Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №5**

**Дисциплина: Приложение нейросетевых алгоритмов**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. А. Иванов

Направление подготовки: 02.03.03 Математическое обеспечение и администрирование информационных систем

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. И. Шиян

**Цель работы**

Получение знаний и практических навыков построения однослойных нейронных сетей.

**Задание**

Требуется разработать алгоритмы и программы решения задач 1-2 на языке Python без использования специализированных библиотек.

**Описание решения**

*Задача: рекуррентная сеть Хопфилда.*

Для решения данной задачи была использована библиотека numpy. Сам код задачи включает в себя класс Hopfield\_Network, который содержит все функции: \_\_init\_\_ создаёт объект нейронной сети, задаёт количество нейронов и инициализирует пустые веса (заполненные нулями); train() тренирует нейросеть следующим способом: на вход подаётся список «чистых» примеров, где каждый пример представлен матрицей 10х10. Далее, каждый пример прибавляется к общей матрице весов, и после добавления всех элементов диагональ заполняется нулями (это нужно для того, чтобы нейроны не активировали сами себя); recall() восстанавливает «шумное» изображение по матрице весов: задаётся предел итераций (100), на каждой итерации пример умножается на матрицу весов, и результат пропускается через сигмоидальную функцию активации. Если новый полученный пример идентичен прошлому – функция его возвращает, иначе продолжает пытаться его восстановить пока не закончатся итерации.

Алгоритм работы программы следующий:

- на вход сети подаются «чистые» примеры, на которых она обучается;

- далее, сети подаются по одному «шумные» примеры, которые она пытается восстановить;

- после восстановления сеть возвращает новый пример, который сравнивается с изначальным «чистым» для подсчёта точности.

**Реализация программы на языке Python**

import numpy as np

class HopfieldNetwork:

    def \_\_init\_\_(*self*, *num\_neurons*: int) -> None:

        """

        Функция, которая инициализирует нейронную сеть в виде матрицы весов нейронов

        Args: num\_neurons - число нейронов в сети

        Return: отсутствует

        """

*self*.num\_neurons = *num\_neurons*

*self*.weights = np.zeros((*self*.num\_neurons, *self*.num\_neurons))

    def train(*self*, *patterns*: list[list[list], list[list]]) -> None:

        """

        Функция, которая обучает нейроны на базовых "чистых" примерах

        Args: patterns - список матриц

        Return: отсутствует

        """

        for pattern in *patterns*:

            pattern = np.array(pattern)

*self*.weights += pattern

*#устанавливаем 0 по диагонали, чтобы нейроны как-то там сами себя не активировали*

            np.fill\_diagonal(*self*.weights, 0)

    def recall(*self*, *pattern*: list, *max\_iter*=100) -> list:

        """

        Функция, которая пытается восстановить полученный паттерн

        Args: pattern - строка матрицы, max\_iter - максимально допустимое число итераций

        Return: последний полученный паттерн (под таким паттерном имеется в виду либо тот, который совпал с изначальным, либо полученный на 100 итерации)

        """

*pattern* = np.array(*pattern*)

        for \_ in range(*max\_iter*):

            new\_pattern = np.sign(np.dot(*pattern*, *self*.weights))

            if np.array\_equal(new\_pattern, *pattern*):

                return new\_pattern

*pattern* = new\_pattern

        return *pattern*

*#1 пример - собака, 2-5 - одна и та же буква И (нужно для корректной работы алгоритма сравнения, лучше не придумал)*

examples = [[[1, 1, 1, 1, 1, -1, 1, 1, 1, 1],

            [-1, 1, 1, 1, 1, -1, 1, 1, 1, 1],

            [-1, 1, 1, 1, 1, -1, -1, 1, 1, 1,],

            [-1, 1, 1, 1, 1, -1, -1, -1, -1, 1],

            [1, -1, -1, -1, -1, -1, -1, -1, -1, 1],

            [1, -1, -1, -1, -1, -1, -1, -1, -1, 1],

            [1, -1, -1, -1, -1, 1, 1, 1, 1, 1],

            [1, -1, 1, 1, -1, 1, 1, 1, 1, 1],

            [1, -1, 1, 1, -1, 1, 1, 1, 1, 1],

            [1, -1, -1, 1, -1, -1, 1, 1, 1, 1]],

            [[-1, -1, 1, 1, 1, 1, 1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, -1, -1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],

             [-1, -1, -1, 1, 1, 1, 1, 1, -1, -1]],

             [[-1, -1, 1, 1, 1, 1, 1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, -1, -1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],

