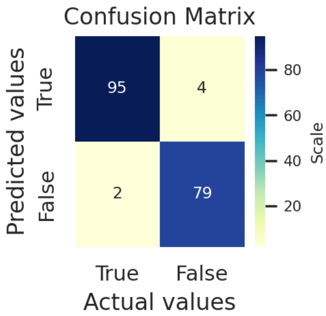


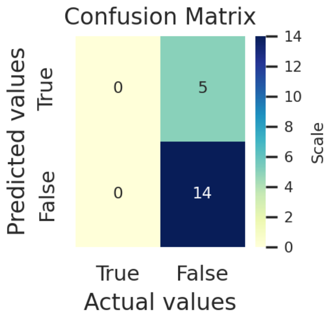
Iteration 1



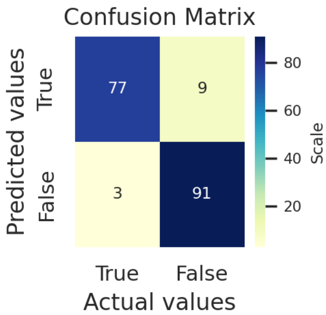


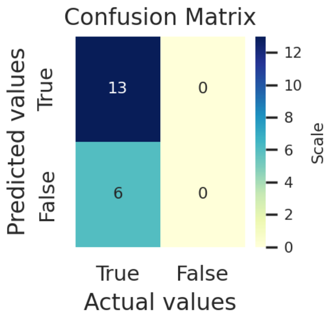
Iteration 2



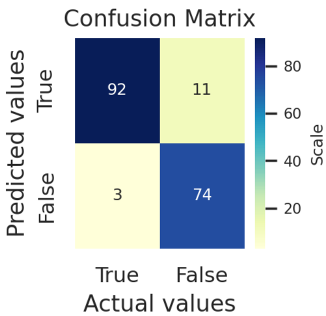


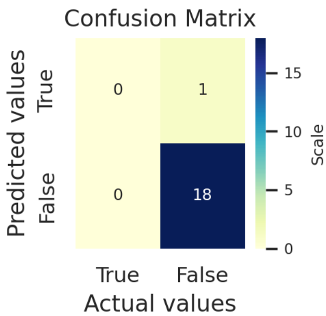
Iteration 3



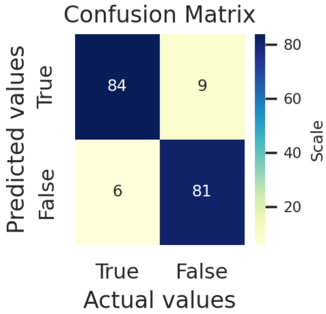


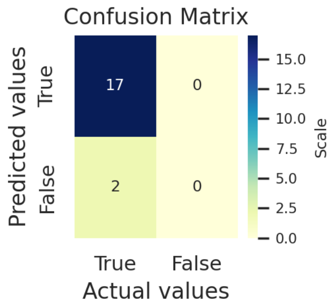
Iteration 4



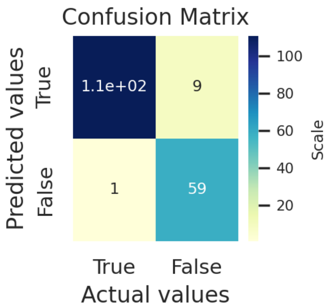


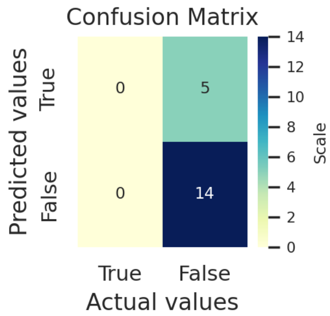
Iteration 5



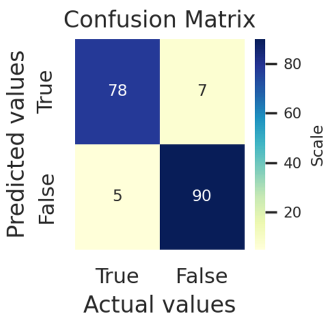


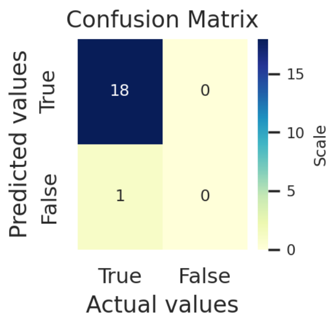
Iteration 6



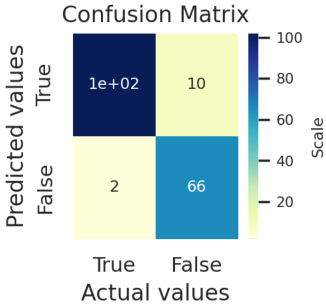


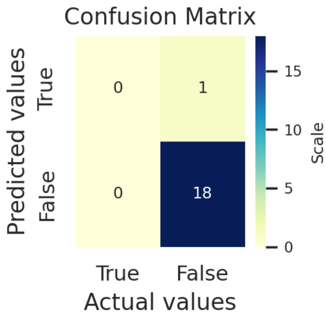
Iteration 7



