Hopfield Network Model

Yanmeng Kong (Anny)[1], Xinyue Zheng (Dora)[1], Pengfei He (Murphy)[1]

[1] Department of Applied Mathematics, University of Washington, Seattle, WA

Abstract

Hopfield Neural Network is a simple model of biological cognitive processes. In this paper, we will discuss how the Hopfield Neural Network operates. We analyze the dynamics to show how to store information that is learned in the stable point; we demonstrate how to build each digit as a matrix using the array data structure.

1 Problem

In real-life scenario, we always encounter situations when having a hard time recognizing one's handwritten, or fascinating by an item but do not know how to search it in online store. Our group focus on the problem of how to convert these noisy memories into some complete memories so that people can easy view or make use of the result. We choose to use Hopfield Network Model since it is commonly used where pattern reorganization is useful, for example, image recognition and restoration.

Hopfield consists of interconnect neurons which can update their activation values either synchronously or asynchronously, and every unit acts as an input and an output of the network. The activation values are either 1 or -1. Each unit is dependent with all other units, which means a unit i will be influenced by another unit j under a certain weight w_{ij}, and possibly threshold. Training and recall are two main operation modes. In the training mode, the key-value pair is experiencing transformation from x_k to y_k. In general , y_k will return the result best matches the input x. This function can be interpret as the process to eliminate the residual between x_k and input x, or convert an incomplete x_k diagram to a complete diagram of x. Either way is being considered as the key function of biological brain. In our project, we decide to choose updating nodes synchronous at each state.

Hopfield network was a ground-breaking creation in the field of neural network and gave an essential dynamism to latter research. A lot of applications are used in computer science and

real time systems because of their flexible ability to learn by example as well as their parallel architectures.

2 Mathematical methods

2.1 Problem in the mathematical model

Typically hopfield network make use of its associative memory like human memory to store patterns and use them to recognize or restore images. And images are transformed to a set of 1s and 0s to represents black and white colors which can be differentiated. In our test program, we process sequences of 1s and 0s and return a restored version based on the pattern we already have.

2.2 Basic function and notations

The activation state can be calculated through the function below:

$$x_i(t+1) = sign(\sum_{j=1}^{n} x_j(t) w_{ij} - \theta_i) \tag{1}$$

or

$$X = sign(WX - T) \tag{2}$$

X, sign, W and T are defined as following:

X is input data which is the activation value of the n units/neurons:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ . \\ x_n \end{bmatrix}$$

every x_i represents a neuron which is a node in a hopfield network graph.

$$sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

You can also view it in Figure1 on Page 5.

W is the symmetric matrix of weight:

$$W = \begin{bmatrix} w_{11} & w_{12} & ... & w_{1n} \\ w_{21} & w_{22} & ... & w_{2n} \\ . & . & . & . \\ w_{n1} & . & . & w_{nn} \end{bmatrix}$$

It is worth noting that $w_{ij} = w_{ji}$ which is undirected edges in between node i and node j in the hopfield network graph. The weight on the edge affect both sides of data. T is the threshold of each unit:

$$T = \begin{bmatrix} \theta_1 \\ \theta_2 \\ . \\ \theta_n \end{bmatrix}$$

Threshold can be understood as a external conditions. It varies to affect the input value of sign function, and then activation state would be affected. We can represent Hopfield Network by treating each unit as a vertex, the weighted edge between the pair vertices is w_{ij} in W. Each unit is binary, usually represented as 0 and 1 or -1 and 1 according to the relationship with others and the weight on edges, the units will move from one place to another and finally reach a stable state. It is worth noting that our program is based on 0 and 1.

3 Computational methods

Generally speaking, we will converge the energy of Hopfield Network to a local state and thus generate a energy function for this certain state. The energy function can be defined as E There are two options depending on whether considering the threshold or not, we decide to chose to analysis with the influence of a threshold. Therefore an artificial unit will be added and linked to the others with the negative value of their threshold. During the process, if one unit is selected, it can either stay the same or change to the opposite ($1 \rightarrow -1$ or $-1 \rightarrow 1$). This will be done by computing energy function. Each operation will cause energy to decrease and when it reach a state that the energy cannot decrease anymore, that is its stable state. We will be including graph demonstrating how neurons move under unstable state and eventually reach its stable state at the local minimum of the energy function.

3.1 Assumption

In real life, we often need to restore images to its original state, and it is usually impossible to restore it. Then we have to rely on some patterns that the picture might have, and try to use the pattern to complete it. And this is exactly what hopfield network do. From the prospective of electronic media, pictures are composed of pixels. And in our case, we just use the color combination of black and white since they are enough to differentiate different meanings such as numbers and characters. To be more simplified, we use 0s and 1s to represent the pattern in our program, and those binary data are efficient to process data and less expensive to store in the memory. Also, given two updating techniques of synchronization and synchronization, we chose the synchronous updating rule to simplify the restoration as much as possible. All picture data can be considered a 1-D array, and we will train them into a 2-D pattern in a matrix form. Plus, there is also tanh function to make the updating continuously instead of discretely. Given the limited time and the purpose of the project, it appears less necessary to do so. After all those considerations, we choose the synchronous updating rule and sign function as our starting point of hopfield network project.

3.2 Process of learning pattern

Suppose we have one eight-length sequence of data,

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

and each individual x is the neuron. We want to learn the pattern of the data, or say all connections. We will set up a 8x8 symmetric matrix to stand for their synapses. Set M_{ii} to zero for all is because it's meaningless for connection of a node to its own. We will write M as matrix and X as the array for short. Then we compute the M[1][2] = X[1] * X[2] = 1*(-1) = -1, M[1][3]= X[1]*[3] = 1*1 =1, and so on so forth until we get the full matrix. As the matrix shows below.

$$\begin{pmatrix} 0 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 0 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 0 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 0 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 0 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 0 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 & 1 & 0 \end{pmatrix}$$

In our program, users can either input one matrix or a set of matrices as the graph shows, and then we need average them to find a suitable pattern for latter restoration.

3.3 Process of restoration

To begin with, it randomly pick one neuron as its target updating neuron. And we calculate the sum of the product of edge connected to the neuron and another side of node's weight. We will call this sum "field" in the later essay to represent that of the updating neuron. After applying sign function to simplify it to +1 or -1. As the graph shows, any input value larger than zero, the result would be 1. On the opposite, any input value less than zero, the result would be 0.

