

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
Задание 1: Создание ориентированного графа	4
Задание 2: Создание функции по графу.....	7
Задание 3: Вычисление значения функции на графе	9
Задание 4: Построение многослойной нейронной сети	12
Задание 5: Реализация метода обратного распространения ошибки для многослойной НС.....	14
ПРИЛОЖЕНИЕ А	16
ПРИЛОЖЕНИЕ В	19
ПРИЛОЖЕНИЕ С	23
ПРИЛОЖЕНИЕ D	29
ПРИЛОЖЕНИЕ E.....	32

Общие положения выполнения лабораторных работ

1. Лабораторные работы выполнены на языке программирования Python 3.11.2 с использованием встроенных библиотек:

- 1.1. `numpy` – библиотека со встроенными математическими операциями линейной алгебры
- 1.2. `os` – библиотека для работы с системным окружением
- 1.3. `json` – библиотека для работы с файлами в формате `.json`
- 1.4. `math` – библиотека содержащая примитивные математические функции
- 1.5. `argparse` – библиотека для работы с параметрами командной строки при запуске программы
- 1.6. `re` – библиотека для работы с регулярными выражениями
- 1.7. `minidom` – библиотека для работы с файлами `.xml`

2. Каждое задание реализуется отдельной программой. Реализованные программы представлены в приложениях (А, В, С, D, Е).

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 выводиться 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:

$$(a_1, b_1, n_1), (a_2, b_2, n_2), \dots, (a_k, b_k, n_k)$$

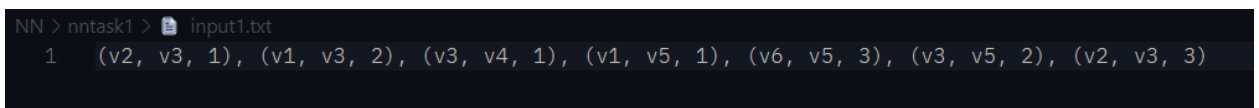
где a_i - начальная вершина дуги i , b_i - конечная вершина дуги i , n_i - порядковый номер дуги в списке всех заходящих в вершину b_i дуг.

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 v_i , где i – 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 А.

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



```
NN > nntask1 > input1.txt
1 (v2, v3, 1), (v1, v3, 2), (v3, v4, 1), (v1, v5, 1), (v6, v5, 3), (v3, v5, 2), (v2, v3, 3)
```

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:

```
PS D:\Projects\Course_5\NN\nntask1> & "C:/Python Projects/venv/Scripts/python.exe" d:/Projects/Course_5/NN/nntask1/nntask1.py -i input1.txt
Файл 1: input1.txt
Вершины: ['v1', 'v2', 'v3', 'v4', 'v5', 'v6']
Граф: [('v2', 'v3', 1), ('v1', 'v3', 2), ('v3', 'v4', 1), ('v1', 'v5', 1), ('v6', 'v5', 3), ('v3', 'v5', 2), ('v2', 'v3', 3)]

XML документ сохранён в 'graph_output_1.xml'
PS D:\Projects\Course_5\NN\nntask1>
```

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

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

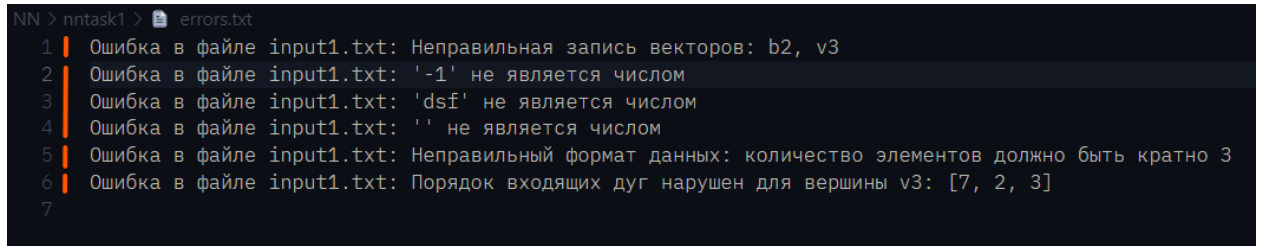
```
1  <?xml version="1.0" ?>
2  <graph>
3      <vertex>v1</vertex>
4      <vertex>v2</vertex>
5      <vertex>v3</vertex>
6      <vertex>v4</vertex>
7      <vertex>v5</vertex>
8      <vertex>v6</vertex>
9      <arc>
10         <from>v2</from>
11         <to>v3</to>
12         <order>1</order>
13     </arc>
14     <arc>
15         <from>v1</from>
16         <to>v3</to>
17         <order>2</order>
18     </arc>
19     <arc>
20         <from>v3</from>
21         <to>v4</to>
22         <order>1</order>
23     </arc>
24     <arc>
25         <from>v1</from>
26         <to>v5</to>
27         <order>1</order>
28     </arc>
29     <arc>
30         <from>v6</from>
31         <to>v5</to>
32         <order>3</order>
33     </arc>
34     <arc>
35         <from>v3</from>
36         <to>v5</to>
37         <order>2</order>
38     </arc>
39     <arc>
40         <from>v2</from>
41         <to>v3</to>
42         <order>3</order>
43     </arc>
44 </graph>
45
```