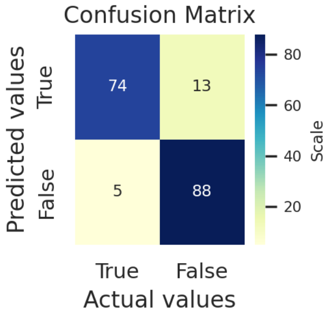


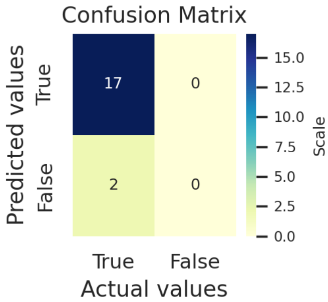
Iteration 8

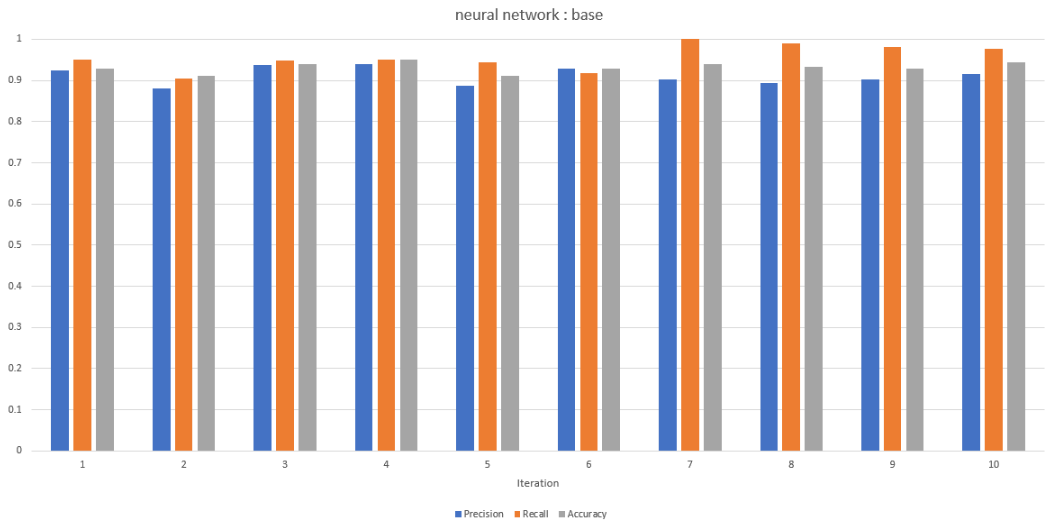


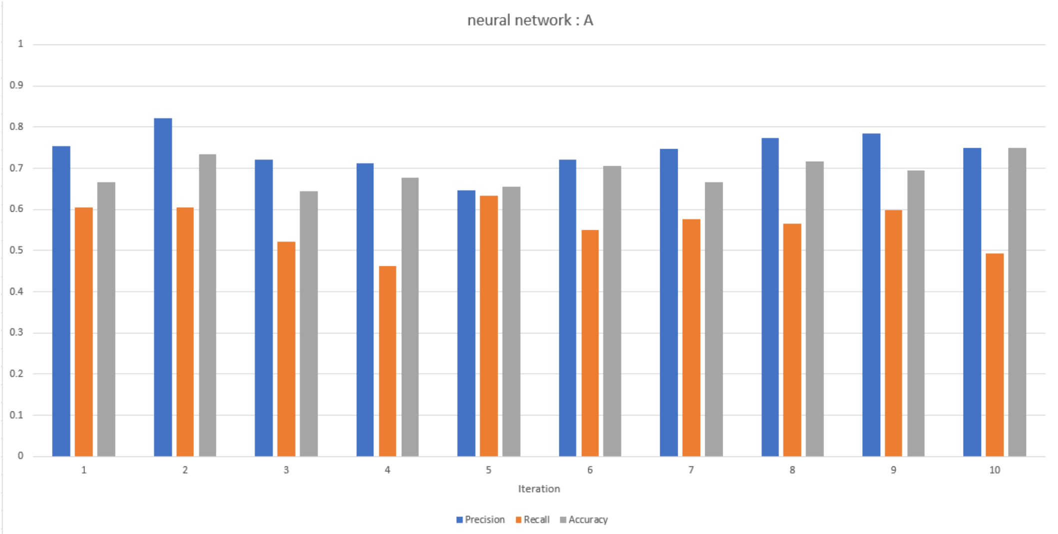


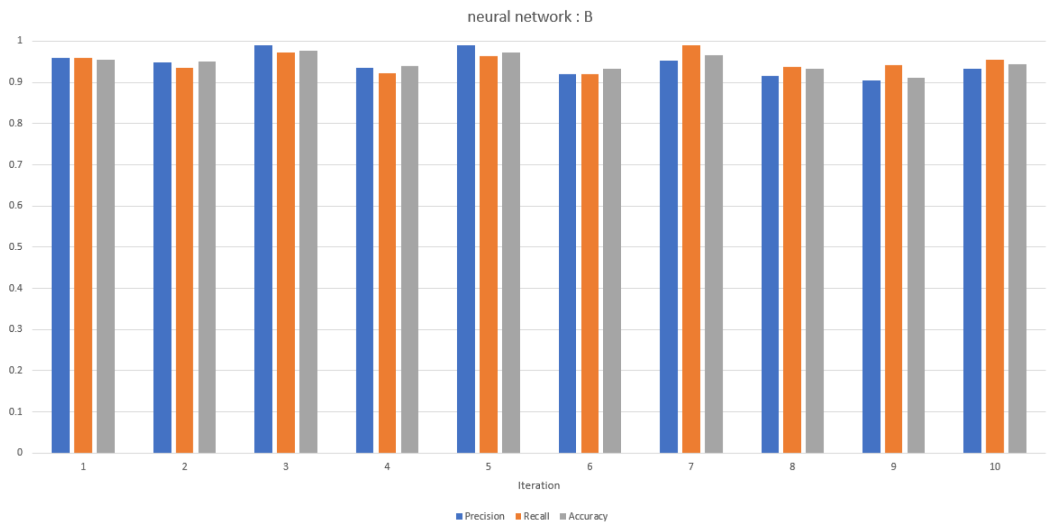
Iteration 9

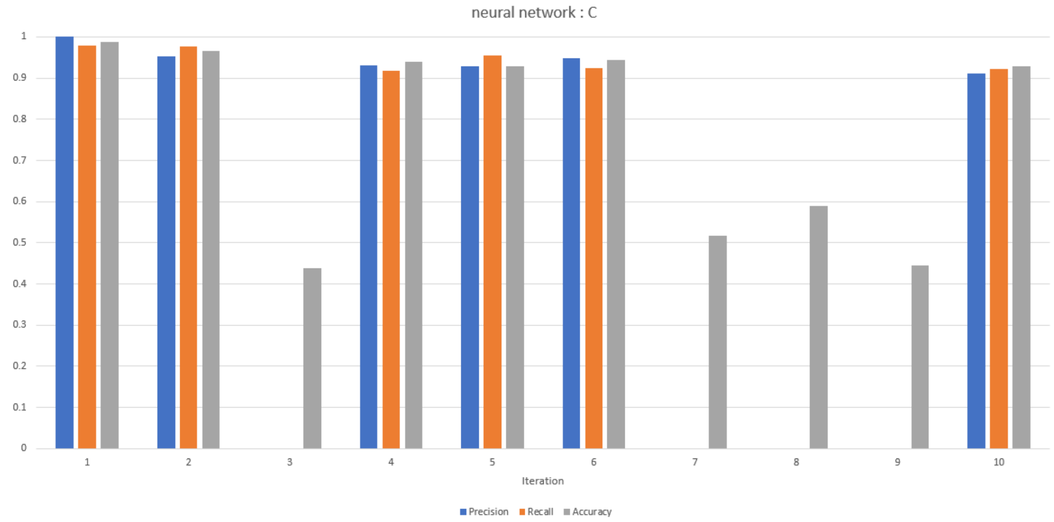







neural network : A
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
6
7
8
9
10
Iteration
Precision
Recall
Accuracy

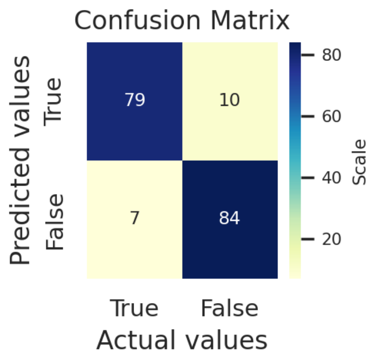

neural network : B
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
6
7
8
9
10
Iteration
Precision
Recall
Accuracy


neural network : C
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
6
7
8
9
10
Iteration
Precision
Recall
Accuracy