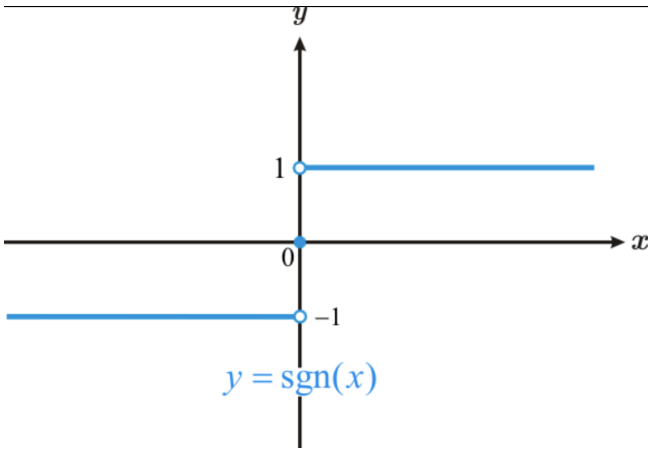

Figure 1: Sign function

We will look if it matches it original neuron state. If they are not same, then we change the sign to the opposite. We will call it "flip" in the later passage. If they are same, then there is no need to change. This single update will also affect the total energy which will be elaborated in the following subsection. After all updates, the input data should be restored to the state same as the pattern. Ideally, we should update the nodes asynchronously to avoid the oscillation problem. However, in practice, we currently find that the synchronously updating has the same effect, and we will analyze it in the "improvement section".

Take the graph for example(Figure 2), red edges are -1, and blue edges are +1. Yellow nodes are +1, black nodes are -1. If we want to update the red node above the yellow node. We will find four pairs of edge and vertex adjacent to it. First, we multiply the "red edge" and "the yellow node". Second, we multiply the "blue edge" and the "black node". Third, we multiply the "red edge" and the "black node". Forth, we multiply the "blue edge" and the "black node". Finally, we get the sum of (-1)+(-1)+(-1)+(-1) = -2. Then we get sgn(-2) = -1 which does not correspond to its original state. Then we "flip" it to the positive state which is

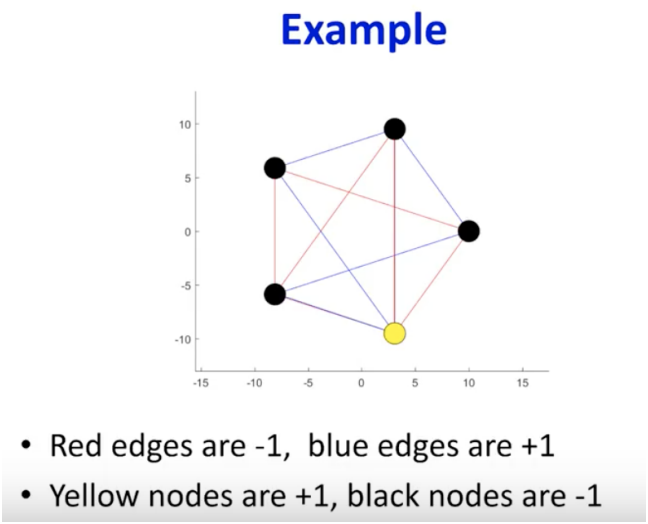

Figure 2: Example before updating

represented by yellow node(Figure3).

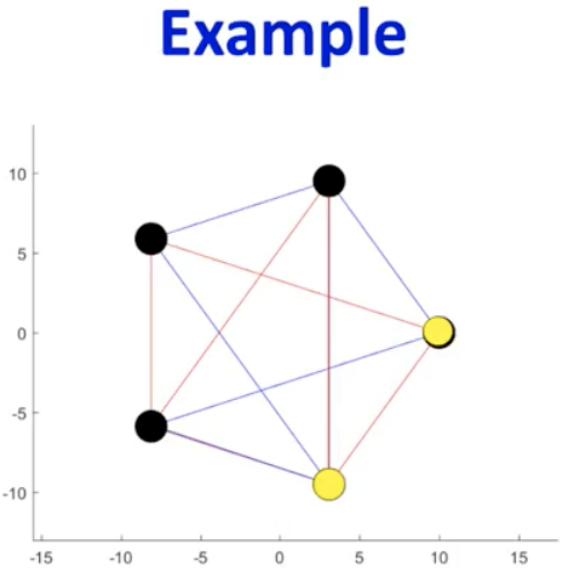

Figure 3: Example after updating

3.4 Energy function

Energy for individual node is $E_i = -\frac{1}{2}h_i x_i$. The whole energy function is

$$h = \sum_i -\frac{1}{2}h_i x_i = \sum_{ij} -\frac{1}{2}w_{ij}x_i x_j.$$

The final aim is to find the minima energy which makes it to stable state.

As the graph shows, there is a large energy gap or say update from the maxima to minima.

$\Delta D(y_k) = D(y_1, ..., y_k^+, ..., y_N) - D(y1, ..., y_k^-, ..., y_N)$. It represents the sum of all updates which is the total energy gap. y^- means the node state before each update. Accordingly, y^+ means the node after each update. To be more specific, the effect on energy gap of every single update on the individual node should be treated in two different cases. In the first case, the y^- matches the "field" of the node. Therefore, this node is currently stable, and the energy gap does not need to change. $y_i^+(\sum_{j\neq i} w_{ij} + b_i) - y_i^-(\sum_{j\neq i} w_{ij} + b_i) = 0$. In the second case, y_i^- does not match the "field" of the node. $y_i^+(\sum_{j\neq i} w_{ij} + b_i) - y_i^-(\sum_{j\neq i} w_{ij} + b_i) = 2y_i^+(\sum_{j\neq i} w_{ij} + b_i)$. In this way, the energy gap increases. As the updates keep going, the sum of those energy gap would gain increasingly. Now, we define the enery of the network as $E = -\sum_{i,j<i} w_{ij}y_iy_j - \sum_i b_iy_i$. Therefore, the evolution of a hopfield network constantly decreases its energy. We can also understand it physically and graphically that a ball fall into the minima in the contour, and its potential energy keep decreasing.

