Оглавление

Глава 1. Построение карты Кохонена

Карта Кохонена

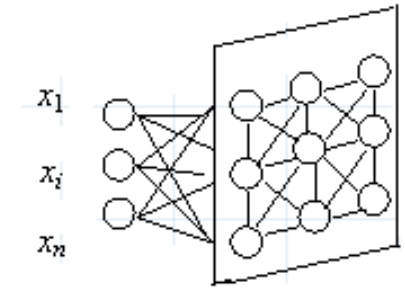
Самоорганизующаяся карта — нейронная сеть с обучением без учителя, выполняющая задачу визуализации и кластеризации.

Самоорганизующиеся карты могут использоваться для решения таких задач, как моделирование, прогнозирование, поиск закономерностей в больших массивах данных, выявление наборов независимых признаков и сжатие информации.

Принцип работы

П вариант карты строится по заданным данным, размерность которых заранее известно. В процессе обучения векторы веса узлов приближаются к входным данным. Для каждого наблюдения выбирается наиболее похожий по вектору веса узел (BMU), значение его вектора веса приближается к наблюдению. Также к наблюдению приближаются векторы веса всех соседей, т.е. узлов, расположенных рядом. Таким образом если в множестве входных данных два наблюдения были схожи, на карте им будут соответствовать близкие узлы.

Циклический процесс обучения, перебирающий входные данные, заканчивается по достижении картой заданной погрешности, или по совершении заданного количества итераций. Таким образом, в результате обучения карта Кохонена классифицирует входные данные на кластеры и визуально отображает многомерные входные данные в двумерной плоскости.



входные нейроны

выходные нейроны

Глава 2. Практическая реализация Карты Кохонена

Алгоритмы/структуры, использованные при построении карты Кохонена

1. Нахождение для него лучшей единицы соответствия (Best Matching Unit, BMU) — узла на карте, вектор веса которого меньше всего отличается от наблюдения.

$$d\left(\mathbf{x},\mathbf{w}_j\right)=\min_{1\le i\le n} d\left(\mathbf{x},\mathbf{w}_i\right)$$

2. В качестве расстояния используется Евклидова мера

$$d\left(\mathbf{x},\mathbf{w}_i\right)=\left\|\mathbf{x}-\mathbf{w}_i\right\|=\sqrt{\sum_{j=1}^{n}\left(x_j-w_{ij}\right)^2}$$

3. Определение радиуса обучения
Вокруг BMU образует окружение – радиус обучения. Он определяет какие нейроны подвергаются обучению на текущей итерации. Уменьшается с ростом количества итераций

4. Веса нейронов, лежащих в радиусе обновляются по правилу Кохонена:

$$w_{ij}(t+1)=w_{ij}(t)+\alpha_i(t)[x(t)-w_{ij}(t)],\quad \text{or}$$
$$w_{ij}(t+1)=w_{ij}(t)+\alpha_i(t)\beta_{ij}(t)[x(t)-w_{ij}(t)]$$

$$\sigma(t)=\sigma_0\cdot\exp(\frac{-t}{\lambda}),\quad \text{where } t=1,2,3\ldots n$$
$$\alpha(t)=\alpha_0\cdot\exp(\frac{-t}{\lambda}),\quad \text{where } t=1,2,3\ldots n$$

Таким образом, вектор весов перемещается ближе к «входному вектору»

5. Применение SOM алгоритма
Выбираем случайный вектор.

Пускаем цикл до числа итераций :

Выбираем вектор отличный от старого.

Вычисляем BMU.

Обновляем соседей.