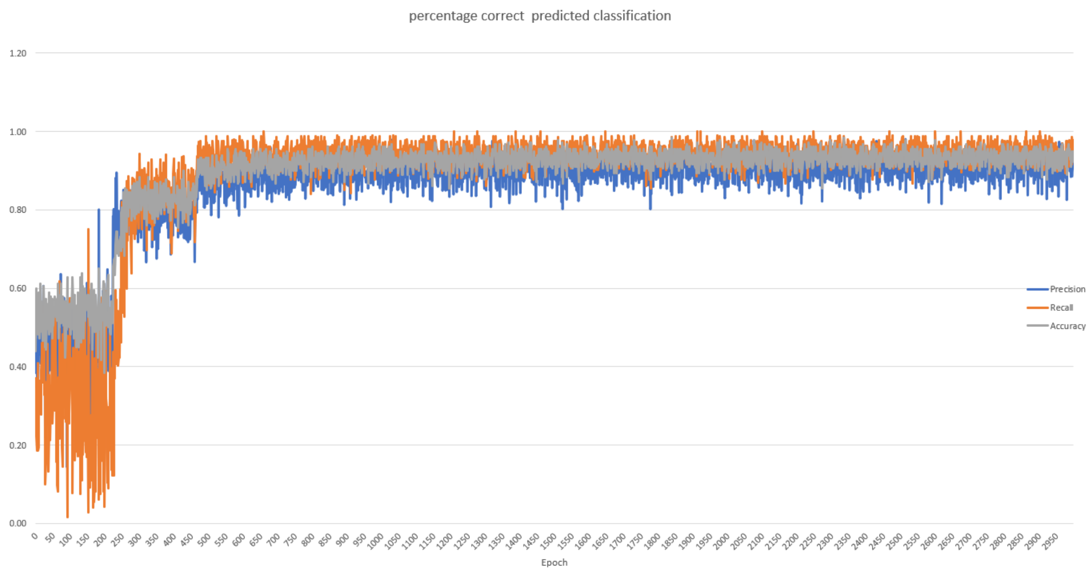
2.6 neural network analysis

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

```java
import java.io.*;
import java.util.ArrayList;

public class main {

    private static ArrayList<ArrayList<Double[]>> test_dataset = new ArrayList<>();
    private static ArrayList<ArrayList<Double[]>> test_desired_data = new ArrayList<>();
    private static ArrayList<ArrayList<Double[]>> train_dataset = new ArrayList<>();
    private static ArrayList<ArrayList<Double[]>> train_desired_data = new ArrayList<>();

    private static int NumberOftest = 10;

    public static void main(String[] args) throws IOException {

        for(int tain_i = 0 ; tain_i < NumberOftest ; tain_i ++) {

            ArrayList<Double[]> test_dataset_i = new ArrayList<>();
            ArrayList<Double[]> test_desired_data_i = new ArrayList<>();
            ArrayList<Double[]> train_dataset_i = new ArrayList<>();
            ArrayList<Double[]> train_desired_data_i = new ArrayList<>();

            FileInputStream fstream = new FileInputStream("src/Flood_dataset.txt");
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String data;

            int line_i = 0;
            while ((data = br.readLine()) != null) { // each line
                String[] tmp = data.split("\t");    //Split space

                Double[] tmp_dataset = new Double[tmp.length-1];
                Double[] tmp_desired_data = new Double[1];

                int word_i = 0;
                for (String t : tmp) {  // each word
                    double tmp_val = (Double.parseDouble(t)/700.0);  // norm

                        if (word_i == tmp.length - 1) { //  desired_data
                            tmp_desired_data[0] = (tmp_val) ;
                        } else {
                            tmp_dataset[word_i] = (tmp_val)  ;
                        }

                    word_i++;
                }

                if (line_i % (10) == tain_i) { // test 10% data
                    test_dataset_i.add(tmp_dataset);
                    test_desired_data_i.add(tmp_desired_data);
                } else { //train 90%
                    train_dataset_i.add(tmp_dataset);
                    train_desired_data_i.add(tmp_desired_data);
                }
```

```
                line_i++;
            }
            test_dataset.add(test_dataset_i);
            test_desired_data.add(test_desired_data_i);
            train_dataset.add(train_dataset_i);
            train_desired_data.add(train_desired_data_i);

        }

        for(int test_i = 1 ; test_i < NumberOftest ; test_i ++) {

            brain b1 = new brain("8,5,5,5,1", 5000, 0.00001, 1, 0.05, 0.9);
            System.out.println("train: " + test_i);
            b1.train( train_dataset.get(test_i), train_desired_data.get(test_i));
            System.out.println("test: " + test_i);
            b1.test(test_dataset.get(test_i), test_desired_data.get(test_i));

        }
    }

}
```

