МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

**Лабораторные работы 1 – 5**

ОТЧЁТ

ПО ДИСЦИПЛИНЕ

«НЕЙРОННЫЕ СЕТИ»

студента 5 курса 531 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Сенокосова Владислава Владимировича

|  |  |  |
| --- | --- | --- |
| Преподаватель, доцент | \_\_\_\_\_\_\_\_\_\_ | И.И.Слеповичев |
|  | подпись, дата |  |

Саратов 2024

**Содержание**

[Общие положения выполнения лабораторных работ 3](#_Toc182840902)

[Задание 1: Создание ориентированного графа 4](#_Toc182840903)

[Задание 2: Создание функции по графу 7](#_Toc182840904)

[Задание 3: Вычисление значения функции на графе 9](#_Toc182840905)

[Задание 4: Построение многослойной нейронной сети 12](#_Toc182840906)

[Задание 5: Реализация метода обратного распространения ошибки для многослойной НС 14](#_Toc182840907)

[ПРИЛОЖЕНИЕ А 16](#_Toc182840908)

[ПРИЛОЖЕНИЕ В 19](#_Toc182840909)

[ПРИЛОЖЕНИЕ С 23](#_Toc182840910)

[ПРИЛОЖЕНИЕ D 29](#_Toc182840911)

[ПРИЛОЖЕНИЕ Е 32](#_Toc182840912)

# **Общие положения выполнения лабораторных работ**

1. Лабораторные работы выполнены на языке программирования Python 3.11.2 с использованием встроенных библиотек:
   1. numpy – библиотека со встроенными математическими операциями линейной алгебры
   2. os – библиотека для работы с системным окружением
   3. json – библиотека для работы с файлами в формате .json
   4. math – библиотека содержащая примитивные математические функции
   5. argparse – библиотека для работы с параметрами командной строки при запуске программы
   6. re – библиотека для работы с регулярными выражениями
   7. minidom – библиотека для работы с файлами .xml
2. Каждое задание реализуется отдельной программой. Реализованные программы представлены в приложениях (A, B, C, D, E).
3. Программа запускается в консольном режиме с параметрами: «имя входного файла» «имя выходного файла», в некоторых программах при отсутствии входных данных применяются стандартные значения.
4. В программах реализованы необходимые проверки на корректность входных параметров, а также данных, изменяющихся в процессе работы программы. Ошибки могут как отображаться в терминале, так и выводится в отдельный файл. Реализовано игнорирование лишних пробелов и символов табуляции во входных файлах.
5. Кодировка файлов – UTF-8.

# **Задание 1: Создание ориентированного графа**

На входе:

Текстовый файл с описанием графа в виде списка дуг:

(a\_1, b\_1, n\_1), (a\_2, b\_2, n\_2), ..., (a\_k, b\_k, n\_k)

где a\_i - начальная вершина дуги i, b\_i - конечная вершина дуги i, n\_i - порядковый номер дуги в списке всех заходящих в вершину b\_i дуг.

На выходе:

1. Ориентированный граф с именованными вершинами и линейно упорядоченными дугами (в соответствии с порядком из текстового файла).

2. Сообщение об ошибке в формате файла, если ошибка присутствует.

3. Сериализованная структура графа в формате XML.

В программе была задана структура подаваемых дуг, которая обязательно должна содержать вершины в формате , где – номер вершины. В случае подачи вершин другого формата будет поднята ошибка и выполнение программы прекратится. Также реализованы проверки на количество элементов во входных файлах, так как передаются тройки элементов, то длина входных параметров должна быть кратна 3. Осуществляется проверка порядка входящих в вершину дуг, что позволяет предотвратить некорректную обработку в дальнейших заданиях. Программная реализация представлена в приложении A.

На вход программе подается множество текстовых файлов в следующем формате (См. Рис. 1):

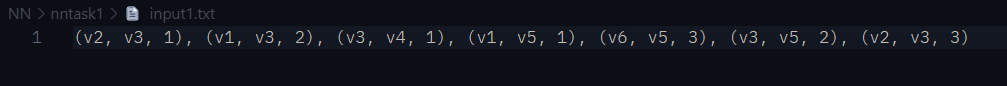


Рисунок 1 – Текстовый файл содержащий список дуг рассматриваемого графа

Протестируем программу nntask1.py на входном файле input1.txt и посмотрим на полученный результат. В результате работы программы получили сообщение в терминале о:

1. Рассматриваемом файле (в данном случае input1.txt)
2. Вершинах в графе
3. О структуре графа в результате обработки текстового файла
4. Название файла, в который сохранился граф в формате XML

Результаты отображены на рисунке 2:

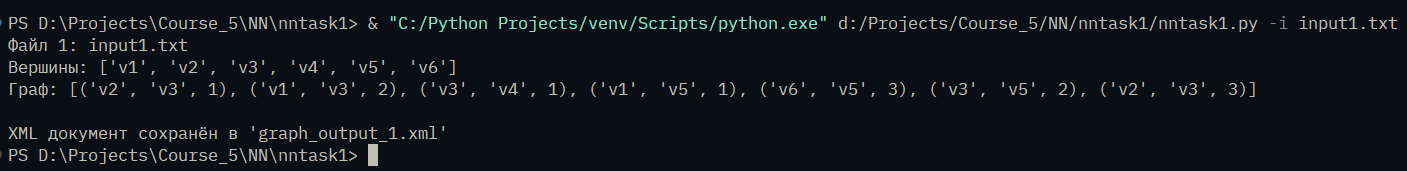
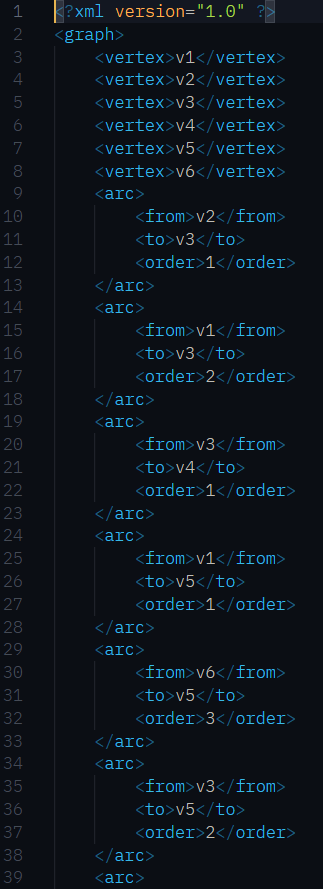


Рисунок 2 – Вывод информации в терминал при работе программы nntask1.py

Структура XML файла следующая (См. Рис. 3):