Реализация карты Кохонена на Java на примере задачи о кластеризации цветов

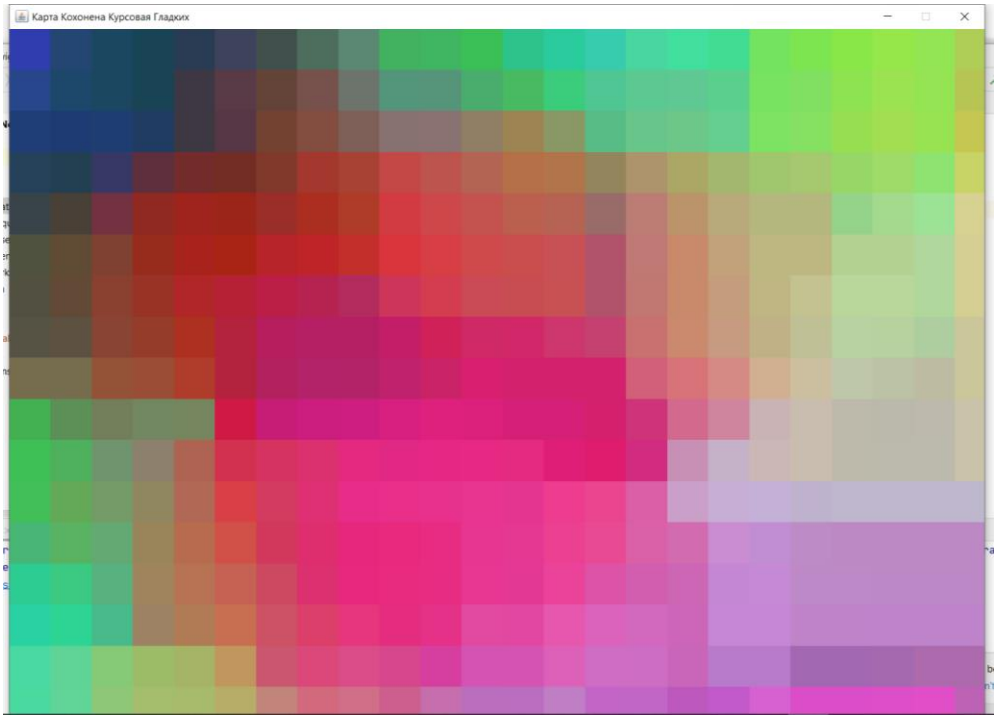
Для написания курсовой работы был использован язык Java и библиотека Swing для визуализации данных.

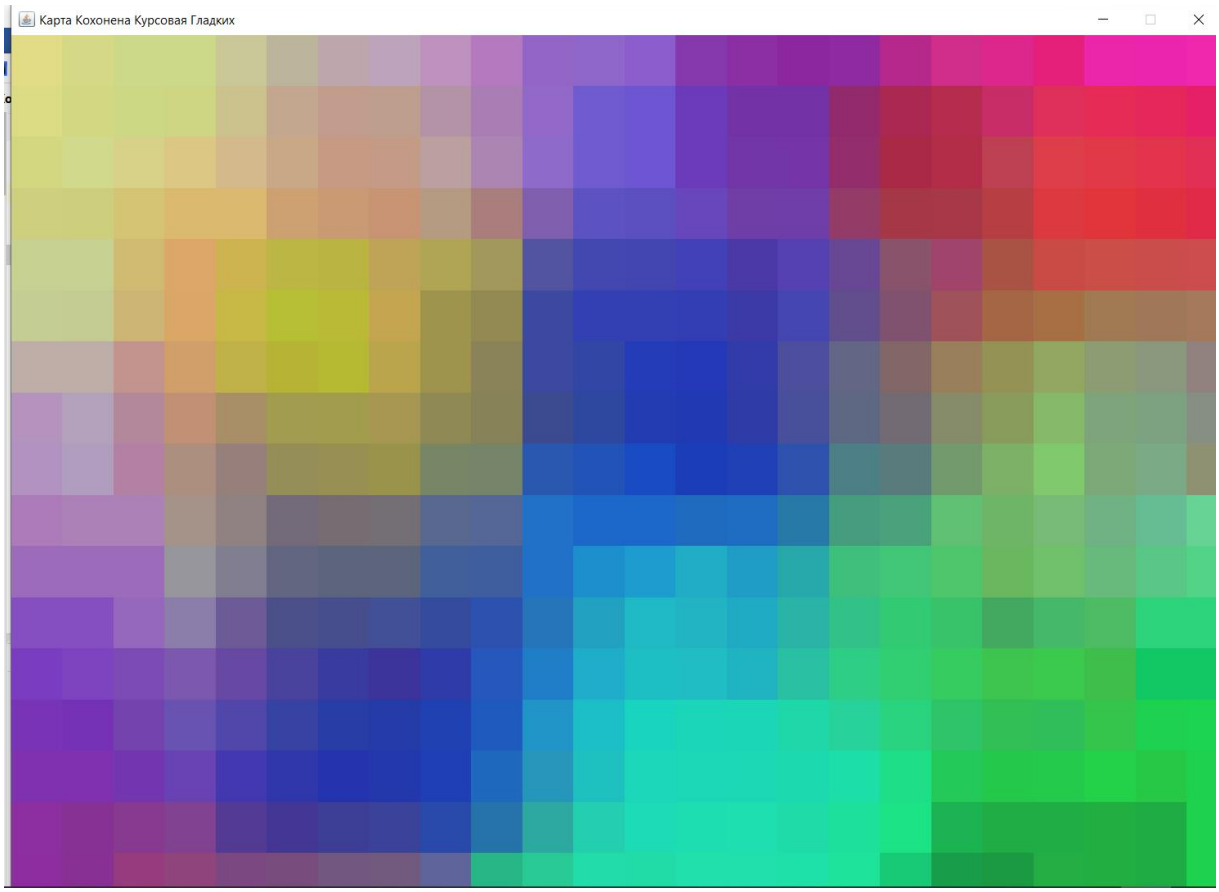
Класс *Neuron* содержит список весов, их координаты, может возвращать и устанавливать их.
Класс *Network* содержит двумерный слой нейронов. Содержит функции для вычисления Евклидова расстояния между двумя векторами, а также функции для нахождения BMU и обновления соседей.
Класс *Kohonen* содержит в себе обучающую сеть и входную сеть, к которым применяет функцию *somAlgorythm*, которая реализует в себе основной алгоритм Кохонена.
Для наглядности с использованием библиотеки Swing данные визуализируются. За визуализацию отвечает класс *ColorSquareRenderScreen*, в нем же происходит начальная инициализация цветов. Класс *FileParser* помогает парсить текстовые файлы.
Точкой запуска, где инициализируется окно и запускается алгоритм Кохонена, служит класс *Application*. Для создания эффекта анимации, на каждой итерации рабочий поток засыпает на 20 миллисекунд.

