**Министерства науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования**

**«Заполярный государственный университет»**

**Кафедра информационных систем и технологий**

**Отчет**

**По дисциплине: «Нейронные сети»**

**«Лабораторная работа № 4»**

**Выполнил:**

студент группы ИС-19

Исрафилов Сабухи Мадад оглы

**Проверил преподаватель:**

Дыптан Елена Арнольдовна

Норильск 2022

## Цель работы

Реализовать в программе нейронную сеть Хопфилда для решения задачи восстановления искаженных изображений, предварительно обученной на эталонных изображениях.

## Описание архитектуры сети

**Сеть Хопфилда** — полносвязная нейронная сеть с симметричной матрицей связей. Такая сеть может быть использована для организации ассоциативной памяти, как фильтр, а также для решения некоторых задач оптимизации. Сеть Хопфилда является абсолютно однородной структурой без какой-либо внутренней специализации ее нейронов. Её классический вариант состоит из единственного слоя нейронов, число которых является одновременно числом входов и выходов сети. Каждый нейрон сети связан со всеми остальными нейронами, а также имеет один вход, через который осуществляется ввод сигнала.

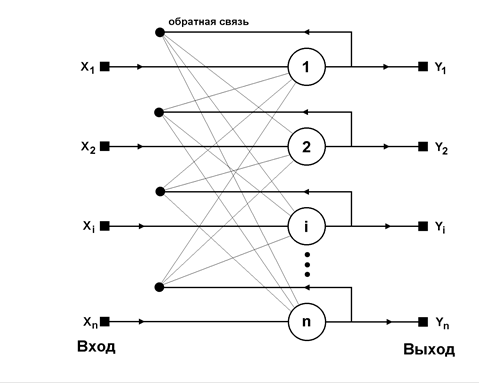
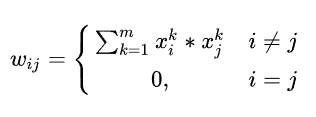


Рисунок 1 – структурная схема нейронной сети Хопфилда

## Метод обучения

Алгоритм:

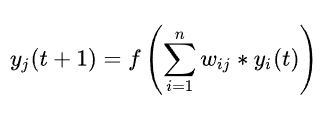
1. Инициализация. Веса нейронов устанавливаются по следующей формуле:



m – количество образов

xi и xj – i-ый и j-ый элементы вектора k-ого образа

1. На входы сети подается неизвестный сигнал
2. Рассчитывается выход сети:



f – пороговая активационная функция с областью значений [-1;1]

t – номер итерации

j = 1…n;n – количество входов и нейронов

1. Проверка изменения выходных значений за последнюю итерацию. Если выходы изменились — переход к пункту 3, иначе, если выходы стабилизировались, завершение функционирования. При этом выходной вектор представляет собой образец, наилучшим образом сочетающийся с входными данными.
2. Если сеть сошлась к образу Z, то веса пересчитываем, т.е. ухудшаем память:

*(0 < < 0.1)*

После нескольких разобучений свойства сети улучшаются

## Практическая часть

**Задача:** реализовать нейронную сеть Хопфилда для восстановления искаженных изображений, предварительно обученной на эталонных изображениях. Пользователь рисует эталонные изображения, после чего рисует зашумленное, которое нейронная сеть должна корректно распознать.

В зависимости от изменения параметров обучения нейронная сеть реагирует по-разному, данные экспериментов представлены в таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество образов** | **Эпсилон** | **Итераций разобучений** | **Ошибка** |
| 2 | 0.05 | 2 | 0 |
| 2 | 0.01 | 1 | 0 |
| 3 | 0.05 | 5 | 0.17 |
| 3 | 0.01 | 11 | 0.3 |
| 4 | 0.05 | 10 | 0.35 |
| 4 | 0.01 | 21 | 0.38 |

## Листинг

let gridSize = 10; // Размер сетки

const squareSize = 45; // клетка

let inputVector = gridSize \* gridSize;

let isDrawing = false;

const weights = [];

let userImageState = [];

let all\_Images = [];

let result = [];

const E = 0.05; // Коэф для переобучения;

for (let i = 0; i < inputVector; i++) {

weights[i] = new Array(inputVector).fill(0);

userImageState[i] = -1;

}

const userCanvas = document.getElementById('userCanvas');

const userContext = userCanvas.getContext('2d');

const netCanvas = document.getElementById('netCanvas');

const netContext = netCanvas.getContext('2d');

const calcIndex = (x, y, size) => x + y \* size;

const isValidIndex = (index, len) => index < len && index >= 0;

const getNewSquareCoords = (canvas, clientX, clientY, size) => {

const rect = canvas.getBoundingClientRect();

const x = Math.ceil((clientX - rect.left) / size) - 1;

const y = Math.ceil((clientY - rect.top) / size) - 1;

return { x, y };