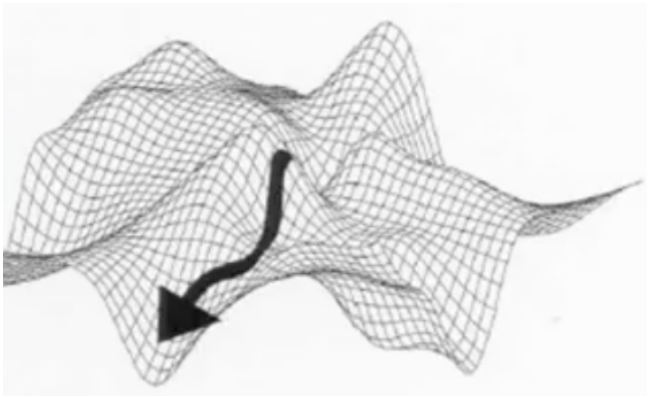

Figure 4: Graphic Decreasing Energy

4 Solution

Our solution has two main step: Training (store the digit in to memory as a standard) and Restoration (pass in a noisy image and test which system it converge to).

4.1 How to store the training pattern

First pass in the training pattern which is represented by a 1*N row matrix consisting only "0" and "1". Then obtain a bipolar matrix associate with that input row matrix. We can get the contribution matrix of this input by multiplying the transpose of its bipolar matrix

with its bipolar matrix and subtract the Identity matrix. After obtaining the computational matrix, we want to update the current weight matrix that already stored in the memory.

New current weight matrix = contribution matrix + current weight matrix

This is how training pattern is stored in memory- by updating and store in the new weight matrix.

Below is the Java method we wish to show about how to train a pattern.

```
/**
         * Train an input digit pattern.
         * @param inputDigitPattern input digit pattern.
         * @throws Exception
         */
        public void train(double[] inputDigitPattern) throws Exception {
                // Transform input digit pattern to a bipolar pattern.
                double[] bipolarInput = toBipolar(inputDigitPattern);
                // Build bipolar input as a matrix.
                Matrix bipolarMatrix = Matrix.toRowMatrix(bipolarInput);
                // Transpose the bipolar matrix: (bipolar matrix)^T.
                Matrix transposedBipolarMatrix = bipolarMatrix.transpose();
                // (Transposed bipolar matrix) x (bipolar matrix).
                Matrix crossProductMatrix = transposedBipolarMatrix.multiply(bipolarMatrix);
                // Trained matrix = (3) - (Identity Matrix).
                Matrix trainedMatrix = crossProductMatrix.subtract(Matrix.identity(weightMatrix.getData().length));

                // Show mathematical process for getting trained matrix if mode = SHOW_MATH.
                if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                        System.out.println("#-- train --#");
                        System.out.println("#-- Calculate the Trained Matrix --#");
                        System.out.println("1) Get the bipolar matrix \n" + bipolarMatrix);
                        System.out.println("2) Transpose the bipolar matrix:\n"+ transposedBipolarMatrix);
                        System.out.println("3) (Transposed bipolar matrix) x (bipolar matrix):\n"+ crossProductMatrix);
                        System.out.println("4) Trained matrix = (3) - (Identity Matrix):\n"+ trainedMatrix);
                        System.out.println("<-- Update Weight Matrix -->");
                        System.out.println("current weight matrix:\n" + weightMatrix.toString("N", "N"));
                }

                // Update weight matrix.
                weightMatrix = weightMatrix.add(trainedMatrix);

                // Show mathematical process if mode = SHOW_MATH.
                if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                        System.out.println("Updated Weight Matrix = (Trained Matrix) + (Current Weight Matrix)\n"
                                        + weightMatrix.toString("N","N"));
                }
        }
```

4.2 How to match the given input with the pattern in the memory

Firstly, do the same as above, pass in a test pattern which is a 1× N row matrix. Obtain the bipolar matrix for the input and acquire the current weight matrix stored in the memory. Calculate the dot product of bipolar matrix with each of the column in current weight matrix.

If the resulting scalar is larger than zero, the replace the original index by 1, else if result is smaller or equal to zero, replacing the value in that index as 0.

$$\text{index[0]} = \text{(bipolar matrix)} \cdot \text{(weight matrix column 00)}$$

$$\text{index[1]} = \text{(bipolar matrix)} \cdot \text{(weight matrix column 01)}$$

$$\ldots$$

$$\text{index[i]} = \text{(bipolar matrix)} \cdot \text{(weight matrix column i)}$$

Finally, we print out the resulting row matrix graphics as the output which is same as pattern.

Below is the Java method we wish to show about how to recognize an input digit.

```
/**
     * Recognize an input digit pattern.
     * @param inputDigitPattern input digit pattern
     * @return an output pattern.
     */
    public double[] recognize(double[] inputDigitPattern) {
        // Transform the input digit pattern to a bipolar pattern.
        double[] bipolarInput = toBipolar(inputDigitPattern);
        // Create a output digit pattern.
        double[] outputDigitPattern = new double[inputDigitPattern.length];
        // Build the bipolar input as a matrix.
        Matrix bipolarMatrix = Matrix.toRowMatrix(bipolarInput);

        // Show mathematical process for recognizing an input digit pattern.
        if (DigitRecognizer.mode == Mode.SHOW_MATH) {
            System.out.println("#-- recognize --#");
            System.out.println("1) Weight matrix:\n" + weightMatrix.toString("N", "N"));
            System.out.println("2) Get the bipolar matrix for input \n" + bipolarMatrix);
            System.out.println("3) dot product bipolar matrix & each of the columns in weight matrix");
        }

        // Updating nodes.
        for (int i = 0; i < inputDigitPattern.length; i++) {
            try {
                // Create a column matrix.
                Matrix columnMatrix = weightMatrix.getColumnMatrix(i);
                double dotProductResult = bipolarMatrix.dotProduct(columnMatrix);

                // Show mathematical process for dot product.
                if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                    System.out.print("[3."+ String.format("%02d", i) +
                                    "] (bipolar matrix) . (Weight matrix column "+
                                    String.format("%02d", i)+") = ");
                }

                // Update weights.
                if (dotProductResult > 0) {
                    outputDigitPattern[i] = 1.00;
                    if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                        System.out.println(" "+dotProductResult + "  > 0  ==>  1");
                    }
                } else {
                    outputDigitPattern[i] = 0;
```

```
                        if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                            System.out.println(dotProductResult + " <= 0  ==>  0");
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            return outputDigitPattern;
        }
```