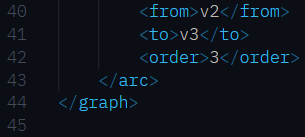


Рисунок 3 - Сериализованная структура графа в формате XML

Рассмотрим примеры с ошибками. В файл errors.txt записываются ошибки, и файл не очищается после перезапуска программы. Допустим следующие ошибки и посмотрим какие записи появятся в файле ошибок:

1. Неправильные значения вершин графа
2. Номер дуги не является числом или имеет отрицательное значение
3. Количество элементов во входящем файле не соответствует тройкам
4. Порядок входящих дуг в вершину нарушен

В результате получим следующий файл с ошибками (См. Рис 4):

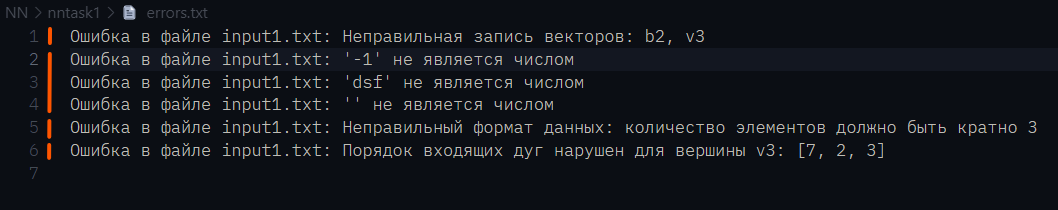


Рисунок 4 – Файл с ошибками при подаче некорректных значений в программу

# **Задание 2: Создание функции по графу**

На входе: ориентированный граф с именованными вершинами как описано в задании 1.

На выходе: линейное представление функции, реализуемой графом в префиксной скобочной записи:

A1(B1(C1(...),..., Cm(...)),..., Bn(...))

Способ проверки результата:

1. выгрузка в текстовый файл результата преобразования графа в имя функции.
2. сообщение о наличии циклов в графе, если они присутствуют.

Для построения префиксной записи будет использоваться метод обхода графа в глубину (DFS), этот же алгоритм будет использоваться для нахождения циклов в графе. Функция будет строится по степени вхождения в вершины, то есть если есть две вершины , , которые входят в вершину , тогда префиксная функция будет иметь вид , расположение вершин определяется порядком дуг входящих в вершину содержащие эти вершины. Вершины и могут представлять как константные значения (если в них не входит никакая дуга) или функцией (если в них входит какая-то дуга). Поэтому для корректного построения префиксной записи необходимо найти все вершины – стоки (вершины, из которых не исходят ни одной дуги). Так как у нас может быть несколько стоков в графе, то будут строится функции относительно каждого стока, потому на выходе может быть несколько функций. Префиксная функции записываются отдельно для каждого входного файла. Листинг программы представлен в приложении B. Рассмотрим работу программы на следующих входных файлах (См. Рис. 5, 6):



Рисунок 5 – Содержимое входного файла input3.txt

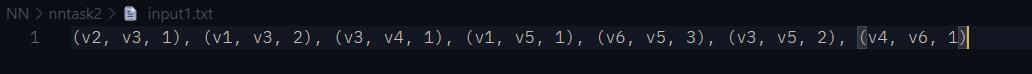


Рисунок 6 – Содержимое входного файла input2.txt

В результате работы программы получили следующее сообщение в терминале, которое отображает краткую информацию по проведенной работе (См. Рис. 7):

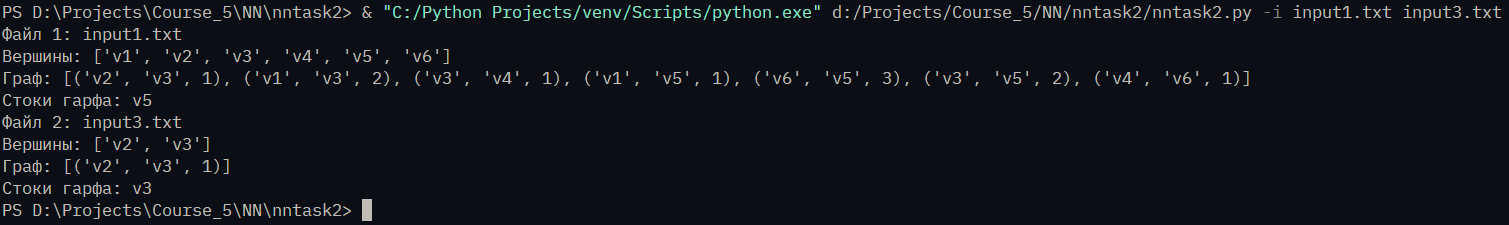


Рисунок 7 – Вывод информации о работе программы nntask2.py в терминал

Были созданы следующие файлы с префиксными функциями (См. Рис.8, 9):

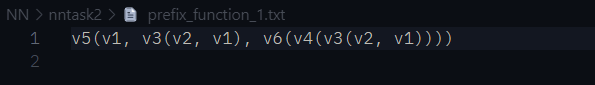


Рисунок 8 – Содержимое файла с префиксной формой первого графа



Рисунок 9 – Содержимое файла с префиксной формой второго графа

В случае если граф содержит цикл, то ошибка будет записана в файл errors.txt. Продемонстрируем работу по обнаружению цикла. Пусть существует цикл в первом входном файле (input1.txt) следующего вида: Тогда получим следующее сообщение в файле errors.txt (См. Рис. 10):

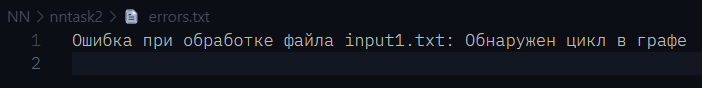
**

Рисунок 10 – Сообщение о наличии цикла в входном файле input1.txt

# **Задание 3: Вычисление значения функции на графе**

На входе:

1. Текстовый файл с описанием графа в виде списка дуг.
2. Текстовый файл соответствий арифметических операций именам вершин:

a\_1 : операция\_1

a\_2 : операция\_2

...

a\_n : операция\_n,

где a\_i - имя i-й вершины, операция\_i - символ операции, соответствующий вершине a\_i.

Допустимы следующие символы операций:

1. +– сумма значений,
2. \* – произведение значений,
3. exp – экспонирование входного значения,
4. число – любая числовая константа

На выходе: значение функции, построенной по графу а) и файлу б).

Способ проверки результата: результат вычислений, выведенный в файл.