             [-1, -1, -1, 1, 1, 1, 1, 1, -1, -1]],

             [[-1, -1, 1, 1, 1, 1, 1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, -1, -1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],

             [-1, -1, -1, 1, 1, 1, 1, 1, -1, -1]],

             [[-1, -1, 1, 1, 1, 1, 1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, 1, -1, -1, -1, -1],

             [-1, -1, 1, 1, 1, -1, -1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

             [-1, -1, 1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],

             [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],

             [-1, -1, -1, 1, 1, 1, 1, 1, -1, -1]]]

*#1 пример - "шумная" собака, 2-5 - разные варианты "шумной" буквы И*

examples\_with\_noise = [[[1, 1, 1, 1, 1, -1, 1, -1, 1, 1],

                        [-1, 1, 1, 1, 1, -1, 1, 1, 1, 1],

                        [-1, 1, 1, 1, 1, -1, -1, 1, 1, 1],

                        [-1, 1, 1, -1, 1, -1, -1, -1, -1, 1],

                        [1, -1, -1, -1, -1, -1, -1, -1, -1, 1],

                        [1, -1, -1, -1, -1, -1, -1, -1, -1, 1],

                        [1, 1, -1, -1, -1, 1, 1, 1, 1, -1],

                        [1, -1, 1, -1, -1, 1, 1, 1, 1, 1],

                        [1, -1, 1, 1, -1, 1, 1, 1, 1, 1],

                        [1, -1, -1, 1, -1, -1, 1, 1, 1, 1]],

                       [[-1, -1, 1, 1, 1, 1, 1, -1, -1, -1],

                        [-1, -1, 1, 1, 1, 1, -1, 1, -1, -1],

                        [-1, 1, 1, -1, 1, 1, -1, -1, -1, 1],

                        [-1, -1, -1, 1, 1, -1, -1, 1, -1, -1],

                        [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

                        [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1],

                        [-1, -1, -1, -1, -1, -1, 1, 1, -1, -1],

                        [-1, -1, -1, -1, -1, 1, 1, 1, 1, -1],

                        [-1, -1, -1, 1, -1, 1, 1, 1, -1, -1],

                        [-1, -1, -1, 1, 1, 1, 1, 1, -1, -1]],

                       [[-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],

                        [1, -1, 1, 1, 1, 1, -1, 1, -1, -1],

                        [-1, 1, -1, 1, 1, -1, -1, -1, -1, -1],

                        [-1, -1, 1, -1, 1, -1, 1, 1, -1, 1],

                        [-1, -1, 1, -1, 1, -1, 1, 1, -1, -1],

                        [1, -1, -1, 1, -1, -1, -1, 1, -1, -1],

                        [-1, -1, 1, -1, -1, 1, 1, 1, -1, -1],

                        [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],

                        [-1, -1, -1, -1, 1, -1, -1, 1, -1, 1],

                        [-1, -1, -1, 1, -1, 1, 1, 1, 1, -1]],

                       [[-1, -1, -1, 1, -1, 1, 1, -1, -1, -1],

                        [-1, -1, 1, -1, 1, 1, -1, -1, -1, -1],

                        [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],

                        [-1, 1, 1, 1, -1, 1, -1, 1, -1, 1],

                        [-1, -1, 1, 1, 1, -1, 1, -1, -1, 1],

                        [1, 1, 1, 1, -1, -1, 1, 1, 1, 1],

                        [1, 1, 1, -1, -1, -1, 1, 1, -1, -1],

                        [-1, 1, 1, -1, 1, 1, -1, -1, -1, -1],

                        [-1, 1, -1, 1, 1, 1, 1, 1, -1, -1],

                        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1]],

                       [[-1, -1, 1, 1, -1, -1, -1, 1, -1, -1],

                        [1, 1, 1, -1, 1, -1, -1, 1, 1, -1],

                        [1, 1, 1, -1, -1, 1, 1, -1, -1, -1],

                        [-1, 1, 1, 1, 1, 1, -1, 1, 1, -1],

                        [-1, -1, 1, 1, -1, -1, 1, 1, 1, -1],

                        [1, -1, 1, 1, -1, -1, -1, -1, -1, 1],

                        [-1, 1, -1, -1, 1, 1, -1, -1, -1, -1],

                        [-1, -1, -1, -1, 1, 1, 1, -1, -1, -1],

                        [1, 1, -1, -1, 1, 1, -1, 1, 1, -1],

                        [1, 1, -1, -1, 1, 1, 1, -1, 1, 1]]]

network = HopfieldNetwork(*num\_neurons*=10)

network.train(examples)