5 Improvement

5.1 Problem of spurious minima

Each time we try to memorize a pattern, we hope to create a new energy minimum. And given the limited memory based on the hopfield memory rule, it is possible that two nearby minima merge(Figure5) to generate a minimum at an intermediate location(Figure6). And it limits the capacity of the network. If we need more dense pattern in our data like Chinese character recognition, it is a critical problem. Plus, we find that after we input a fair amount of data, the pattern will mixed up and no longer print out the expected output. In our opinion, to avoid this, we need to have the initially processed neuron randomly processed asynchronously. Also, the updating strategy leads to the second problem.

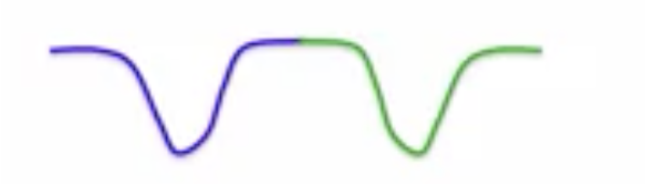

Figure 5: Normal spurious minima

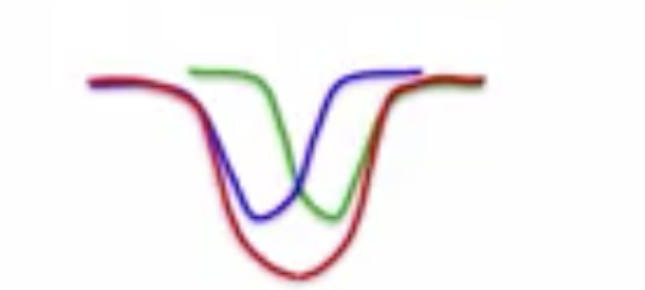

Figure 6: Abnormal spurious minima

5.2 Problem of oscillations

As I stated before, our model process all input data synchronously which might cause oscillation that the period is two. To be more clear, our nodes are computed simultaneously which make the neuron state change in a period of two. For example, in the first phase of oscillation(Figure7), we want to update the two neurons. And because the edge's weight is abnormally low as it shows, the "field" will be negative. Then two neurons has to be updated to zero(Figure8). Later on, the negative edges have no effect on the "field" because the other side node is zero. Then the two "field" are updated to 5 back again. Since no other conditions are changed during the process. This process will keep going permanently and decrease the energy tremendously and abnormally. To solve this problem, we need random updating neurons and different interval of updating. Hopefully, we will implement those techniques in the future project.

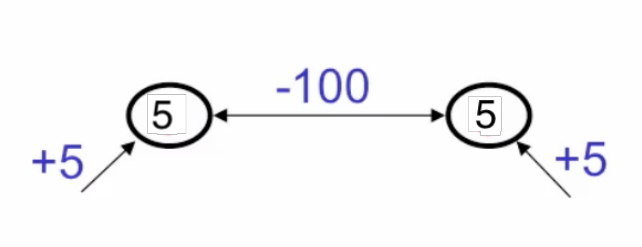

Figure 7: Oscillation phase1

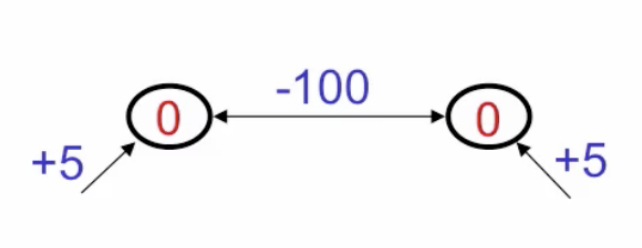

Figure 8: Oscillation phase2

6 Conclusion

Hopfield Neural Network has been used for storing and recalling digit patterns in our program due to its associative pattern. Since we update the node at each state synchronously, which may cause the result that some of the situation may never lead to a stable network state.

While asynchronously updating rule guarantee to reach a stable state under finite iteration time due to the nature of recurrent of network. Thus, asynchronous updating rule will generate a better biological model. However, both biological process is not perfect and will encounter errors when dealing with larger and more complex data other than just digit. In our programming, if we only store the first five digit as training data, we can get most patterns trained to a network. However, as the amount of training data increase, the accuracy decrease tremendously. But the neural network needs a large amount of training data to be accurate. Given the limitation of memory storage of hopfield network and our simple updating rule, our program cannot handle the pattern correctly after too many inputs. Therefore, we will improve the updating rule, the updating interval and possibly the tanh function to adjust the activation values as the purpose changes in the future project.

7 Reference

Raul Rojas. Neural Networks - A Systematic Introduction. 1996

Hertz, J., Krogh, A., Palmer, R.G. (1991). Introduction to the theory of neural computation. Redwood City, CA: Addison-Wesley.

Jehoshua Bruck. On the convergence properties of the hopfield model. Procceedings of the IEEE, 78(10), October 1990

J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. Vol. 79, pp. 2554-2558, April 1982