Задача заключается в разборе префиксной записи функции и проверке корректности применяемых операций. Входные операции могут использоваться не все, зависит от того какие вершины использованы в префиксной записи графа. Также происходит обработка ошибок в случае, если в файле подаются неизвестные операции или некорректные значения констант. Программная реализация представлена в приложении C. Протестируем программу nntask3.py, на следующих входных значениях (См. Рис. 11, 12):

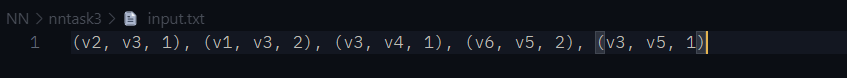


Рисунок 11 – Содержимое первого входящего файла input.txt

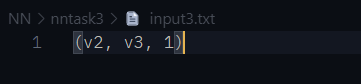


Рисунок 12 – Содержимое второго входящего файла input3.txt

Содержимое файлов операций следующее (См. Рис. 13):

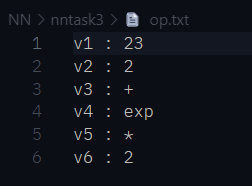


Рисунок 13 – Содержимое файла операций op.txt

В результате выполнения программы получим следующее сообщение в терминале (См. Рис. 14):

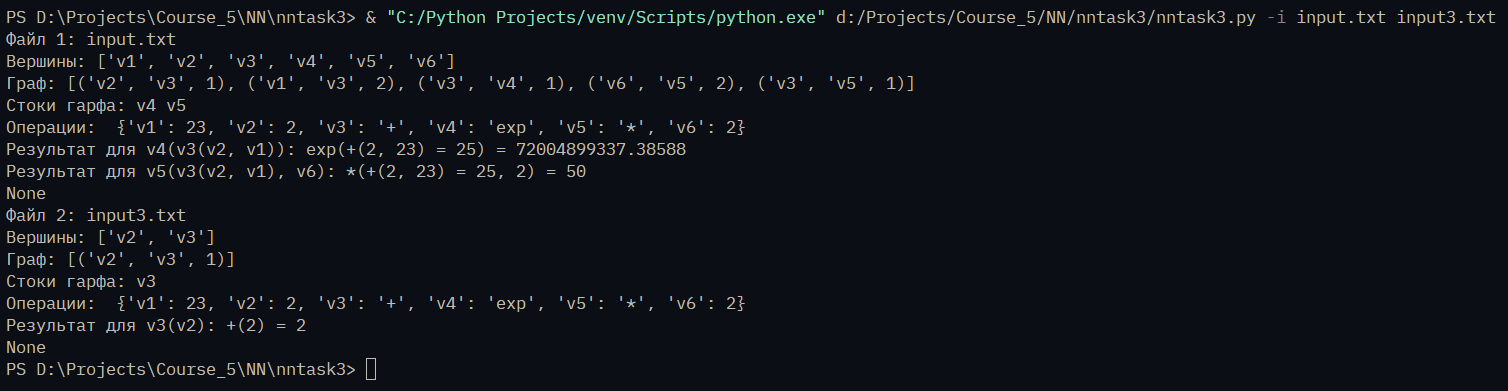


Рисунок 14 – Информация отображающееся в терминале при выполнении программы

В результате мы также получим выходные файлы со следующим содержимым (См. Рис. 15, 16):

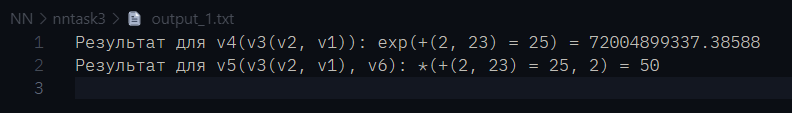


Рисунок 15 – Содержимое вычисленного значения префиксной функции для графа, представленного файлом input1.txt

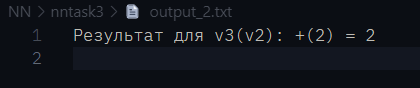


Рисунок 16 - Содержимое вычисленного значения префиксной функции для графа, представленного файлом input3.txt

Протестируем программу на некорректных данных. Протестируем ситуацию, при которой у нас во входном файле указаны неизвестные операции, значения функций заменяются значениями констант. Укажем во входном файле операций у вершины v3 значение 3 (изначально v3 было функцией). Результат тестирования представлен на рисунке 17:

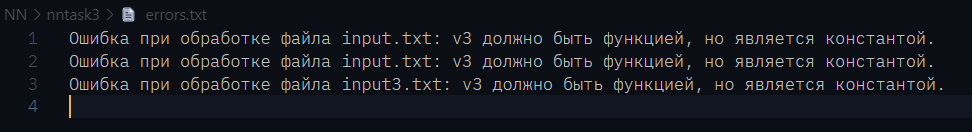


Рисунок 17 – Файл с ошибками при обработке некорректного применения вершин в префиксной функции

Протестируем теперь что будет, если указать неизвестную операцию (для v3 поставим значение -) (См. Рис. 18):

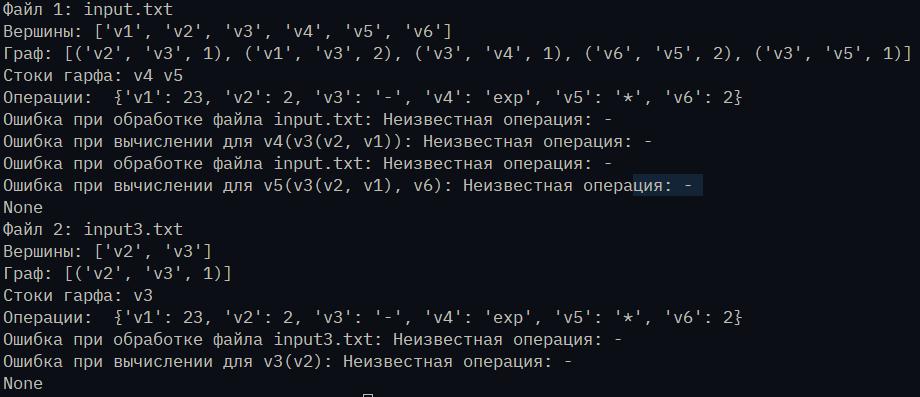


Рисунок 18 - Сообщение о том, что поступила неизвестная операция

# **Задание 4: Построение многослойной нейронной сети**

На входе:

1. Текстовый файл с набором матриц весов межнейронных связей:

М1 : [M1[1,1], M1[1,2],..., M1[1,n]], ..., [M1[m,1], M1[m,2],...,M1[m,n]]

М2 : [M2[1,1], M2[1,2],..., M2[1,n]], ..., [M2[m,1], M2[m,2],...,M2[m,n]]

...