Результаты работы программы

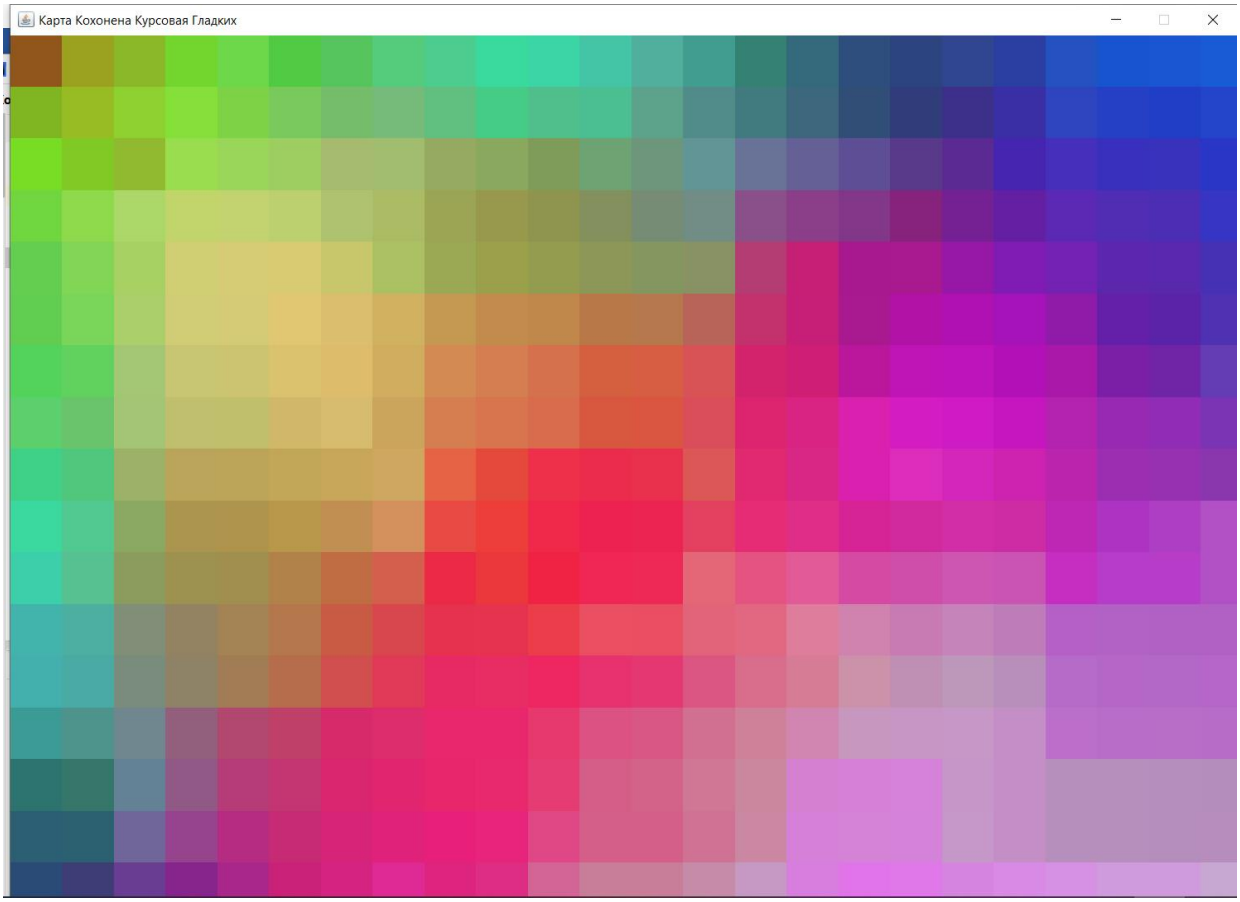
Пример входных данных:

Примеры выходных данных (2000 итераций):





Пример выходных данных (3000 итераций):



Заключение

Сеть Кохонена хорошо подходит для распознания кластеров в данных, а также в установлении близости классов. Таким образом, можно улучшить свое понимание структуры данных, чтобы затем правильно выбрать нейросетевую модель. Если в данных распознаны классы, то их можно обозначить, после чего сеть сможет решать задачи классификации.

Сети Кохонена можно использовать и в тех задачах классификации, где классы уже заданы - тогда преимущество будет в том, что сеть сможет выявить сходство между различными классами.

Место реализованной структуры данных / алгоритма среди других алгоритмов.

Кроме того, сеть Кохонена достаточно проста т.к. имеет всего два слоя – входной и выходной.

Список литературы

1. *Кохонен, Т.* Самоорганизующиеся карты / Т. Кохонен – М.: БИНОМ, Лаборатория знаний, 2008
2. *Осовский, С*. Нейронные сети для обработки информации / С. Осовский – М.: Финансы и статистика, 2002
3. *Хайкин, С*. Нейронные сети: полный курс / С. Хайкин. - М. Вильямс, 2006

Приложения

Класс *Neuron*:

```
//Список весов
private final ArrayList<Double> weights;

private final int x;
private final int y;
```

Листинг 1. Список весов *weights*. Координаты расположения нейрона *x и y*.

```
// Конструктор, принимающий список весов + расположение
public Neuron(ArrayList<Double> w, int x, int y) {
    this.weights = w;
    this.x = x;
    this.y = y;
}

public ArrayList<Double> getWeights() {
    return weights;
}

//Получить вес i позиции из списка весов
public Double getWeight(int i) {
    return this.weights.get(i);
}

//Получить позицию нейрона по абсциссам
public int getX() {
    return x;
}

// Позиция нейрона по ординатам
public int getY() {
    return y;
}

// Установить вес по i позиции
public void setWeight(int i, double value) {
    this.weights.set(i, value);
}

// Получить количество весов
public int getSizeWeights() {
    return this.weights.size();
}
```

Листинг 2. Конструктор, геттеры и сеттеры

Класс *Network*:

```
 //Двумерный слой нейронов
private final Neuron[][] network;
```

Листинг 3. Двумерный слой нейронов *network*

```
 // Размер сети
private final int width;
private final int height;
```