8 Appendix

8.1 DigitRecognizer.java

```
package amath383;
```

```
import java.util.Scanner;

/**
 * Command Line Interface:
 *   This application needs jdk8 installed on user machine
 */
public class DigitRecognizer {
        /**
         * users can select between DEFAULT and SHOW_MATH mode
         *
         * DEFAULT - show digit recognition result only
         * SHOW_MATH - show digit recognition step by step
         */
        public enum Mode {
                DEFAULT,  // Clean Mode
                SHOW_MATH  // Math Mode - to see what is behind the scene
        };

        public static Mode mode = Mode.DEFAULT;  // default mode
        public static final int NUM_OF_ROWS = 5;  // number of rows for the digit pattern
        public static final String[] TRAIN_PATTERNS =
                {"11111001100110011111", "01100110011001101111",
                 "11110001111110001111", "11110001111100011111",
                 "10011001111100010001", "11111000111100011111",
                 "11111000111110011111", "11110001000100010001",
                 "11111001111110011111", "11111001111100010001"};

        public static void main(String[] args) throws Exception {
                // Print welcome words and ask for number of columns.
                showWelcome();

                // Read input, number of columns.
                Scanner scan = new Scanner(System.in);
                int numOfcols = Integer.parseInt(scan.nextLine());
                int numOfNodes = numOfcols * NUM_OF_ROWS;  // num of nodes

                // Create Hopfield Network with given number of columns,
                // and initialize input and output digit patterns.
                Hopfield hopfield = new Hopfield(numOfNodes);  // hopfield for training

                boolean flag = true;
                while (flag) {
                        // Show available options.
                        showOptions();

                        // Take an input option and translate to a command.
                        String in = scan.nextLine();
                        String command = toCommand(in);

                        // Do the corresponding command.
                        switch (command) {
                                case "train" :
                                        train(hopfield, numOfNodes, scan);
                                        break;
                                case "recognize":
                                        recognize(hopfield, numOfNodes, scan);
                                        break;
                                case "recognizeAsyn":
                                        recognizeAsyn(hopfield, numOfNodes, scan);
                                case "clear":
                                        clear(hopfield);
                                        break;
```

```
                        case "select mode":
                                selectMode(scan);
                                break;
                        case "exit":
                                flag = false;
                                break;
                        case "alreadyTrained":
                                alreadyTrained(hopfield, numOfNodes);
                                break;
                    }
                }
                scan.close();
                System.exit(0);
        }


        private static void alreadyTrained(Hopfield hopfield, int numOfNodes) throws Exception {
                for (int i = 0; i < 10; i++) {
                        double[] inputDigitPattern =
                                        getInput(TRAIN_PATTERNS[i], numOfNodes);
                        hopfield.train(inputDigitPattern);
//                      System.out.print(Matrix.getMatrix(inputDigitPattern, NUM_OF_ROWS).toPackedString());
                }
                System.out.println("---------train success----------\n");
        }
        /**
         * Select Mode.
         * @param scan scanner
         */
        private static void selectMode(Scanner scan) {
                System.out.println("> Select mode: ");
                System.out.println("> 1) DEFAULT");
                System.out.println("> 2) SHOW_MATH");
                if (scan.nextLine().equalsIgnoreCase("SHOW_MATH") || scan.equals("2")) {
                        mode = Mode.SHOW_MATH;
                } else {
                        mode = Mode.DEFAULT;
                }
        }

        /**
         * Clear the weight matrix.
         * @param hopfield
         */
        private static void clear(Hopfield hopfield) {
                hopfield.getWeightMatrix().clear();
                System.out.println("------- Weight matrix cleared --------");
        }

        /**
         * Recognize the input digit pattern.
         * @param hopfield Hopfield model
         * @param numOfNodes number of nodes
         * @param scan scanner
         * @throws Exception
         */
        private static void recognizeAsyn(Hopfield hopfield, int numOfNodes, Scanner scan) throws Exception {
                System.out.println("> Provide input pattern: ");
                double[] inputDigitPattern = getInput(scan.nextLine(), numOfNodes);
                double[] onputDigitPattern = hopfield.updateAsyn(inputDigitPattern);
                System.out.println("Input pattern:");
                System.out.print(Matrix.getMatrix(inputDigitPattern, NUM_OF_ROWS).toPackedString());
                System.out.println("Output pattern:");
```

```
            System.out.print(Matrix.getMatrix(onputDigitPattern, NUM_OF_ROWS).toPackedString());
        }

        /**
         * Recognize the input digit pattern.
         * @param hopfield Hopfield model
         * @param numOfNodes number of nodes
         * @param scan scanner
         * @throws Exception
         */
        private static void recognize(Hopfield hopfield, int numOfNodes, Scanner scan) throws Exception {
            System.out.println("> Provide input pattern: ");
            double[] inputDigitPattern = getInput(scan.nextLine(), numOfNodes);
            double[] onputDigitPattern = hopfield.recognize(inputDigitPattern);
            System.out.println("Input pattern:");
            System.out.print(Matrix.getMatrix(inputDigitPattern, NUM_OF_ROWS).toPackedString());
            System.out.println("Output pattern:");
            System.out.print(Matrix.getMatrix(onputDigitPattern, NUM_OF_ROWS).toPackedString());
        }

        /**
         * Train the input digit pattern.
         * @param hopfield Hopfield model
         * @param numOfNodes number of nodes
         * @param scan scanner
         * @throws Exception
         */
        private static void train(Hopfield hopfield, int numOfNodes, Scanner scan) throws Exception {
            System.out.println(
                    "> Please give your training digit pattern " + "(" + numOfNodes + ") : ");
            double [] inputDigitPattern =
                            getInput(scan.nextLine(), numOfNodes);
            hopfield.train(inputDigitPattern);
            System.out.print(Matrix.getMatrix(inputDigitPattern, NUM_OF_ROWS).toPackedString());
            System.out.println("----------train success----------\n");
        }

        /**
         * Show options available.
         */
        private static void showOptions() {
            System.out.println("> Select options: ");
            System.out.println("> 1) train");
            System.out.println("> 2) recognize");
//          System.out.println("> 3) recognizeAsyn");
            System.out.println("> 4) clear");
            System.out.println("> 5) select mode");
            System.out.println("> 6) exit");
            System.out.println("> 7) alreadyTrained");
        }

        /**
         * Translates input to the corresponding command:
         *      1) train
         *  2) recognize
         *  3) recognizeAsyn
         *  4) clear
         *  5) select mode
         *  6) exit
         *  7) alreadyTrained
         * @param in input
         * @return "" if no command matches,
         *                  the corresponding command.
```

```
         */
        private static String toCommand(String in) {
                if (in.equalsIgnoreCase("train") || in.equals("1")) {
                        return "train";
                } else if (in.equalsIgnoreCase("recognize") || in.equals("2")) {
                        return "recognize";
                } else if (in.equalsIgnoreCase("recognizeAsyn") || in.equals("3")) {
                        return "recognizeAsyn";
                } else if (in.equalsIgnoreCase("clear") || in.equals("4")) {
                        return "clear";
                } else if (in.equalsIgnoreCase("select mode") || in.equals("5")) {
                        return "select mode";
                } else if (in.equalsIgnoreCase("exit") || in.equals("6")) {
                        return "exit";
                } else if (in.equalsIgnoreCase("alreadyTrained") || in.equals("7")) {
                        return "alreadyTrained";
                }
                return "";
        }

        /**
         * Show welcome instructions.
         */
        private static void showWelcome() {
                System.out.println("> ==========================================================");
                System.out.println("> |        # Amath 383 Hopfield Digit Recognizer #         |");
                System.out.println("> ==========================================================");
                System.out.println(">");
                System.out.println("> Input digit pattern format: ");
                System.out.println(">       [# of nodes] = [# of columns] x [5 rows]    ");
                System.out.println(">     ");
                System.out.println(">           # of columns   ");
                System.out.println(">            ------------   ");
                System.out.println("> #    1|  x x x ... x x x ");
                System.out.println("> of   2|  x x x ... x x x ");
                System.out.println("> rows 3|  x x x ... x x x ");
                System.out.println(">      4|  x x x ... x x x ");
                System.out.println(">      5|  x x x ... x x x ");
                System.out.println(">");
                System.out.println("> Please Enter # of columns (recommend 4): ");
        }

        /**
         * @precon input.length() = size
         * @return an array out of input
         */
        private static double[] getInput(String input, int size) throws Exception {
                if (input.length() != size) {
                        throw new Exception("Input length ("
                                        + input.length()
                                        + ") does not match size("
                                        + size + ")");
                }
                double[] result = new double[size];
                for (int i = 0; i < size; i++) {
                        result[i] = Double.parseDouble("" + input.charAt(i));
                }
                return result;
        }
}
/*
40 neurons sample test data ==>
0001100000011000000110000001100011111111
```

```
11111111000000111111111111000000011111111
11111111000000111111111110000001111111111

20 neurons sample test data ==>
01100110011001101111
11110011111111001111
11110011111100111111
10011001100110010000
*/
```