Мp : [Mp[1,1], Mp[1,2],..., Mp[1,n]], ..., [Mp[m,1], Mp[m,2],...,Mp[m,n]]

1. Текстовый файл с входным вектором

На выходе:

1. Сериализованная многослойная нейронная сеть (в формате XML или JSON) с полносвязной межслойной структурой.
2. Файл с выходным вектором - результатом вычислений НС.
3. Сообщение об ошибке, если в формате входного вектора или файла описания НС допущена ошибка.

Сериализованная многослойная нейронная сеть в данной реализации будет сохранятся файл формата .json. В данной программе отслеживаются ошибки на несоответствие размерности векторов, которые в процессе выполнения программы необходимо умножать. Информация о ошибках записывается в отдельный файл. Программная реализация представлена в приложении D. Протестируем программу nntask4.py на следующих входных данных (См. Рис. 19, 20):

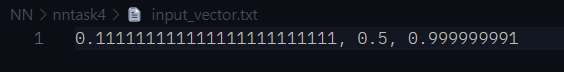


Рисунок 19 – Текстовый файл с входным вектором размерности 3

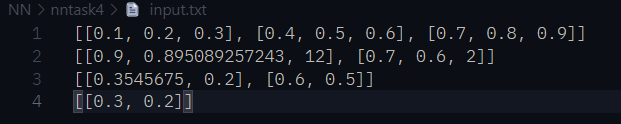


Рисунок 20 - Текстовый файл с набором матриц весов межнейронных связей

В результате получили следующее значение после вычисления (См. Рис. 21)

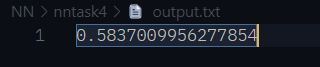


Рисунок 21 – Выходной файл после выполнения программы

Сериализованная многослойная нейронная сеть выглядит следующим образом (См. Рис. 22)

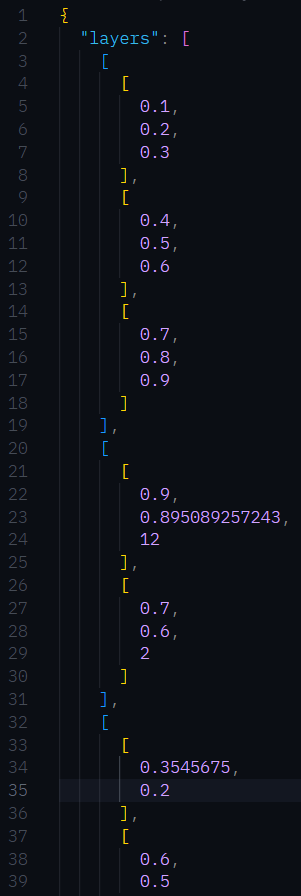


Рисунок 22 - Сериализованная многослойная нейронная сеть

# **Задание 5: Реализация метода обратного распространения ошибки для многослойной НС**

На входе:

а) Текстовый файл с описанием НС (формат см. в задании 4).

б) Текстовый файл с обучающей выборкой:

[x11, x12, ..., x1n] -> [y11, y12, ..., y1m]

...

[xk1, xk2, ..., xkn] -> [yk1, yk2, ..., ykm]

Формат описания входного вектора *x* и выходного вектора *y* соответствует формату из задания 4.

в) Число итераций обучения (в строке параметров).

На выходе:

Текстовый файл с историей N итераций обучения методом обратного распространения ошибки:

1 : Ошибка1

2 : Ошибка2

...

N : ОшибкаN

Всю программу можно поделить на две части, считывание данных из файлов, и вычисление функции обратного распространения ошибки, на основе считанных данных. Программная реализация представлена в приложении E. Рассмотрим выполнение программы nntask5.txt на следующих входных файлах (См. Рис. 23, 24)

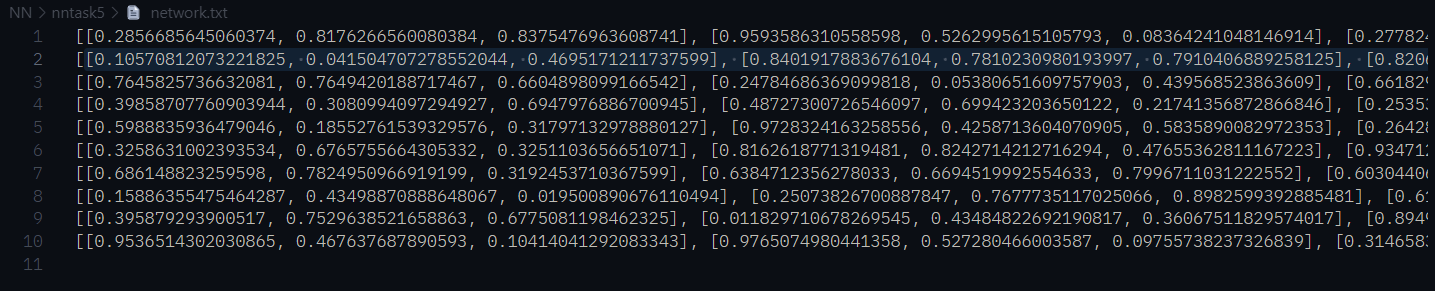


Рисунок 23 – Текстовый файл с описанием НС

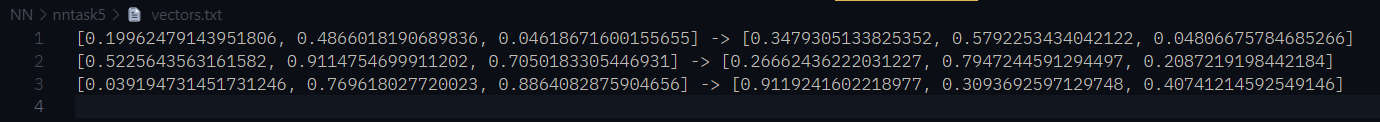


Рисунок 24 - Текстовый файл с обучающей выборкой

В результате тестирования получаем следующую информацию (См. Рис. 25, 26)