Листинг 4. Размер сети. Ширина *width*, высота *height*

```
//  Вычислить Евклидово расстояние между двумя векторами
public double distance(ArrayList<Double> v1, ArrayList<Double> v2) {
    double distance = 0.0;
    for (int i = 0; i < v1.size(); i++)
        distance += Math.pow(v1.get(i) - v2.get(i), 2);

    return Math.sqrt(distance);
}
```

Листинг 5. Функция *distance* для вычисления Евклидового расстояния

```
/**
 * Получить Best Matching Unit сети из входного вектора
 * BMU - это нейрон, у которого наименьшее расстояние от входного вектора
 */
public Neuron getBMU(ArrayList<Double> input) {
    Neuron BMU = null;
    double distanceMin = Double.MAX_VALUE;

    for (int i = 0; i < this.height; i++) {
        for (int j = 0; j < this.width; j++) {

            double distance = distance(input, getNeuron(i, j).getWeights());

            if (distance < distanceMin) {
                distanceMin = distance;
                BMU = getNeuron(i, j);
            }
        }
    }
    return BMU;
}
}
```

Листинг 6. Функция *getBMU* для получения Best Matching Unit

```
/**
 * Обновить значения соседей BMU
 *
 * 1. Вычислить радиус обучения
 * 2. В этом радиусе будут обновляться нейроны
 */
public void updateNeighbors(Neuron BMU, ArrayList<Double> input, int
currentIteration, int steps) {
    double radiusMap = width / 2;
    double lambda = steps / radiusMap;

    // Вычисляем радиус обучения
    double radius = radiusMap * Math.exp(-currentIteration / lambda);

    int firstMin = (BMU.getY() - radius) < 0 ? 0 : (int) (BMU.getY() -
radius);
    int firstMax = (BMU.getY() + radius) > width ? width : (int) (BMU.getY()
+ radius);
    int secondMin = (BMU.getX() - radius) < 0 ? 0 : (int) (BMU.getX() -
radius);
    int secondMax = (BMU.getX() + radius) > width ? width : (int) (BMU.getX()
+ radius);

    //Установите новые веса нейронов в непосредственной близости
    for (int i = firstMin; i < firstMax; i++) {
        for (int j = secondMin; j < secondMax; j++) {
            // Обход весов
            Neuron neuron = getNeuron(i, j);

            // Learning rate
            double lr = 1;

            double sigm = lr * Math.exp(-currentIteration / lambda);

            double sumDistance = distance(BMU.getWeights(),
neuron.getWeights());

            double theta = Math.exp(-1 * (sumDistance / (2 * Math.pow(radius,
2))));

            for (int k = 0; k < BMU.getSizeWeights(); k++) {

                double value = neuron.getWeight(k) + theta * sigm *
(input.get(k) - neuron.getWeight(k));
                neuron.setWeight(k, value);
            }
        }
    }
}
```

Листинг 7. Функция *updateNeighbours* для обновления весов соседей

Класс *Kohonen*:

```
// количество итераций
private final int steps;
```

Листинг 8. Количество итераций

```
//Сеть
public Network network;

// Входная сеть
ArrayList<ArrayList<Double>> input;
```

Листинг 9. Сеть и входная сеть

```
//Сеть
public Network network;

// Входная сеть
ArrayList<ArrayList<Double>> input;
```

Листинг 10. Сеть и входная сеть

```
JPanel screen;
```

Листинг 11. JPanel для визуализации

```
// Конктруктор. Принимает файл параметров, ширину и длину экрана
public Kohonen(String filename, int width, int height, JPanel screen) {
    // Размер сети
    this.network = new Network(width, height);
    this.input = FileParser.getListFromFile(filename);
    this.steps = 2000;
    this.screen = screen;
}
```

Листинг 10. Конструктор класса

```
// Добавляет нейрон в сеть по позиции
public void addNeuron(ArrayList<Double> weights, int i, int j) {
    network.setNeuron(i, j, new Neuron(weights, i, j));
}
```