3.1.1 brain code for Problem set 1

```java
import java.util.ArrayList;

public class brain {
    private final int[] neural_type;
    private ArrayList<Double[]> train_dataset;
    private ArrayList<Double[]> train_desired_data;

    private final int maxEpoch;
    private final double minError;
    private final double learning_rate;
    private final double moment_rate;

    //sum of squared error at iteration n (sse)
    private final ArrayList<Double[]> error_n = new ArrayList<>();
    private double avg_error_n = 1000000000 ; // average sse of all epoch
    private double biases; // threshold connected : biases

    private Matrix[] layer_weight  ;
    private Matrix[] change_weight;

    //private Matrix[] old_change_weight;
    private Double[][]  node  ;
    private Double[][]  local_gradient_node  ;

    public brain(String _neural_type , int _maxEpoch , double _minError ,double _biases, double  _learning_rate , double _moment_rate){

        String[] splitArray = _neural_type.split(",");
        int[] array = new int[splitArray.length];
        for (int i = 0; i < splitArray.length; i++) array[i] = Integer.parseInt(splitArray[i]);

        this.neural_type = array;

        init_Structor();

        this.maxEpoch = _maxEpoch;
        this.minError = _minError;
        this.biases = _biases;
        this.learning_rate = _learning_rate;
        this.moment_rate = _moment_rate;
    }

    private void init_Structor(){
        node = new Double[neural_type.length][];
        local_gradient_node = new Double[neural_type.length][];
        for (int i = 0; i < neural_type.length; i++) {
            node[i] = new Double[neural_type[i]];
            local_gradient_node[i] = new Double[neural_type[i]];
        }

        layer_weight = new Matrix[neural_type.length-1];
        change_weight = new Matrix[neural_type.length-1];
        for (int layer = 0; layer < layer_weight.length; layer++) {
            Matrix weight = new Matrix(neural_type[layer+1],neural_type[layer] ,true);
            Matrix change = new Matrix(neural_type[layer+1],neural_type[layer] ,false);
```

```java
            layer_weight[layer] = weight;
            change_weight[layer] = change;
        }

    }

    public void train(ArrayList<Double[]> _train_dataset,ArrayList<Double[]>
        _train_desired_data  ){
        this.train_dataset = _train_dataset;
        this.train_desired_data = _train_desired_data;

        int N =0;
        while (N < maxEpoch && avg_error_n > minError){ //

        unique_random  uq = new unique_random(train_dataset.size());

            for(int data = 0; data < train_dataset.size() ; data++) {
                //random  one dataset
                int ran_dataset_i = uq.get_line();
                //setup dataset value to input node
                for(int input_i = 0 ; input_i < neural_type[0] ; input_i ++){
                    node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
                }

                //cal sum_(input x weight) -> activation_Fn  for each neuron_node
                forward_pass();

                get_error(ran_dataset_i );
                backward_pass();

                double d = train_desired_data.get(ran_dataset_i)[0]*700 ;
                double g = activation_fn( node[node.length-1][0]*700 );
// System.out.println("desired:" +(int)d+"get:"+g+"\t error_n:"+Math.abs(d-g));
            }

            double sum = 0.0;
            for (Double[] doubles : error_n) {
// sum E(n) = 1/2 sum of e^2    : sum of squared error at iteration n (sse)
                sum += 0.5*Math.pow(doubles[0], 2);
            }
            // avg_E(n) = 1/N sum_ E(n)  : avg (sse)
            avg_error_n =  sum / (error_n.size());

            error_n.clear();

            System.out.println( N + " \t  "+ avg_error_n);
            N++; // next epoch
        }

        System.out.println("avg_error_n final : " + avg_error_n);
    }

    private void forward_pass(){
        for(int layer = 0; layer < neural_type.length-1 ; layer++) {

            // W r_c X N r_1 = N+1 r_1
            if(   layer_weight[layer].cols != node[layer].length){
                System.out.println("invalid matrix");
                return;
            }
```

```

            double  sum_input;
            Double[] sum_inputnode = new Double[neural_type[layer+1]];

            //mutiply matrix
            for (int j = 0; j < neural_type[layer+1] ; j++){
                double sum=0;
                for(int k=0;k<node[layer].length;k++)
                {
                    //w_ji : weight from input neuron j to neron i : in each layer
                     sum += layer_weight[layer].data[j][k]  *  activation_fn( node[
                         layer][k])  ;
                }
                // V_j = sum all input*weight i->j + biases
                sum_input = sum + biases;
                sum_inputnode[j] = sum_input;
            }
            // O_k  =  output of neuron_node k in each layer
            node[layer+1] = sum_inputnode;
        }
    }

    private void get_error(int ran_dataset_i  ) {

         int number_outputn_node  =   node[node.length-1].length;
        Double[] errors = new Double[number_outputn_node];
        for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++) {
    //train_desired_data => d_j desired output for neuron_node j at iteration N