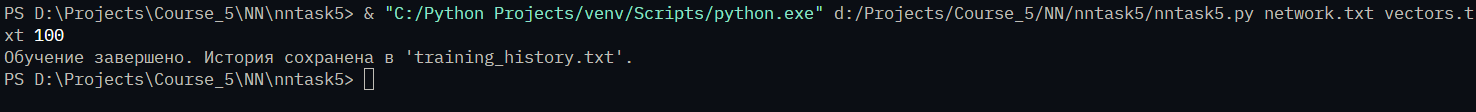


Рисунок 25 – Информация в терминале о успешном выполнении программы

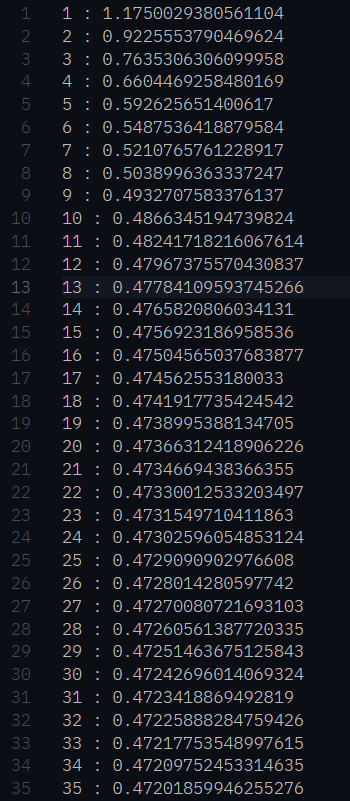


Рисунок 26 – Файл с ошибками 100 итераций

# **ПРИЛОЖЕНИЕ А**

**Программная реализация создания ориентированного графа по входным дугам**

import xml.etree.ElementTree as ET

import xml.dom.minidom as minidom

import argparse

def check\_correct\_data(data):

    graph = []

    vertices = set()

    incoming\_edges = {}

    if len(data) % 3 != 0:

        raise Exception("Неправильный формат данных:

количество элементов должно быть кратно 3")

    for i in range(0, len(data), 3):

        tuple\_ = data[i:i+3]

        v1 = tuple\_[0].strip()

        v2 = tuple\_[1].strip()

        n = tuple\_[2].strip()

        if not check\_vertex(v1) or not check\_vertex(v2):

            raise Exception(f"Неправильная запись векторов: {v1},

{v2}")

        if not n.isdigit():

            raise Exception(f"'{n}' не является числом")

        n = int(n)

        if n <= 0:

            raise Exception(f"Номер дуги должен быть

положительным числом: {n}")

        vertices.add(v1)

        vertices.add(v2)

        if (v1, v2, n) not in graph:

            graph.append((v1, v2, n))

        else:

            raise Exception(f"Ребро указано дважды: {(v1, v2, n)}")

        if v2 not in incoming\_edges:

            incoming\_edges[v2] = [n]

        else:

            incoming\_edges[v2].append(n)

    for v in incoming\_edges:

        max\_el = max(incoming\_edges[v])

        if sorted(incoming\_edges[v]) != list(range(1, max\_el + 1)):

            raise Exception(f"Порядок входящих дуг нарушен

для вершины {v}: {incoming\_edges[v]}")

    return sorted(list(vertices)), graph

def check\_vertex(vertex):

    # Вершина должна состоять из двух символов: 'v' и числа

    if len(vertex) < 2 or vertex[0] != "v" or not vertex[1:].isdigit():

        return False

    return True

def generate\_xml(vertices, graph):

    root = ET.Element("graph")

    for vertex in vertices:

        vertex\_elem = ET.SubElement(root, "vertex")

        vertex\_elem.text = vertex

    for v1, v2, n in graph:

        arc\_elem = ET.SubElement(root, "arc")

        from\_elem = ET.SubElement(arc\_elem, "from")

        from\_elem.text = v1

        to\_elem = ET.SubElement(arc\_elem, "to")

        to\_elem.text = v2

        order\_elem = ET.SubElement(arc\_elem, "order")

        order\_elem.text = str(n)

    xml\_string = ET.tostring(root, encoding='unicode', method='xml')

    pretty\_xml = minidom.parseString(xml\_string).toprettyxml(indent="    ")

    return pretty\_xml

def parse\_file(in\_file, i):

# Обрабатываем каждый файл

    data = read\_file(in\_file).strip().replace('(', '').replace(')', '').split(",")

    try:

        vertices, graph = check\_correct\_data(data)

        print(f"Файл {i}: {in\_file}")

        print("Вершины:", vertices)

        print("Граф:", graph)

        pretty\_xml\_data = generate\_xml(vertices, graph)

        with open(f"graph\_output\_{i}.xml", "w", encoding="UTF-8") as f:

            f.write(pretty\_xml\_data)

        print(f"\nXML документ сохранён в 'graph\_output\_{i}.xml'")

    except Exception as e:

        message = f"Ошибка в файле {in\_file}: {e}"

        print(message)

        with open("errors.txt", "a", encoding="UTF-8") as f:

            f.write(message + '\n')

def read\_file(file\_path):

    with open(file\_path, "r", encoding="UTF-8") as f:

        data = f.readline()

    return data

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Программа для

обработки множества графов.")

    parser.add\_argument('-i', nargs='+',

help='Входные файлы для обработки')

    args = parser.parse\_args()

    for i, data in enumerate(args.i):

        parse\_file(data, i + 1)

# **ПРИЛОЖЕНИЕ В**

**Программная реализация создания функции по заданному графу**

import argparse

# Проверка корректности записи вершин

def check\_vertex(vertex):

    # Вершина должна состоять из двух символов: 'v' и числа

    if len(vertex) < 2 or vertex[0] != "v" or not vertex[1:].isdigit():

        return False

    return True

# Чтение файла с графом

def read\_file(file\_path):

    with open(file\_path, "r", encoding="UTF-8") as f:

        data = f.readline()

    return data

# Чтение графа из файла серилизация

def read\_graph(data):

    graph = []

    vertices = set()

    incoming\_edges = {}

    if len(data) % 3 != 0:

        raise Exception("Неправильный формат данных:

количество элементов должно быть кратно 3")

    for i in range(0, len(data), 3):

        tuple\_ = data[i:i+3]

        v1 = tuple\_[0].strip()

        v2 = tuple\_[1].strip()

        n = tuple\_[2].strip()

        if not check\_vertex(v1) or not check\_vertex(v2):

            raise Exception(f"Неправильная запись векторов: {v1}, {v2}")

        if not n.isdigit():

            raise Exception(f"'{n}' не является числом")

        n = int(n)

        if n <= 0:

            raise Exception(f"Номер дуги должен быть

положительным числом: {n}")

        vertices.add(v1)

        vertices.add(v2)

        if (v1, v2, n) not in graph:

            graph.append((v1, v2, n))

        else:

            raise Exception(f"Ребро указано дважды: {(v1, v2, n)}")

        if v2 not in incoming\_edges:

            incoming\_edges[v2] = [n]

        else:

            incoming\_edges[v2].append(n)

    for v in incoming\_edges:

        max\_el = max(incoming\_edges[v])

        if sorted(incoming\_edges[v]) != list(range(1, max\_el + 1)):

            raise Exception(f"Порядок входящих дуг

нарушен для вершины {v}: {incoming\_edges[v]}")

    return sorted(list(vertices)), graph

#

def get\_dict\_graph(edges):

    graph = {}

    for v1, v2, num in edges:

        if v2 in graph:

            graph[v2].append((v1, num))

        else:

            graph[v2] = [(v1, num)]

    return graph

def find\_stok(graph, vertices):