8.2 Hopfield.java

```
package amath383;

import amath383.DigitRecognizer.Mode;

/**
 * <b>Hopfield</b> uses a hopfield neural network Model
 * to train, and recognize input digit patterns.
 */
public class Hopfield {
        private Matrix weightMatrix;

        /**
         * Constructor to create a hopfield object with a weight matrix.
         * @param numOfNodes number of nodes.
         */
        public Hopfield(int numOfNodes) {
                weightMatrix = new Matrix(numOfNodes, numOfNodes);
        }

        /**
         * Get the weight matrix.
         * @return the weight matrix.
         */
        public Matrix getWeightMatrix() {
                return weightMatrix;
        }

        /**
         * Train an input digit pattern.
         * @param inputDigitPattern input digit pattern.
         * @throws Exception
         */
        public void train(double[] inputDigitPattern) throws Exception {
                // Transform input digit pattern to a bipolar pattern.
                double[] bipolarInput = toBipolar(inputDigitPattern);
                // Build bipolar input as a matrix.
                Matrix bipolarMatrix = Matrix.toRowMatrix(bipolarInput);
                // Transpose the bipolar matrix: (bipolar matrix)^T.
                Matrix transposedBipolarMatrix = bipolarMatrix.transpose();
                // (Transposed bipolar matrix) x (bipolar matrix).
                Matrix crossProductMatrix = transposedBipolarMatrix.multiply(bipolarMatrix);
                // Trained matrix = (3) - (Identity Matrix).
                Matrix trainedMatrix = crossProductMatrix.subtract(Matrix.identity(weightMatrix.getData().length));

                // Show mathematical process for getting trained matrix if mode = SHOW_MATH.
                if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                        System.out.println("#-- train --#");
                        System.out.println("#-- Calculate the Trained Matrix --#");
                        System.out.println("1) Get the bipolar matrix \n" + bipolarMatrix);
                        System.out.println("2) Transpose the bipolar matrix:\n"+ transposedBipolarMatrix);
```

```
                System.out.println("3) (Transposed bipolar matrix) x (bipolar matrix):\n"+ crossProductMatrix);
                System.out.println("4) Trained matrix = (3) - (Identity Matrix):\n"+ trainedMatrix);
                System.out.println("<-- Update Weight Matrix -->");
                System.out.println("current weight matrix:\n" + weightMatrix.toString("N", "N"));
            }

            // Update weight matrix.
            weightMatrix = weightMatrix.add(trainedMatrix);

            // Show mathematical process if mode = SHOW_MATH.
            if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                System.out.println("Updated Weight Matrix = (Trained Matrix) + (Current Weight Matrix)\n"
                        + weightMatrix.toString("N","N"));
            }
    }

//RecallPattern
public double[] updateAsyn(double[] inputDigitPattern){
            // Transform the input digit pattern to a bipolar pattern.
            double[] bipolarInput = toBipolar(inputDigitPattern);
            // weight matrix
    double[][] w = weightMatrix.getData();
    // value matrix
    double[] v = bipolarInput;
    boolean doWhile=true;
    int iteration=1;
    int n = w.length;

    //Recalling
    while (doWhile) {
        int stateChange = 0;
        System.out.println("----\nRecall of " + iteration + "-th state,");
        for (int i = 0; i < n; i++) {
            int v_new = 0, net = 0;
            //Calculate each net[i]
            for (int j = 0; j < n; j++) {
                    //w[i][j]*v[j]
                net += w[i][j] * v[j];
            }
            //Next state of v[i]
            if (net >= 0) {
                v_new = 1;
            }
            if (net < 0) {
                v_new = 0;
            }

            if (v_new != v[i]) {
                stateChange++;
                v[i] = v_new;
            }
        }
        System.out.println("Energy:" + EnergyFunction(v) + "\n----");

        //if Converge(stable)?
        if (stateChange == 0) {
            doWhile = false;
        }
        iteration++;
    }
    return v;
}
```

```
/**
 * Energy function
 * @param inputDigitPattern
 * @return
 */
private double EnergyFunction(double[] inputDigitPattern){
    double energy = 0;
    double[][] w = weightMatrix.getData();
    int n = w.length;
    double[] v = inputDigitPattern;

    for (int a = 0; a < n; a++){
        for (int b = 0; b < n; b++){
            if (a != b){
                    energy += w[a][b] * v[a] * v[b];
            }
        }
    }
    return ((-0.5) * energy);
}

    /**
     * Recognize an input digit pattern.
     * @param inputDigitPattern input digit pattern
     * @return an output pattern.
     */
    public double[] recognize(double[] inputDigitPattern) {
            // Transform the input digit pattern to a bipolar pattern.
            double[] bipolarInput = toBipolar(inputDigitPattern);
            // Create a output digit pattern.
            double[] outputDigitPattern = new double[inputDigitPattern.length];
            // Build the bipolar input as a matrix.
            Matrix bipolarMatrix = Matrix.toRowMatrix(bipolarInput);

            // Show mathematical process for recognizing an input digit pattern.
            if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                    System.out.println("#-- recognize --#");
                    System.out.println("1) Weight matrix:\n" + weightMatrix.toString("N", "N"));
                    System.out.println("2) Get the bipolar matrix for input \n" + bipolarMatrix);
                    System.out.println("3) dot product bipolar matrix & each of the columns in weight matrix");
            }

            // Updating nodes.
            for (int i = 0; i < inputDigitPattern.length; i++) {
                    try {
                            // Create a column matrix.
                            Matrix columnMatrix = weightMatrix.getColumnMatrix(i);
                            double dotProductResult = bipolarMatrix.dotProduct(columnMatrix);

                            // Show mathematical process for dot product.
                            if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                                    System.out.print("[3."+ String.format("%02d", i) +
                                    "] (bipolar matrix) . (Weight matrix column "+ String.format("%02d", i)+") = ");
                            }

                            // Update weights.
                            if (dotProductResult > 0) {
                                    outputDigitPattern[i] = 1.00;
                                    if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                                            System.out.println(" "+ dotProductResult + "  > 0  ==>  1");
                                    }
                            } else {
                                    outputDigitPattern[i] = 0;
```

```
                                        if (DigitRecognizer.mode == Mode.SHOW_MATH) {
                                                System.out.println(dotProductResult + " <= 0  ==>  0");
                                        }
                                }
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
                return outputDigitPattern;
        }

        /**
         * Transform a pattern to a bipolar pattern.
         * @param pattern pattern
         * @return a bipolar pattern
         */
        public static double[] toBipolar(double[] pattern) {
                double[] bipolarPattern = new double[pattern.length];
                for (int i = 0; i < pattern.length; i++) {
                        if (pattern[i] == 0) {
                                bipolarPattern[i] = -1.00;
                        } else {
                                bipolarPattern[i] = 1.00;
                        }
                }
                return bipolarPattern;
        }

        /**
         * Transform a bipolar pattern to a pattern.
         * @param bipolarPattern bipolar pattern
         * @return pattern
         */
        public static double[] fromBipolar(double[] bipolarPattern) {
                double[] pattern = new double[bipolarPattern.length];
                for (int i = 0; i < bipolarPattern.length; i++) {
                        if (bipolarPattern[i] == 1.00) {
                                pattern[i] = 1.00;
                        } else {
                                pattern[i] = 0.00;
                        }
                }
                return pattern;
        }
}
```