    // it have "one data"
    //e_j  = error at neuron j at iteration N
            double desired = train_desired_data.get(ran_dataset_i)[0];
            errors[outnode_j] =  desired -  activation_fn( node[node.length-1][
                outnode_j] )  ;

            double diff_fn  = diff_activation_fn(node[node.length-1 ][outnode_j]);
                node[node.length-1 ][outnode_j]
            local_gradient_node[node.length-1][outnode_j] =  errors[outnode_j] *
                diff_fn;
        }
        error_n.add(errors);
    }


    private void backward_pass() {
        //diff_sse/w_j =  diff_sum_(n) / diff_e_j  *  diff_e_j / diff_Y_j  *
            diff_Y_j / diff_V_j  *  diff_V_j / diff_w_ji
        //diff_sse/w_j = (e_j(n))  *  -1  * diff_Y_j(sum_input) * Y_i

        // delta_weight_ji = - learning rate ( diff_sse/w_j )
        // delta_weight_ji  =  learning rate [ (e_j(n)) * diff_Y_j(sum_input) * Y_i ]
        // add Momentum
        // delta_weight_ji = delta_weight_ji(old) + delta_weight_ji
        // wji_next = wji_now + delta_weight_ji

        // output change_weight
        int output_layer = node.length-1;
         delta_weight_outputnode(output_layer);

        //local gradient output_k= e_k * diff Y_k    ::    local gradient hidden_j =
            diff Y_j * sum_ (  W_kj  *  l_g k)
```

```java
        local_gradient();
        for (int layer = node.length-3 ; layer >= 0  ; layer--) {
            // hidden layer change_weight
            // delta_weight_ji =   learning rate *  local_gradient_j * Y_i
            delta_weight_hiddennode(layer);
        }

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



    public void delta_weight_outputnode(int layer){
     int weight_layer = layer-1;

     //mutiply matrix
     for (int j = 0; j < error_n.get(error_n.size()-1).length ; j++){

      double diff_fn  = diff_activation_fn(node[layer][j]);
       for(int i=0;i< node[layer-1].length ; i++)
        {
        double old_weight = moment_rate * change_weight[weight_layer].data[j][i];
        double delta_weight = learning_rate * (error_n.get(error_n.size()-1)[j] * diff_fn * activation_fn(node[layer-1][i])) ;
        double delta =  old_weight  +  delta_weight;
        change_weight[weight_layer].set(j,i,delta);
        }
      }

    }
    private void local_gradient() {
        for (int layer = layer_weight.length-1 ; layer >= 0  ; layer--) {
            for (int j = 0; j < node[layer].length   ; j++){
                double sum_j = 0;
                for(int k=0;k< node[layer+1].length  ; k++)
                {
                sum_j +=  ( local_gradient_node[layer+1][k])  *  layer_weight[layer].data[k][j] ;
                }
                // node[layer][j]
                double diff_fn  = diff_activation_fn(node[layer][j]);
                local_gradient_node[layer][j] = sum_j * diff_fn;
            }
        }
    }
    public void delta_weight_hiddennode(int weight_layer){

        int node_layer = weight_layer+1;

        //mutiply matrix
        for (int j = 0; j <  node[node_layer].length ; j++){
            for(int i=0;i< node[node_layer-1].length ; i++)
            {
                double old_weight =  moment_rate * change_weight[weight_layer].data[j][i];
                double delta_weight = learning_rate * ( local_gradient_node[
```

```
                    node_layer][j] * activation_fn(node[node_layer-1][i])) ;
                double delta = old_weight  +  delta_weight;

                change_weight[weight_layer].set(j,i,delta);
            }
        }
    }

    public void test(ArrayList<Double[]> _test_dataset,ArrayList<Double[]>
        _test_desired_data){

        //setup input data
        for(int test_i = 0; test_i < _test_dataset.size()-1 ; test_i++) {

            //set dataset value to input node
            for (int input_i = 0; input_i < neural_type[0]; input_i++) {
                node[0][input_i] = _test_dataset.get(test_i)[input_i];
            }

            forward_pass();

            int number_outputn_node  =   node[node.length-1].length;
            Double[] errors = new Double[number_outputn_node];
            for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++)
                {
                double desired = _test_desired_data.get(test_i)[0];
                errors[outnode_j] =  desired -node[node.length-1][outnode_j];
            }
            error_n.add(errors);

            double d = _test_desired_data.get(test_i)[0]*700 ;
            double g = node[node.length-1][0]*700;
            System.out.println("desired:" + (int)d + " get: "+ g + "\t error_n: " +
                Math.abs(d-g));
        }

        double sum = 0.0;
        for (Double[] doubles : error_n) {
            // sum_E(n) = 1/2 sum_e^2   : sum of squared error at iteration n (sse)
            sum += 0.5*Math.pow(doubles[0], 2);
        }
        // avg_E(n) = 1/N  sum_E(n)  : avg (sse)
        avg_error_n =  sum / (error_n.size());

        System.out.println("avg_error_n final : " + avg_error_n);
        error_n.clear();
    }

    public double activation_fn(Double x){
        return Math.max(0.01,x);
    }

    public double diff_activation_fn(Double v ){
        if(v<=0){
            return 0.01;
        }else{
            return 1;
        }
    }
}
```