    # Все вершины, которые являются началом хотя бы одной дуги

    start\_vertices = set()

    for key in graph:

        for i in range(len(graph[key])):

            start\_vertices.add(graph[key][i][0])

    all\_vertices = set(vertices)

    no\_outgoing\_edges = all\_vertices - start\_vertices

    return sorted(list(no\_outgoing\_edges))

# Функция для построения графа из списка рёбер

def build\_graph(edges, vertices):

    graph = {v:[] for v in vertices}

    for v1, v2, \_ in edges:

        graph[v1].append(v2)

    return graph

# Рекурсивная функция DFS для поиска цикла

def dfs\_for\_find\_cycle(vertex, graph, visited, rec\_stack):

    visited.add(vertex)

    rec\_stack.add(vertex)

    for neighbor in graph[vertex]:

        if neighbor in rec\_stack:

            return True

        elif neighbor not in visited:

            if dfs\_for\_find\_cycle(neighbor, graph, visited, rec\_stack):

                return True

    # Убираем вершину из стека рекурсивного вызова перед возвратом

    rec\_stack.remove(vertex)

    return False

# Основная функция для проверки наличия цикла

def has\_cycle(edges, vertices):

    graph = build\_graph(edges, vertices)

    visited = set()

    # Проверяем каждую вершину, чтобы учитывать несвязные компоненты

    for vertex in graph:

        if vertex not in visited:

            rec\_stack = set()

            if dfs\_for\_find\_cycle(vertex, graph, visited, rec\_stack):

                return True

    return False

def build\_prefix\_function(graph, start):

    if start not in graph:

        return start

    sorted\_edges = sorted(graph[start], key=lambda x: x[1])

    inner\_parts = [build\_prefix\_function(graph, v)

for v, \_ in sorted\_edges]

    return f"{start}({', '.join(inner\_parts)})"

# Основная функция для построения префиксной записи для всех вершин без исходящих рёбер

def generate\_prefix\_functions(graph, root\_vertices):

    result = []

    for root in root\_vertices:

        result.append(build\_prefix\_function(graph, root))

    return result

def parse\_file(in\_file, i, out\_file):

# Обрабатываем каждый файл

    data = read\_file(in\_file).strip().replace('(', '').replace(')', '').split(",")

    try:

        vertices, edges = read\_graph(data)

        print(f"Файл {i}: {in\_file}")

        print("Вершины:", vertices)

        print("Граф:", edges)

        if has\_cycle(edges, vertices):

            raise Exception("Обнаружен цикл в графе")

        graph = get\_dict\_graph(edges)

        roots = find\_stok(graph, vertices)

        print("Стоки гарфа:", \*roots)

        prefix\_funcs = generate\_prefix\_functions(graph, roots)

        with open(out\_file[:out\_file.index(".")] + f"\_{i}.txt" , "w",

encoding="UTF-8") as f:

            for p\_fun in prefix\_funcs:

                f.write(p\_fun + "\n")

    except Exception as e:

        message = f"Ошибка при обработке файла {in\_file}: {e}"

        print(message)

        with open("errors.txt", "a", encoding="UTF-8") as f:

            f.write(message + '\n')

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Программа для

обработки множества графов.")

    parser.add\_argument('-i', nargs='+',

help='Входные файлы для обработки')

    parser.add\_argument('-o', default="prefix\_function.txt",

help='Имя выходного файла

(по умолчанию: default\_output.txt)')

    args = parser.parse\_args()

    out\_file = args.o

    for i, data in enumerate(args.i):

        parse\_file(data, i + 1, out\_file)

# **ПРИЛОЖЕНИЕ С**

**Программная реализация вычисления значения функции на графе**

import argparse

import math

import re

# Проверка корректности записи вершин

def check\_vertex(vertex):

    # Вершина должна состоять из двух символов: 'v' и числа

    if len(vertex) < 2 or vertex[0] != "v" or not vertex[1:].isdigit():

        return False

    return True

# Чтение файла с графом

def read\_file(file\_path):

    with open(file\_path, "r", encoding="UTF-8") as f:

        data = f.readline()

    return data

# Чтение графа из файла серилизация

def read\_graph(data):

    graph = []

    vertices = set()

    incoming\_edges = {}

    if len(data) % 3 != 0:

        raise Exception("Неправильный формат данных:

количество элементов должно быть кратно 3")

    for i in range(0, len(data), 3):

        tuple\_ = data[i:i+3]

        v1 = tuple\_[0].strip()

        v2 = tuple\_[1].strip()

        n = tuple\_[2].strip()

        if not check\_vertex(v1) or not check\_vertex(v2):

            raise Exception(f"Неправильная запись векторов: {v1}, {v2}")

        if not n.isdigit():

            raise Exception(f"'{n}' не является числом")

        n = int(n)

        if n <= 0:

            raise Exception(f"Номер дуги должен быть

положительным числом: {n}")

        vertices.add(v1)

        vertices.add(v2)

        if (v1, v2, n) not in graph:

            graph.append((v1, v2, n))

        else:

            raise Exception(f"Ребро указано дважды: {(v1, v2, n)}")

        if v2 not in incoming\_edges:

            incoming\_edges[v2] = [n]

        else:

            incoming\_edges[v2].append(n)

    for v in incoming\_edges:

        max\_el = max(incoming\_edges[v])

        if sorted(incoming\_edges[v]) != list(range(1, max\_el + 1)):

            raise Exception(f"Порядок входящих дуг нарушен

для вершины {v}: {incoming\_edges[v]}")

    return sorted(list(vertices)), graph

#

def get\_dict\_graph(edges):

    graph = {}

    for v1, v2, num in edges:

        if v2 in graph:

            graph[v2].append((v1, num))

        else:

            graph[v2] = [(v1, num)]

    return graph

def find\_stok(graph, vertices):

    # Все вершины, которые являются началом хотя бы одной дуги

    start\_vertices = set()

    for key in graph:

        for i in range(len(graph[key])):

            start\_vertices.add(graph[key][i][0])

    # Все вершины графа, включая начальные и конечные

    all\_vertices = set(vertices)