Листинг 11. Добавление нейрона в сеть

```
/**
 * SOM алгоритм
 * Пока < iterations :
 * - Выбираем рандомный вектор.
```

```
 * - Находим BMU ввода
 * - Обновляем соседей BMU
 */
public void somAlgorythm() {
    // Выбираем рандомный вектор
    ArrayList<Double> oldVector = new ArrayList<>();
    ArrayList<Double> vector;

    for (int t = 0; t < steps; t++) {

        // Выбрать вектор, отличный от предыдущего
        do {
            int randomInput = (int) (Math.random() * input.size());
            vector = input.get(randomInput);
        } while (oldVector.equals(vector));

        oldVector = new ArrayList<>(vector);

        // Вычисляем bmu (Best Matching Unit).
        Neuron bmu = network.getBMU(vector);

        // Обновляем соседей
        if (steps >= bmu.getSizeWeights()) network.updateNeighbors(bmu,
vector, t, steps);

        // Обновляем экран
        this.screen.repaint();
        // Sleep для анимации
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            System.out.println("Unexpected exception" + e);
        }
    }
}
```

Листинг 11. Функция somAlgorythms для применения SOM алгоритма

```
//Парсер текстовый файлов. Для упрощения
public class FileParser {

    public static String read(String filename) {
        StringBuilder text = new StringBuilder();
        try {
            BufferedReader reader = new BufferedReader(new
FileReader(filename));
            String line = reader.readLine();

            while (line != null) {
                text.append(line);
                text.append("\n");
                line = reader.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return text.toString();
```

```
    }

    //вектор + значения
    public static ArrayList<ArrayList<Double>> getListFromFile(String
filename) {
        String inputStr = read(filename);
        ArrayList<ArrayList<Double>> input = new ArrayList<>();

        String[] inputSplit = inputStr.split("\n");
        String[] vectors;
        String line;
        int index = 0;

        for (String str : inputSplit) {
            line = str;

            input.add(new ArrayList<>());
            vectors = line.split("\t");

            for (int i = 1; i < vectors.length; i++) {
                input.get(index).add(Double.parseDouble(vectors[i]));
            }
            index++;
        }

        return input;
    }
}
```

Листинг 12. Парсер данных из текстового файла

Класс *ColorSquareRenderScreen*:

```
private final int width = 1200;
private final int height = 1200;
private final int squareSize = 50;
private final int measuredWidth = width / squareSize;
private final int measuredHeight = height / squareSize;
private final Kohonen kohonen;
```

Листинг 13. *width/height* – ширина и высота экрана, *squareSize* – размер квадратов, отображаемых в окне, *measuredWidth/measuredHeight* – количество квадратов по вертикали/горизонтали, *kohonen* – карта Кохонена, которую будем визуализировать

```
// Инициализировать нейроны рандомными значениями
public void init() {
    for (int i = 0; i < measuredHeight; i++) {
        for (int j = 0; j < measuredWidth; j++) {

            // Установить рандомные цвета
            double red = Math.random();
            double green = Math.random();
            double blue = Math.random();
```

```
            // Создать нейрон и добавить в сеть
            kohonen.addNeuron(new ArrayList<>(Arrays.asList(red, green,
blue)), j, i);
        }
    }
}
```

Листинг 14. Функция *init* для задания рандомных цветов карте Кохонена

```
// Рисуем цветные квадраты
public void paintComponent(Graphics g) {
    super.paintComponent(g);

    for (int i = 0; i < measuredHeight; i++) {
        for (int j = 0; j < measuredWidth; j++) {
            Neuron neuron = kohonen.network.getNeuron(i, j);
            ArrayList<Double> colors = neuron.getWeights();

            g.setColor(new Color((int) (colors.get(0) * 255), (int)
(colors.get(1) * 255), (int) (colors.get(2) * 255)));
            g.fillRect(j * squareSize, i * squareSize, squareSize,
squareSize);
        }
    }
}
```

Листинг 15. Функция *paintComponent* для отрисовки квадратов на экране

Класс *Application*:

```
public class Application {

    public static void main(String[] args) {
        ColorSquareRenderScreen screen = new ColorSquareRenderScreen();
        JFrame frame = new JFrame();
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(screen, "Center");
        frame.setTitle("Карта Кохонена Курсовая Гладких");
        frame.setSize(1200, 1200);
        frame.setVisible(true);

        screen.getKohonen().somAlgorythm();
    }
}
```

Листинг 16. Класс *Applcation*, который служит точкой запуска программы