3.2 Main code for Problem set 2

```
import java.io.*;
import java.util.ArrayList;

public class main_2 {

    private static ArrayList<ArrayList<Double[]>> test_dataset = new ArrayList<>();
    private static ArrayList<ArrayList<Double[]>> test_desired_data = new ArrayList<>();

    private static ArrayList<ArrayList<Double[]>> train_dataset = new ArrayList<>();
    private static ArrayList<ArrayList<Double[]>> train_desired_data = new ArrayList<>();

    private static int NumberOftest = 10;

    public static void main(String[] args) throws IOException {
        for(int tain_i = 0 ; tain_i < NumberOftest ; tain_i ++) {
            ArrayList<Double[]> test_dataset_i = new ArrayList<>();
            ArrayList<Double[]> test_desired_data_i = new ArrayList<>();

            ArrayList<Double[]> train_dataset_i = new ArrayList<>();
            ArrayList<Double[]> train_desired_data_i = new ArrayList<>();

            FileInputStream fstream = new FileInputStream("src/cross.pat");
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String data;

            int line_i = 1;
            while ((data = br.readLine()) != null) { // each line
                if(line_i%3 == 0 || (line_i+1)%3 == 0) { // not p line
                String[] eachLine = data.split("\\s+");
                Double[] temp = new Double[eachLine.length];

                for(int  i =0 ; i<eachLine.length;i++){
                    Double dataNum = Double.parseDouble(eachLine[i]);
                    temp[i] = dataNum ;
                }

                if (line_i % 3 == 0) {  // line desired
                    if (line_i % 10 == tain_i) {  // 10% for test
                        test_desired_data_i.add(temp);
                    } else
                        train_desired_data_i.add(temp);
                } else if ((line_i + 1) % 3 == 0) { // line input
                    if ((line_i + 1) % 10 == tain_i) { // 10% for test
                        test_dataset_i.add(temp);
                    } else
                        train_dataset_i.add(temp);
                }
            }
                line_i++;
            }
            test_desired_data.add(test_desired_data_i);
            train_desired_data.add(train_desired_data_i);
            test_dataset.add(test_dataset_i);
            train_dataset.add(train_dataset_i);
        }
```

```java


        for(int test_i = 0 ; test_i < NumberOftest ; test_i ++) {
            brain_2 b2 = new brain_2("2,8,2", 3000, 0.00007, 1, 0.01, 0.1);
            System.out.println("train: " + test_i);
            b2.train( train_dataset.get(test_i), train_desired_data.get(test_i));
            System.out.println("test: " + test_i);
            b2.test(test_dataset.get(test_i), test_desired_data.get(test_i));
        }
    }
}
```

3.2.1 brain code for Problem set 2

```
import java.util.ArrayList;

public class brain_2 {
    private final int[] neural_type;
    private ArrayList<Double[]> train_dataset;
    private ArrayList<Double[]> train_desired_data;

    private final int maxEpoch;
    private final double minError;
    private final double learning_rate;
    private final double moment_rate;

    private final ArrayList<Double[]> error_n = new ArrayList<>(); //sum of squared
        error at iteration n (sse)
    private double avg_error_n = 1000000000 ; // average sse of all epoch
    private double biases; // threshold connected : biases

    private Matrix[] layer_weight  ;
    private Matrix[] change_weight;

    private Double[][]  node  ;
    private Double[][]  local_gradient_node  ;

    public brain_2(String _neural_type , int _maxEpoch , double _minError ,double
        _biases, double  _learning_rate , double _moment_rate){

        String[] splitArray = _neural_type.split(",");
        int[] array = new int[splitArray.length];
        for (int i = 0; i < splitArray.length; i++) array[i] = Integer.parseInt(
            splitArray[i]);
        this.neural_type = array;

        init_Structor();

        this.maxEpoch = _maxEpoch;
        this.minError = _minError;
        this.biases = _biases;
        this.learning_rate = _learning_rate;
        this.moment_rate = _moment_rate;
    }
    private void init_Structor(){
        node = new Double[neural_type.length][];
        local_gradient_node = new Double[neural_type.length][];
        for (int i = 0; i < neural_type.length; i++) {
            node[i] = new Double[neural_type[i]];
            local_gradient_node[i] = new Double[neural_type[i]];
        }

        layer_weight = new Matrix[neural_type.length-1];
        change_weight = new Matrix[neural_type.length-1];
        for (int layer = 0; layer < layer_weight.length; layer++) {
            Matrix weight = new Matrix(neural_type[layer+1],neural_type[layer] ,true)
                ;
            Matrix change = new Matrix(neural_type[layer+1],neural_type[layer] ,false
                );
            layer_weight[layer] = weight;
            change_weight[layer] = change;
        }
```

```
    }

    public void train(ArrayList<Double[]> _train_dataset,ArrayList<Double[]>
        _train_desired_data  ){
        this.train_dataset = _train_dataset;
        this.train_desired_data = _train_desired_data;

        int N =0;
        while (N < maxEpoch && avg_error_n > minError){ //
            double true_positive =0;
            double true_negative =0;
            double false_positive =0;
            double false_negative =0;

            unique_random  uq = new unique_random(train_dataset.size());

            for(int data = 0; data < train_dataset.size() ; data++) {
               //random  one dataset
                int ran_dataset_i = uq.get_line();
                //setup dataset value to input node
                for(int input_i = 0 ; input_i < neural_type[0] ; input_i ++){
                    node[0][input_i] = train_dataset.get(ran_dataset_i)[input_i];
                }

                //cal sum_(input x weight) -> activation_Fn  for each neuron_node
                forward_pass();

                get_error(ran_dataset_i );
                backward_pass();

//                double d = train_desired_data.get(ran_dataset_i)[0]*700 ;
//                double g = activation_fn( node[node.length-1][0]*700 );
//                System.out.println("desired:" + (int)d + " get: "+ g + "\t error_n:
    " + Math.abs(d-g));

                // class set
                 if(node[node.length-1][0] > node[node.length-1][1]){
                    node[node.length-1][0] = 1.0;
                    node[node.length-1][1] = 0.0;
                }else {
                    node[node.length-1][0] = 0.0;
                    node[node.length-1][1] = 1.0;
                }

//                System.out.println("get");
//                for (Double val : node[node.length-1]) {
//                    System.out.println(val);
//                }
//                System.out.println("desired");
//                for (Double val : train_desired_data.get(ran_dataset_i)) {
//                    System.out.println(val);
//                }
//                System.out.println("++");

                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++;
                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++;