    # Вершины, из которых не исходит ни одной дуги

    no\_outgoing\_edges = all\_vertices - start\_vertices

    return sorted(list(no\_outgoing\_edges))

# Функция для построения графа из списка рёбер

def build\_graph(edges, vertices):

    graph = {v:[] for v in vertices}

    for v1, v2, \_ in edges:

        graph[v1].append(v2)

    return graph

# Рекурсивная функция DFS для поиска цикла

def dfs\_for\_find\_cycle(vertex, graph, visited, rec\_stack):

    visited.add(vertex)

    rec\_stack.add(vertex)

    for neighbor in graph[vertex]:

        if neighbor in rec\_stack:

            return True

        elif neighbor not in visited:

            if dfs\_for\_find\_cycle(neighbor, graph, visited, rec\_stack):

                return True

    # Убираем вершину из стека рекурсивного вызова перед возвратом

    rec\_stack.remove(vertex)

    return False

def has\_cycle(edges, vertices):

    graph = build\_graph(edges, vertices)

    visited = set()

    for vertex in graph:

        if vertex not in visited:

            rec\_stack = set()

            if dfs\_for\_find\_cycle(vertex, graph, visited, rec\_stack):

                return True

    return False

def build\_prefix\_function(graph, start):

    if start not in graph:

        return start

    sorted\_edges = sorted(graph[start], key=lambda x: x[1])

    inner\_parts = [build\_prefix\_function(graph, v) for v, \_ in sorted\_edges]

    return f"{start}({', '.join(inner\_parts)})"

def generate\_prefix\_functions(graph, root\_vertices):

    result = []

    for root in root\_vertices:

        result.append(build\_prefix\_function(graph, root))

    return result

# Функция для выполнения операции

def evaluate\_operation(operation, args):

    if operation == '+':

        return sum(args)

    elif operation == '\*':

        result = 1

        for arg in args:

            result \*= arg

        return result

    elif operation == 'exp':

        if len(args) != 1:

            raise ValueError("Функция exp должна

принимать только один аргумент.")

        return math.exp(args[0])

    else:

        raise ValueError(f"Неизвестная операция: {operation}")

# Функция для подстановки значений из словаря и вычисления выражения

def substitute\_values(expression, operations):

    # Рекурсивная функция для разбора и вычисления

    def recursive\_evaluate(expr):

        expr = expr.strip()

        # Если это просто вершина, возвращаем значение из словаря

        if expr in operations:

            value = operations[expr]

            if isinstance(value, str):  # Если это функция, а не константа

                raise ValueError(f"{expr} должно быть функцией,

но является константой.")

            return value, str(value)

        # Ищем функцию с аргументами

        match = re.match(r'([a-zA-Z0-9\_]+)\((.\*)\)', expr)

        if match:

            node = match.group(1)

            if node not in operations:

                raise ValueError(f"Неизвестная вершина

{node} в выражении.")

            operation = operations[node]

            if isinstance(operation, (int, float)):

                raise ValueError(f"{node} должно быть функцией,

но является константой.")

            # Разбираем аргументы функции, учитывая вложенность

            args\_str = match.group(2)

            args = []

            bracket\_level = 0

            current\_arg = []

            for char in args\_str:

                if char == ',' and bracket\_level == 0:

                    args.append(''.join(current\_arg).strip())

                    current\_arg = []

                else:

                    if char == '(':

                        bracket\_level += 1

                    elif char == ')':

                        bracket\_level -= 1

                    current\_arg.append(char)

            args.append(''.join(current\_arg).strip())

            # Рекурсивно вычисляем каждый аргумент

            evaluated\_args = []

            substituted\_args = []

            for arg in args:

                eval\_result, sub\_expr = recursive\_evaluate(arg)

                evaluated\_args.append(eval\_result)

                substituted\_args.append(sub\_expr)

            result = evaluate\_operation(operation, evaluated\_args)

            # Формируем строку подстановки для итогового вывода

            substituted\_expression = f"{operation}(" + ", ".join(substituted\_args) + f") = {result}"

            return result, substituted\_expression

        else:

            raise ValueError(f"Неправильное выражение: {expr}")

    # Выполняем подстановку для выражения

    \_, full\_expression = recursive\_evaluate(expression)

    return full\_expression

# Пример использования

def process\_graph(graph, operations, out\_file, i, in\_file):

    with open(out\_file[:out\_file.index(".")] + f"\_{i}.txt", "w",

encoding="UTF-8") as f:

        for expr in graph:

            try:

                result = substitute\_values(expr, operations)

                str\_ = f"Результат для {expr}: {result}"

                f.write(str\_ + "\n")

                print(str\_)

            except Exception as e:

                message = f"Ошибка при обработке файла {in\_file}: {e}"

                print(message)

                with open("errors.txt", "a", encoding="UTF-8") as f:

                    f.write(message + '\n')

                print(f"Ошибка при вычислении для {expr}: {e}")

def load\_operations(file\_path):

    operations = {}

    # Открываем файл для чтения

    with open(file\_path, 'r') as f:

        for line in f:

            if line:

                key\_value = line.split(':')

                if len(key\_value) == 2:

                    key = key\_value[0].strip()

                    value = key\_value[1].strip()

                    try:

                        value = int(value)

                    except (ValueError, SyntaxError):

                        pass

                    operations[key] = value

    return operations

def parse\_file(in\_file, i, out\_file, operation\_file):

# Обрабатываем каждый файл

    data = read\_file(in\_file).strip().replace('(', '').replace(')', '').split(",")

    try:

        vertices, edges = read\_graph(data)

        print(f"Файл {i}: {in\_file}")

        print("Вершины:", vertices)

        print("Граф:", edges)

        if has\_cycle(edges, vertices):

            raise Exception("Обнаружен цикл в графе")

        graph = get\_dict\_graph(edges)

        roots = find\_stok(graph, vertices)

        print("Стоки гарфа:", \*roots)

        prefix\_funcs = generate\_prefix\_functions(graph, roots)

        operations = load\_operations(operation\_file)

        print("Операции: ", operations)

        print(process\_graph(prefix\_funcs, operations,

out\_file, i, in\_file))

    except Exception as e:

        message = f"Ошибка при обработке файла {in\_file}: {e}"

        print(message)

        with open("errors.txt", "a", encoding="UTF-8") as f:

            f.write(message + '\n')

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Программа для

обработки множества графов.")