8.3 Matrix.java

```
package amath383;

/**
 * <b>Matrix</b> is an class for immutable matrix objects
 * and implements the same behaviors of mathematical matrices,
 * including addition, subtraction, dot product, cross product.
 */
public class Matrix {
        private double data[][];
        enum ScalarOperation { ADD, SUBTRACT, MULTIPLY, DIVIDE };

        /**
         * create an empty matrix of rows * columns
         * where every position contains 0
```

```
 * @precon rows > 0, columns > 0
 * @param rows - the count of rows
 * @param columns - the count of columns
 */
public Matrix(int rows, int columns) {
        data = new double[rows][columns];
}

/**
 * create an matrix from data stored in 2d array
 * @param data - the 2d array storage of matrix
 */
public Matrix(double data[][]) {
        this.data = new double[data.length][data[0].length];
        for (int i = 0; i < this.data.length; i++) {
                for (int j = 0; j < this.data[0].length; j++) {
                        this.data[i][j] = data[i][j];
                }
        }
}

/**
 * Matrix addition
 * @param matrix - the other matrix to be added to this
 * @return a new matrix of the result
 * @throws Exception - if the other matrix has different same size than this
 */
public Matrix add(Matrix matrix) throws Exception {
        if ((data.length != matrix.data.length) ||
                (data[0].length != matrix.data[0].length))
                throw new Exception("matrices must have matching size");
        double returnData[][] = new double[data.length][data[0].length];
        for (int i = 0; i < data.length; i++) {
                for (int j = 0; j < data[0].length; j++) {
                        returnData[i][j] = data[i][j] + matrix.data[i][j];
                }
        }
        return new Matrix(returnData);
}

/**
 * Matrix subtraction
 * @param matrix - the other matrix to be subtracted from this
 * @return a new matrix of the result
 * @throws Exception - if the other matrix has different same size than this
 */
public Matrix subtract(Matrix matrix) throws Exception {
        return add(matrix.scalarOperation(-1, ScalarOperation.MULTIPLY));
}

/**
 * Matrix cross product
 * @param matrix - the other matrix to multiply with this
 * @return a new matrix of the result
 * @throws Exception - if the row counts of the other matrix
 *                                          != column counts of this matrix,
 *                                          meaning they cannot multiply each other
 */
public Matrix multiply(Matrix matrix) throws Exception {
        if (data[0].length != matrix.data.length)
            throw new Exception("matrices must have matching inner dimension");
        double returnData[][] = new double[data.length][matrix.data[0].length];
        for (int i = 0; i < data.length; i++) {
```

```
                for (int j = 0; j <  matrix.data[0].length; j++) {
                        double result = 0;
                        for (int k = 0; k < data[0].length; k++) {
                                result += data[i][k] * matrix.data[k][j];
                        }
                        returnData[i][j] = result;
                }
        }
        return new Matrix(returnData);
}

/**
 *   Helper function - a shorthand to apply scalar operation
 *   to individual elements in this matrix
 * @param x - offset used in operation
 * @param scalarOperation - macro indicates which operation to use,
 *                          options stored in {@code ScalarOperation}
 * @return a new matrix of result
 */
public Matrix scalarOperation(double x, ScalarOperation scalarOperation) {
        double returnData[][] = new double[data.length][data[0].length];
        for (int i = 0; i < data.length; i++) {
                for (int j = 0; j < data[0].length; j++) {
                        switch (scalarOperation) {
                                case ADD:
                                        returnData[i][j] = data[i][j] + x;
                                        break;
                                case SUBTRACT:
                                        returnData[i][j] = data[i][j] - x;
                                        break;
                                case MULTIPLY:
                                        returnData[i][j] = data[i][j] * x;
                                        break;
                                case DIVIDE:
                                        returnData[i][j] = data[i][j] / x;
                                        break;
                        }
                }
        }
        return new Matrix(returnData);
}

/**
 * helper function - IDENTITY
 * @param size - the row/column counts of identity matrix
 * @return an identity matrix
 */
public static Matrix identity(int size) {
        Matrix matrix = new Matrix(size, size);
        for (int i = 0; i < size; i++) {
                matrix.data[i][i] = 1;
        }
        return matrix;
}

/**
 * transpose this matrix
 * @return return a matrix of result
 */
public Matrix transpose() {
        double[][] returnData = new double[data[0].length][data.length];
        for (int i = 0; i < data.length; i++) {
                for (int j = 0; j < data[0].length; j++) {
```

```
                        returnData[j][i] = data[i][j];
                }
        }