```

```
                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++;
                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++;

            }
            //Precision = true_positive/true_positive+false_positive
            // Recall = true_positive/(true_positive+false_negative)
            // Accuracy = (true_positive+true_negative)/(true_positive+true_negative+
                false_positive+false_negative)
            if(N==maxEpoch-1){
                // true_positive       true_negative     false_positive
                    false_negative
                System.out.println(true_positive+"\t"+true_negative+"\t"+
                    false_positive+"\t"+false_negative);
                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)) ;
            }
            error_n.clear();
            N++; // next epoch
        }
    }

    private void forward_pass(){
        for(int layer = 0; layer < neural_type.length-1 ; layer++) {

            // W r_c X N r_1 = N+1 r_1
            if(   layer_weight[layer].cols != node[layer].length){
                System.out.println("invalid matrix");
                return;
            }

            double  sum_input;
            Double[] sum_inputnode = new Double[neural_type[layer+1]];

            //mutiply matrix
            for (int j = 0; j < neural_type[layer+1] ; j++){
                double sum=0;
                for(int k=0;k<node[layer].length;k++)
                {
                    //w_ji : weight from input neuron j to neron i : in each layer
                    sum += layer_weight[layer].data[j][k]  *  activation_fn( node[
                        layer][k])  ;
                }
                // V_j = sum all input*weight i->j + biases
                sum_input = sum + biases;
                sum_inputnode[j] = sum_input;
            }
            // O_k  =  output of neuron_node k in each layer
            node[layer+1] = sum_inputnode;

        }
    }

    private void get_error(int ran_dataset_i  ) {