    parser.add\_argument('-i', nargs='+',

help='Входные файлы для обработки')

    parser.add\_argument('-o', default="output.txt",

help='Имя выходного файла (по умолчанию:

output.txt)')

    parser.add\_argument('-op', default="op.txt",

help='Имя файла операций (по умолчанию: op.txt)')

    args = parser.parse\_args()

    out\_file = args.o

    operation\_file = args.op

    for i, data in enumerate(args.i):

        parse\_file(data, i + 1, out\_file, operation\_file)

# **ПРИЛОЖЕНИЕ D**

**Реализация прямого распространения через многослойную нейронную сеть**

import argparse

import json

import math

import os

# Чтение вектора из файла

def read\_vector(file\_path):

    try:

        with open(file\_path, 'r') as file:

            return list(map(float, file.readline().strip().split(',')))

    except Exception as e:

        raise ValueError(f"Ошибка чтения входного вектора: {e}")

# Чтение состояния нейронной сети (весов) из файла

def read\_nn\_state(file\_path):

    try:

        with open(file\_path, 'r') as file:

            weights = []

            for line in file:

                weights.append(json.loads(line.strip()))

            return weights

    except Exception as e:

        raise ValueError(f"Ошибка чтения состояния сети: {e}")

# Сигмоидная функция активации

def sigmoid(z, c):

    return 1.0 / (1.0 + math.exp(-c \* z))

# Прямое распространение

def forward\_pass(weights, input\_vector, c=1):

    activations = input\_vector

    for layer in weights:

        if len(activations) != len(layer[0]):

            raise ValueError("Длина входного вектора

не совпадает с матрицей весов")

        next\_activations = []

        for neuron\_weights in layer:

            z = sum(w \* a for w, a in zip(neuron\_weights, activations))

            next\_activations.append(sigmoid(z, c))

        activations = next\_activations

    return activations

# Запись выходного вектора в файл

def write\_output(output\_vector, file\_path):

    try:

        with open(file\_path, 'w') as file:

            file.write(', '.join(map(str, output\_vector)))

    except Exception as e:

        raise ValueError(f"Ошибка записи выходного вектора: {e}")

# Сериализация состояния сети

def serialize\_nn(weights, file\_path):

    try:

        with open(file\_path, 'w') as file:

            json.dump({'layers': weights}, file, indent=2)

    except Exception as e:

        raise ValueError(f"Ошибка сериализации сети: {e}")

def main(args):

    error\_log = "errors.txt"

    if os.path.exists(error\_log):

        os.remove(error\_log)

    for idx, nn\_file in enumerate(args.nn\_files, start=1):

        try:

            weights = read\_nn\_state(nn\_file)

            input\_vector = read\_vector(args.input\_vector)

            output\_vector = forward\_pass(weights, input\_vector)

            if len(args.nn\_files) > 1:

                output\_file = f"output\_{idx}.txt"

                output\_network\_file = f"outputNetwork\_{idx}.json"

            else:

                output\_file = args.output\_file

                output\_network\_file = args.output\_network

            write\_output(output\_vector, output\_file)

            serialize\_nn(weights, output\_network\_file)

            print(f"Результат успешно сохранён для {nn\_file}:")

            print(f"  Выходной вектор: {output\_file}")

            print(f"  Сериализация сети: {output\_network\_file}")

        except ValueError as e:

            with open(error\_log, 'a', encoding="UTF-8") as error\_file:

                error\_file.write(f"Ошибка в файле {nn\_file}: {e}\n")

            print(f"Ошибка обработки {nn\_file}: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

    parser = argparse.ArgumentParser(description="Программа для прямого

распространения в нейронной сети.")

    parser.add\_argument(

        '-i', '--nn\_files', nargs='+', default=['input.txt'],

        help="Список файлов с весами нейронной сети

(по умолчанию: input.txt)"

    )

    parser.add\_argument(

        '--input\_vector', default='input\_vector.txt',

        help="Файл с входным вектором (по умолчанию: input\_vector.txt)"

    )

    parser.add\_argument(

        '--output\_file', default='output.txt',

        help="Файл для записи выходного вектора (по умолчанию:

output.txt)"

    )

    parser.add\_argument(

        '--output\_network', default='outputNetwork.json',

        help="Файл для сериализации состояния сети

(по умолчанию: outputNetwork.json)"

    )

    args = parser.parse\_args()

    main(args)

# **ПРИЛОЖЕНИЕ Е**

**Реализация алгоритма обратного распространения ошибки и обучения нейронной сети**

import numpy as np

# Функция активации (сигмоида) и её производная

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Загрузка нейронной сети из файла

def load\_network(file\_path):

    with open(file\_path, "r") as file:

        lines = file.readlines()

        layers = [eval(line.strip()) for line in lines]

    return [np.array(layer) for layer in layers]

# Загрузка обучающей выборки

def load\_training\_data(file\_path):

    with open(file\_path, "r") as file:

        lines = file.readlines()

        data = []

        for line in lines:

            x, y = line.strip().split("->")

            x = eval(x.strip())

            y = eval(y.strip())

            data.append((np.array(x), np.array(y)))

    return data

# Метод обратного распространения ошибки

def train\_network(network, training\_data, iterations, output\_file="training\_history.txt"):

    history = []

    for iteration in range(1, iterations + 1):

        total\_error = 0

        for x, y in training\_data:

            # Прямое распространение

            activations = [x]

            for layer in network:

                activations.append(sigmoid(np.dot(activations[-1],

layer)))

            # Ошибка выходного слоя

            error = y - activations[-1]

            total\_error += np.sum(error \*\* 2)

            # Обратное распространение

            deltas = [error \* sigmoid\_derivative(activations[-1])]

            for i in range(len(network) - 1, 0, -1):

                delta = deltas[-1].dot(network[i].T) \* \

sigmoid\_derivative(activations[i])

                deltas.append(delta)

            deltas.reverse()

            # Обновление весов

            for i in range(len(network)):

                network[i] += activations[i][:, np.newaxis] @

deltas[i][np.newaxis, :]

        # Сохранение ошибки для текущей итерации

        history.append(f"{iteration} : {total\_error}")

    # Запись истории обучения в файл

    with open(output\_file, "w") as file:

        file.write("\n".join(history))

# Основная программа

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    if len(sys.argv) != 4:

        print("Использование: python nntask5.py

<network\_file> <training\_file> <iterations>")

        sys.exit(1)

    # Параметры командной строки

    network\_file = sys.argv[1]

    training\_file = sys.argv[2]

    iterations = int(sys.argv[3])

    # Загрузка данных

    network = load\_network(network\_file)

    training\_data = load\_training\_data(training\_file)

    # Обучение нейронной сети

    train\_network(network, training\_data, iterations)

    print("Обучение завершено.

История сохранена в 'training\_history.txt'.")