*#проверка работы на "зашумленных" примерах*

for i, example in enumerate(examples\_with\_noise):

    total\_matches = 0

    print("Шумный пример ", i+1)

    for j, noisy\_pattern in enumerate(example):

*#восстановленный паттерн*

        recalled\_pattern = network.recall(noisy\_pattern)

        print("Восстановленный паттерн: ", recalled\_pattern)

        for k in range(len(recalled\_pattern)):

*#проверяем совпадения*

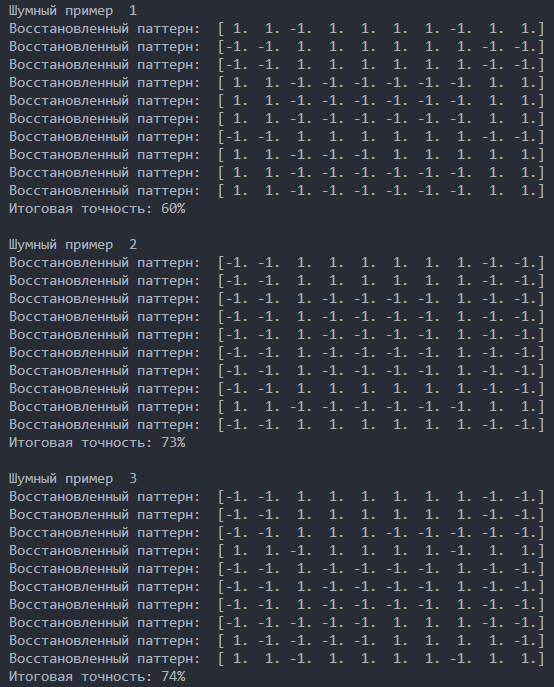
            if recalled\_pattern[k] == examples[i][j][k]:

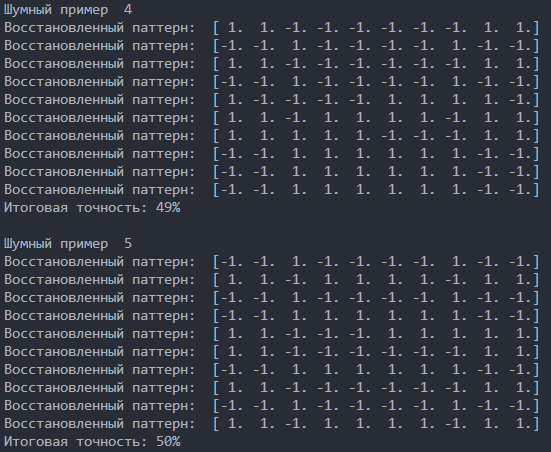
                total\_matches += 1

    print(f"Итоговая точность: {total\_matches}%")

    print()

**Пример запусков и выводы программы**





*Задача: рекуррентная сеть Хэмминга.*

Для решения данной задачи была использована библиотека numpy. Сам код программы состоит из класса Hamming\_Network, который реализует все основные функции, функции euclid\_norm() и общего блока.

Функция euclid\_norm() принимает 2 вектора, находит их норму по отдельности, а затем находит норму между двумя векторами (таким подходом исключается ошибка несовпадения длин векторов). Класс Hamming\_Network включает: \_\_init\_\_() создаёт объект нейронной сети и задаёт пустые слои нейронов; функция init\_basic\_mat() заполняет матрицу нейронов первого слоя через входные примеры, а потом заполняет матрицу нейронов второго (скрытого) слоя через матрицу первого слоя; fun\_activation() реализует линейную пороговую функцию активации; calculate\_outputs\_w1() выполняет подсчёт выходов первого слоя нейронной сети; calculate\_outputs\_w2() выполняет подсчёт выходов второго слоя сети, который и является результатом работы сети.

Алгоритм работы программы следующий:

- создаётся объект нейронной сети;

- выполняется обучение сети на «чистых» примерах;

- выполняется подсчёт выходов первого слоя через «шумные» примеры;

- выполняется восстановление шумных примеров через выходы первого слоя;

- выводится результат работы сети.