```

```java
        int number_outputn_node  =   node[node.length-1].length;
        Double[] errors = new Double[number_outputn_node];
        for ( int outnode_j = 0 ; outnode_j < number_outputn_node ; outnode_j++) {
            //train_desired_data => d_j desired output for neuron_node j at iteration N // it have "one data"
            //e_j  = error at neuron j at iteration N
            double desired = train_desired_data.get(ran_dataset_i)[outnode_j];
            errors[outnode_j] =  desired -  activation_fn( node[node.length-1][outnode_j] )  ;

            double diff_fn  = diff_activation_fn(node[node.length-1 ][outnode_j]);
            local_gradient_node[node.length-1][outnode_j] =  errors[outnode_j] * diff_fn;
        }
        error_n.add(errors);
    }


    private void backward_pass() {
        //diff_ sse/w_j =  diff_ sum_(n) / diff_ e_j  *  diff_ e_j / diff_ Y_j  *  diff_ Y_j / diff_ V_j  *  diff_ V_j / diff_ w_ji
        //diff_ sse/w_j =     (e_j(n))     *           -1    * diff Y_j(sum_input)  * Y_i

        // Y_j is  linear_fn
        //diff Y_j =  linear_fn -> 1

        // delta_weight_ji = -learning rate (  diff_ sse/w_j )
        // delta_weight_ji  = learning rate [ (e_j(n)) * diff Y_j(sum_input) * Y_i ]
        // delta_weight_ji = delta_weight_ji(old) + delta_weight_ji
        // wji_next = wji_now + delta_weight_ji

        // output change_weight
        int output_layer = node.length-1;
        delta_weight_outputnode(output_layer);

        //local gradient output_k= e_k * diff Y_k    ::    local gradient hidden_j = diff Y_j * sum_  (  W_kj  *  l_g k)
        local_gradient();
        for (int layer = node.length-3 ; layer >= 0  ; layer--) {
            // hidden layer change_weight
            // delta_weight_ji =    learning rate *  local_gradient_j * Y_i
            delta_weight_hiddennode(layer);
        }

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



    public void delta_weight_outputnode(int layer){
        int weight_layer = layer-1;
        //mutiply matrix
        for (int j = 0; j < error_n.get(error_n.size()-1).length ; j++){

            double diff_fn  = diff_activation_fn(node[layer][j]);
```

```
            for(int i=0;i< node[layer-1].length ; i++)
            {
                double old_weight =  moment_rate * change_weight[weight_layer].data[j][i];
                double delta_weight =  learning_rate * (error_n.get(error_n.size()-1)[j] * diff_fn * activation_fn(node[layer-1][i])) ;
                double delta =  old_weight  +  delta_weight;
                change_weight[weight_layer].set(j,i,delta);
            }
        }
    }
    private void local_gradient() {
        for (int layer = layer_weight.length-1 ; layer >= 0  ; layer--) {
            for (int j = 0; j < node[layer].length   ; j++){

                double sum_j = 0;
                for(int k=0;k< node[layer+1].length  ; k++)
                {
                    sum_j +=  ( local_gradient_node[layer+1][k])  *  layer_weight[layer].data[k][j] ;
                }
                                                  // node[layer][j]
                double diff_fn  = diff_activation_fn(node[layer][j]);
                local_gradient_node[layer][j] = sum_j * diff_fn;
            }
        }
    }
    public void delta_weight_hiddennode(int weight_layer){
        int node_layer = weight_layer+1;
        //mutiply matrix

        for (int j = 0; j <  node[node_layer].length ; j++){
            for(int i=0;i< node[node_layer-1].length ; i++)
            {
                double old_weight =  moment_rate * change_weight[weight_layer].data[j][i];
                double delta_weight = learning_rate * ( local_gradient_node[node_layer][j] * activation_fn(node[node_layer-1][i])) ;
                double delta = old_weight  +  delta_weight;

                change_weight[weight_layer].set(j,i,delta);
            }
        }
    }

    public void test(ArrayList<Double[]> _test_dataset,ArrayList<Double[]> _test_desired_data){

        //setup input data
        double t_p =0;
        double t_n =0;
        double f_p =0;
        double f_n =0;
        for(int test_i = 0; test_i < _test_dataset.size()-1 ; test_i++) {

            //set dataset value to input node
            for (int input_i = 0; input_i < neural_type[0]; input_i++) {
                node[0][input_i] = _test_dataset.get(test_i)[input_i];
            }
            forward_pass();
```

```java
//              double d = _test_desired_data.get(test_i)[0]*700 ;
//              double g = node[node.length-1][0]*700;
// System.out.println("desired:" + (int)d + " get: "+ g + "\t error_n: " + Math.abs(d-g));
            // class set
            if(node[node.length-1][0] > node[node.length-1][1]){
                node[node.length-1][0] = 1.0;
                node[node.length-1][1] = 0.0;
            }else {
                node[node.length-1][0] = 0.0;
                node[node.length-1][1] = 1.0;
            }
//              System.out.println("get");
//                  for (Double val : node[node.length-1]) {  System.out.println(val);  }
//              System.out.println("desired");
//                  for (Double val : _test_desired_data.get(test_i)) { System.out.println(val);  }

            if(node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) && node[node.length-1][0].equals(1.0) ) t_p++;
            if(node[node.length-1][0].equals(_test_desired_data.get(test_i)[0]) && node[node.length-1][0].equals(0.0)  ) t_n++;

            if(!node[node.length-1][0].equals(_test_desired_data.get(test_i)[0])  && node[node.length-1][0].equals(1.0)  ) f_p++;
            if(!node[node.length-1][0].equals(_test_desired_data.get(test_i)[0])  && node[node.length-1][0].equals(0.0)  ) f_n++;
        }
        // t_p       t_n    f_p   f_n
        System.out.println(t_p+"\t"+t_n+"\t"+f_p+"\t"+f_n);
        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)) ;
        error_n.clear();
    }

    public double activation_fn(Double x){
        return Math.max(0.01,x);
    }

    public double diff_activation_fn(Double v ){
        if(v<=0){
            return 0.01;
        }else{
            return 1;
        }
    }
}
```

3.3 My library

3.3.1 Code of unique_random.java

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class unique_random {
    int size;
    List<Integer> list_number;
    public unique_random(int size){
        list_number = new ArrayList<>();
        this.size = size;
        for (int i = 0; i < size; i++) {
            list_number.add(i);
        }
        Collections.shuffle(list_number);

    }
    public int get_line(){
        int temp = list_number.get(0);
        list_number.remove(0);
        return temp;
    }
}
```

3.3.2 Code of Matrix.java

```java
import java.util.Random;

public class Matrix {
    double[][] data;
    int rows,cols;

    /**
     * W ji weight form input neuron   i to j
     * @param rows j node
     * @param cols i node
     */

    public Matrix(int rows, int cols , boolean random){
        data = new double[rows][cols];
        this.rows=rows;
        this.cols=cols;
        Random generator = new Random(10);

        if(random){
            for(int j=0;j<rows;j++)
            {
                for(int i=0;i<cols;i++)
                {
                    double ran = 0;
                    while(ran == 0){
                        ran = generator.nextDouble(-1,1);
                        data[j][i]=ran;
                    }
                }
            }
        }
    }

    public static Matrix plus_matrix(Matrix a, Matrix b) {
        Matrix temp=new Matrix(a.rows,a.cols , false);
        for(int j=0;j<a.rows;j++)
        {
            for(int i=0;i<a.cols;i++)
            {
                temp.data[j][i]=a.data[j][i]+b.data[j][i];
            }
        }
        return temp;
    }

    public void set(int row, int col, double value) {
      this.data[row][col] = value;
    }
}
```