};

// Сетка

const drawGrid = (ctx) => {

ctx.beginPath();

ctx.fillStyle = 'white';

ctx.lineWidth = 2;

ctx.strokeStyle = 'black';

for (let row = 0; row < gridSize; row++) {

for (let column = 0; column < gridSize; column++) {

const x = column \* squareSize;

const y = row \* squareSize;

ctx.rect(x, y, squareSize, squareSize);

ctx.fill();

ctx.stroke();

}

}

ctx.closePath();

};

// Рисуем образ сети

const drawImageFromArray = (data, ctx) => {

const twoDimData = [];

while (data.length) twoDimData.push(data.splice(0, gridSize));

drawGrid(ctx);

for (let i = 0; i < gridSize; i++) {

for (let j = 0; j < gridSize; j++) {

if (twoDimData[i][j] === 1) {

ctx.fillStyle = 'black';

ctx.fillRect(j \* squareSize, i \* squareSize, squareSize, squareSize);

}

}

}

**// Просчёт ошибки**

let all\_errors = [];

let flatRecog = twoDimData.flat();

for (let index = 0; index < all\_Images.length; index++) {

let currImage = all\_Images[index];

let error = 0;

for (let i = 0; i < flatRecog.length; i++) {

if (flatRecog[i] !== currImage[i]) error++;

}

error = error / flatRecog.length;

all\_errors.push(error);

}

let error\_index = all\_errors.indexOf(Math.min(...all\_errors));

console.log(

`На ${(1 - all\_errors[error\_index]) \* 100}% это образ ${error\_index + 1}`

);

console.log(all\_errors);

};

// Ивенты мыши

const handleMouseDown = (e) => {

userContext.fillStyle = 'black';

userContext.fillRect(

Math.floor(e.offsetX / squareSize) \* squareSize,

Math.floor(e.offsetY / squareSize) \* squareSize,

squareSize,

squareSize

);

const { clientX, clientY } = e;

const coords = getNewSquareCoords(userCanvas, clientX, clientY, squareSize);

const index = calcIndex(coords.x, coords.y, gridSize);

if (isValidIndex(index, inputVector) && userImageState[index] !== 1) {

userImageState[index] = 1;

}

isDrawing = true;

};

const handleMouseMove = (e) => {

if (!isDrawing) return;

userContext.fillStyle = 'black';

userContext.fillRect(

Math.floor(e.offsetX / squareSize) \* squareSize,

Math.floor(e.offsetY / squareSize) \* squareSize,

squareSize,

squareSize

);

const { clientX, clientY } = e;

const coords = getNewSquareCoords(userCanvas, clientX, clientY, squareSize);

const index = calcIndex(coords.x, coords.y, gridSize);

if (isValidIndex(index, inputVector) && userImageState[index] !== 1) {

userImageState[index] = 1;

}

};

const stopDrawing = () => {

isDrawing = false;

};

// Сброс картинки

const clearCurrentImage = () => {

drawGrid(userContext);

drawGrid(netContext);

userImageState = new Array(gridSize \* gridSize).fill(-1);

};

**// Запомнить образ**

const memorizeImage = () => {

for (let i = 0; i < inputVector; i += 1) {

for (let j = 0; j < inputVector; j += 1) {

if (i === j) weights[i][j] = 0;

else {

weights[i][j] += userImageState[i] \* userImageState[j];

}

}

}

all\_Images.push(userImageState);

};

**// Переобучить**

const reTrain = () => {

for (let i = 0; i < inputVector; i++) {

for (let j = 0; j < inputVector; j++) {

if (i === j) weights[i][j] = 0;

else {

weights[i][j] -= E \* (result[i] \* result[j]);

}

}

}

};

**// Распознать**

const recognizeImage = () => {

let prevNetState;

const currNetState = [...userImageState];

let counter = 0;

do {

counter += 1;

prevNetState = [...currNetState];

for (let i = 0; i < inputVector; i++) {

let sum = 0;

for (let j = 0; j < inputVector; j++) {

sum += weights[i][j] \* prevNetState[j];

}

currNetState[i] = sum >= 0 ? 1 : -1;

}

if (counter === 999) {

console.log('Не могу вспомнить');

return;

}

} while (!\_.isEqual(currNetState, prevNetState));

result = currNetState.slice();

drawImageFromArray(currNetState, netContext);

};

## Приложение