**Реализация программы на языке Python**

import numpy as np

def euclid\_norm(*y\_new*: list[float], *y\_previous*: list[float]) -> float:

    """

    Вспомогательная функция которая считает евклидову норму между векторами

    Args: y\_new и y\_previous - 2 вектора

    Return: норма между векторами

    """

    norm\_new = np.linalg.norm(*y\_new*)

    norm\_previous = np.linalg.norm(*y\_previous*)

    return np.linalg.norm([norm\_new], [norm\_previous])

class Hamming\_Network:

    def \_\_init\_\_(*self*) -> None:

        """

        Функция, которая инициализирует слои сети

        Args: отсутствуют

        Return: отсутствует

        """

*self*.w1 = [[0 for \_ in range(49)] for \_ in range(10)]

*self*.w2 = [[0 for \_ in range(10)] for \_ in range(10)]

*self*.w3 = 0

    def init\_basic\_mat(*self*, *examples*: list[list[list[int]]]) -> None:

        """

        Функция, которая заполняет матрицы первого и второго слоя нейронов

        Args: examples - список примеров, каждый их которых - матрица 7х7

        Return: отсутствует

        """

*#заполняем матрицу первого слоя*

        for example in *examples*:

*#переводим пример из 7х7 в 1х49*

            example = np.array(example).reshape(-1,).tolist()

*#в методичке предлагается делать так, а на сайте, с которого была взята информация, второй вариант*

            for i in range(len(*self*.w1)):

*self*.w1[i] = example

*#print(self.w1[i])*

            """ for i in range(len(self.w1)):

                for j in range(len(self.w1[i])):

                    self.w1[i][j] = 1/2 \* example[j]

                #print(self.w1[i]) """

*#заполняем матрицу второго слоя*

        for i in range(len(*self*.w2)):

            for j in range(len(*self*.w2[i])):

                if i == j:

*self*.w2[i][j] = 1

                else:

*#в методичке предлагается делать так, а на сайте, с которого была взята информация, второй вариант (хотя тут второй вариант какой-то непонятный)*

*#создаём случайное малое число от 0 до 1 (по умолчанию создаётся матрица 1х1, так что указываем сам элемент [0][0])*

                    epsilon = np.random.rand(1, 1)[0][0]

*self*.w2[i][j] = -(1 / 9) + epsilon

                    """ self.w2[i][j] = -epsilon """

*#print(self.w2[i])*

    def fun\_activation(*self*, *y*: float) -> float:

        """

        Функция, которая реализует линейную пороговую функцию активации

        Args: y - входное значение

        Return: значение функции активации

        """

*#в методичке предлагается делать так, а на сайте, с которого была взята информация, второй вариант*

        return *y* if *y* >= 0 else 0

        """ t = 49/2

        if y < 0: return 0

        elif 0 < y <= t: return y

        else: return t """

    def calculate\_outputs\_w1(*self*, *examples\_with\_noise*: list[list[list[int]]]) -> list[float]:

        """

        Функция, которая считает состояния и выходы первого слоя нейронов

        Args: examples\_with\_noises - список шумных примеров в виде матриц 7х7

        Return: список выходов нейронов

        """

*#рассчитываем состояния*

        states = []

*#тут чё-то ругалось на for i in range(len(examples\_with\_noise)): example = np.array(examples\_with\_noise[i]).reshape(-1,).tolist()*

        for i, example in enumerate(range(len(*examples\_with\_noise*))):

*#в методичке используется синтаксис sum\_1\_to\_49(x^t\_i \* x\_i), где t (скорее всего) от 1 до p (т.е. 10), но сам пример имеет длину 49, так что костылим*

            example = np.array(example).reshape(-1,).tolist()

            tmp\_avg\_sum\_w = sum(*self*.w2[i]) / len(*self*.w2[i])

            tmp\_sum = 0

            for j in range(len(example)):

                tmp\_sum += example[j] \* tmp\_avg\_sum\_w

            y = 1 - ((1/2 \* (49 - tmp\_sum)) / 49)

            states.append(y)

*#рассчитываем выходы*

        results = []

        for i in range(len(states)):

            results.append(*self*.fun\_activation(states[i]))

*#print(results)*

        return results

    def calculate\_outputs\_w2(*self*, *outputs\_w1*: list[float], *examples\_with\_noise*: list[list[int]], *max\_iter*=100, *e*=0.1) -> list[int]:

        """

        Функция, которая считает выходы второго слоя нейронов

        Args: outputs\_w1 - выходы первого слоя нейронов; examples\_with\_noise - список шумных примеров; max\_iter - s - максимальное кол-во итераций; e - норма разности векторов

        Return: список выходов нейронов

        """

        result = []

        for i, example in enumerate(*examples\_with\_noise*):

*#y\_previous = np.array(example).reshape(-1,).tolist()*

            y\_previous = *outputs\_w1*

            tmp\_avg\_sum\_w = sum(*self*.w2[i]) / len(*self*.w2[i])

            for \_ in range(*max\_iter*):

                tmp\_sum = 0

                for j in range(len([y\_previous])):

*#после первой итерации y\_previous перестаёт быть списком и становится числом*

                    try:

                        tmp\_sum += y\_previous[j] \* tmp\_avg\_sum\_w

                    except:

                        tmp\_sum = y\_previous \* tmp\_avg\_sum\_w

                y\_new = *self*.fun\_activation(tmp\_sum)

*#y\_new оборачивается в массив чтобы не было ошибки object numpy.float64 has no len()*

                if euclid\_norm([y\_new], y\_previous) <= *e*:

                    result.append(y\_new)

                    break

*#конвертируем y\_new в список, чтобы не было проблем с атрибутом len()*

                y\_previous = y\_new.tolist()

*#проверка (если закончились итерации, а в массиве на 1 меньше элементов - добавляем)*

            if (len(result) < i + 1):

                result.append(y\_new)

*#создаём и заполняем списки для каждого примера*

        all\_result = [[0 for \_ in range(10)] for \_ in range(10)]

        for i in range(len(result)):

*#округляем каждый полученный индекс до ближайшего целого*

            tmp\_res = (result[i].round(2) \* 100).round(1)

*#заполняем соответствующий примеру список единицей на указанной позиции*

            all\_result[i][int(tmp\_res) - 1] = 1

*#return result*

        return all\_result

*#тестовые примеры с шумом (максимально странные примеры в выборке, конечно...)*

examples\_with\_noise = [[[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, -1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, -1, 1, 1, 1, -1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1]],

                       [[1, 1, 1, 1, 1, -1, -1],

                        [1, 1, 1, -1, 1, 1, 1],

                        [1, 1, 1, -1, 1, -1, 1],

                        [1, 1, 1, -1, 1, 1, -1],

                        [1, 1, 1, -1, -1, 1, 1, ],

                        [1, 1, 1, -1, 1, 1, 1],

                        [1, 1, 1, 1, 1, 1, 1]],

                       [[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, -1],

                        [1, 1, -1, 1, 1, -1, 1],

                        [-1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1]],

                       [[1, 1, -1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, -1, 1],

                        [1, 1, -1, -1, -1, 1, 1],

                        [1, -1, 1, 1, 1, -1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, -1, 1, 1]],

                       [[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, 1, -1, -1, 1, 1],

                        [1, 1, -1, 1, -1, 1, 1],

                        [1, -1, 1, 1, -1, -1, 1],

                        [1, -1, -1, -1, -1, 1, 1],

                        [1, 1, 1, 1, -1, -1, 1],

                        [1, 1, 1, 1, 1, -1, 1]],

                       [[1, -1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, -1, -1, 1, 1, 1, -1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, -1, -1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1]],

                       [[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, -1],

                        [1, 1, 1, 1, 1, -1, 1],

                        [1, -1, 1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, -1, 1]],

                       [[1, 1, -1, 1, 1, 1, 1, ],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, -1, 1, 1, -1, -1, 1],

                        [1, 1, 1, -1, 1, 1, 1],

                        [1, 1, 1, -1, 1, 1, 1],

                        [1, 1, 1, -1, 1, -1, 1],

                        [1, 1, 1, 1, -1, 1, 1]],

                       [[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [-1, -1, 1, 1, 1, -1, 1],

                        [-1, 1, -1, -1, 1, 1, -1],

                        [1, -1, 1, 1, -1, -1, 1],

                        [1, -1, -1, -1, -1, -1, 1],

                        [1, 1, 1, 1, 1, 1, 1]],

                       [[1, 1, 1, 1, 1, 1, 1],

                        [1, -1, -1, -1, -1, 1, 1],

                        [1, -1, 1, 1, 1, -1, 1],

                        [1, -1, -1, -1, -1, 1, 1],

                        [1, 1, 1, 1, 1, -1, -1],

                        [1, -1, -1, 1, -1, 1, 1],

                        [1, 1, 1, 1, 1, 1, 1]]]

*#"чистые" примеры*

examples = [[[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, 1, -1, -1, -1, 1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, 1, 1, -1, -1, 1, 1],

             [1, 1, -1, 1, -1, 1, 1],

             [1, -1, 1, 1, -1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, -1, 1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, -1, 1],

             [1, 1, 1, 1, -1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, -1, 1, 1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, 1, -1, -1, -1, 1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]],

            [[1, 1, 1, 1, 1, 1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, -1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, -1, 1],

             [1, -1, -1, -1, -1, -1, 1],

             [1, 1, 1, 1, 1, 1, 1]]]

network = Hamming\_Network()

network.init\_basic\_mat(examples)

outputs\_w1 = network.calculate\_outputs\_w1(examples\_with\_noise)

answers = network.calculate\_outputs\_w2(outputs\_w1, examples\_with\_noise)

print(answers)

**Примеры запусков и выводы программы**