        return new Matrix(returnData);
}

/**
 * matrix dot product
 * @param matrix - the other matrix to multiply with this
 * @return a new matrix of result
 * @throws Exception - if the other matrix doesn't have same size as this
 *                                              or either is not vector
 */
public double dotProduct(Matrix matrix) throws Exception {
        if (!this.isVector() || !matrix.isVector()) {
                throw new Exception("can only dot product 2 vectors");
        } else if ((this.flatten().length != matrix.flatten().length)) {
                throw new Exception("both vectors must have same size");
        }
        double returnValue = 0;
        for (int i = 0; i < this.flatten().length; i++)
            returnValue += this.flatten()[i] * matrix.flatten()[i];
        return returnValue;
}

/**
 * set individual elements to 0
 * @return a reference to this
 */
public Matrix clear() {
        for (int i = 0; i < data.length; i++) {
                for (int j = 0; j < data[0].length; j++) {
                        data[i][j] = 0;
                }
        }
        return this;
}

/**
 * create a row matrix
 * @param array - store elements to put in this array
 * @return return a matrix of 1 * array.length
 */
public static Matrix toRowMatrix(double[] array) {
        double[][] data = new double[1][array.length];
        System.arraycopy(array, 0, data[0], 0, array.length);
        return new Matrix(data);
}

/**
 * create a snapshot of a column in this matrix
 * @param column - the column number
 * @return return a new matrix copy of that column
 */
public Matrix getColumnMatrix(int column) {
        double[][] data = new double[this.data.length][1];
        for (int i = 0; i < this.data.length; i++) {
                data[i][0] = this.data[i][column];
        }
        return new Matrix(data);
}

/**
```

```
 * check if this is a vector
 */
public boolean isVector() {
        boolean flag = false;
        if (this.data.length == 1) flag = true;
        else if( this.data[0].length == 1) flag = true;
        return flag;
}

/**
 * flatten this matrix
 * @return an array of individual elements in this matrix
 */
public double[] flatten() {
        double returnValue[] = new double[data.length * data[0].length];
        int i = 0;
        for (int row = 0; row < data.length; row++) {
                for (int column = 0; column < data[0].length; column++) {
                        returnValue[i++] = data[row][column];
                }
        }
        return returnValue;
}

/**
 * @return the 2d array storing this matrix
 */
public double[][] getData() {
        return data;
}

/**
 * retrieve a matrix from a flattened one, represented by
 * an array
 * @param data - the array storing the individual elements
 * @param numbOfRows - row counts of resulting matrix
 * @return a new matrix from data
 * @throws Exception - if size of data is not dividable by numbOfRows
 */
public static Matrix getMatrix(double data[], int numbOfRows) throws Exception {
        if (data.length % numbOfRows != 0)
            throw new Exception("size of data not divisible by number of rows");
        Matrix drawingMatrix = new Matrix(numbOfRows, data.length / numbOfRows);
        int i = 0;
        for (int row = 0; row < drawingMatrix.data.length; row++) {
                for (int column = 0; column < drawingMatrix.data[0].length; column++) {
                        drawingMatrix.data[row][column] = data[i++];
                }
        }
        return drawingMatrix;
}

/**
 * compressed string representation of this matrix
 * @return
 */
public String toPackedString() {
        StringBuffer bodySB = new StringBuffer();
        for (int i = 0; i < data.length; i++) {
                for (int j = 0; j < data[0].length; j++) {
                        bodySB.append((int)data[i][j]);
                }
                bodySB.append("\n");
```

```
            }
            return bodySB.toString();
        }

        /**
         * a formatted string representation of this matrix
         * @param columnLabel
         * @param rowLabel
         * @return
         */
        public String toString(String columnLabel, String rowLabel) {
            StringBuffer headingSB = new StringBuffer();
            headingSB.append("     | ");
            for (int i = 0; i < data[0].length; i++) {
                headingSB.append(" " + columnLabel + String.format("%02d", i) +"");
            }
            headingSB.append("\n");
            StringBuffer bodySB = new StringBuffer();
            for (int i = 0; i < headingSB.length(); i++) {
                bodySB.append("-");
            }
            bodySB.append("\n");
            for (int i = 0; i < data.length; i++) {
                bodySB.append(rowLabel + String.format("%02d", i) +" |");
                for (int j = 0; j < data[0].length; j++) {
                    if (data[i][j] >= 0) {
                        bodySB.append("   " + (int) data[i][j]);
                    } else {
                        bodySB.append("  "+  (int)data[i][j]);
                    }
                }
                bodySB.append("\n");
            }
            return headingSB.toString() + bodySB.toString();
        }

        @Override
        public String toString() {
            return toString("C", "R");
        }
}
```