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):



```
NN > nntask1 > errors.txt
1 | Ошибка в файле input1.txt: Неправильная запись векторов: b2, v3
2 | Ошибка в файле input1.txt: '-1' не является числом
3 | Ошибка в файле input1.txt: 'dsf' не является числом
4 | Ошибка в файле input1.txt: '' не является числом
5 | Ошибка в файле input1.txt: Неправильный формат данных: количество элементов должно быть кратно 3
6 | Ошибка в файле input1.txt: Порядок входящих дуг нарушен для вершины v3: [7, 2, 3]
7
```

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.

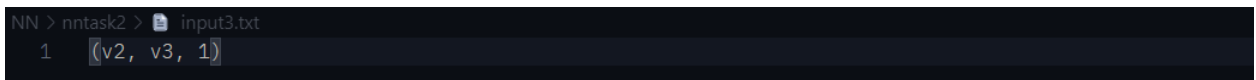
На выходе: линейное представление функции, реализуемой графом в префиксной скобочной записи:

$$A1(B1(C1(...),..., C_m(...)),..., B_n(...))$$

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 строиться по степени вхождения в вершины, то есть если есть две вершины $v1$, $v2$, которые входят в вершину $v3$, тогда префиксная функция будет иметь вид $v3(v1, v2)$, расположение вершин $v1, v2$ определяется порядком дуг входящих в вершину $v3$ содержащие эти вершины. Вершины $v1$ и $v2$ 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 строиться функции относительно каждого стока, потому на выходе может быть несколько функций. Префиксная функции записываются отдельно для каждого входного файла. Листинг программы представлен в приложении В. Рассмотрим работу программы на следующих входных файлах (См. Рис. 5, 6):



```
NN > nntask2 > input3.txt
1 [v2, v3, 1]
```

Рисунок 5 – Содержимое входного файла input3.txt

```

NN > nntask2 > input1.txt
1 (v2, v3, 1), (v1, v3, 2), (v3, v4, 1), (v1, v5, 1), (v6, v5, 3), (v3, v5, 2), (v4, v6, 1)

```

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):

```

PS D:\Projects\Course_5\NN\nntask2> "C:/Python Projects/venv/Scripts/python.exe" d:/Projects/Course_5/NN/nntask2/nntask2.py -i input1.txt input3.txt
Файл 1: input1.txt
Вершины: ['v1', 'v2', 'v3', 'v4', 'v5', 'v6']
Граф: [('v2', 'v3', 1), ('v1', 'v3', 2), ('v3', 'v4', 1), ('v1', 'v5', 1), ('v6', 'v5', 3), ('v3', 'v5', 2), ('v4', 'v6', 1)]
Стоки графа: v5
Файл 2: input3.txt
Вершины: ['v2', 'v3']
Граф: [('v2', 'v3', 1)]
Стоки графа: v3
PS D:\Projects\Course_5\NN\nntask2>

```

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

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

```

NN > nntask2 > prefix_function_1.txt
1 v5(v1, v3(v2, v1), v6(v4(v3(v2, v1))))
2

```

Рисунок 8 – Содержимое файла с префиксной формой первого графа

```

NN > nntask2 > prefix_function_2.txt
1 v3(v2)
2

```

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: $v2 \rightarrow v3 \rightarrow v2$. Тогда получим следующее сообщение в файле errors.txt (См. Рис. 10):

```

NN > nntask2 > errors.txt
1 Ошибка при обработке файла input1.txt: Обнаружен цикл в графе
2

```

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.

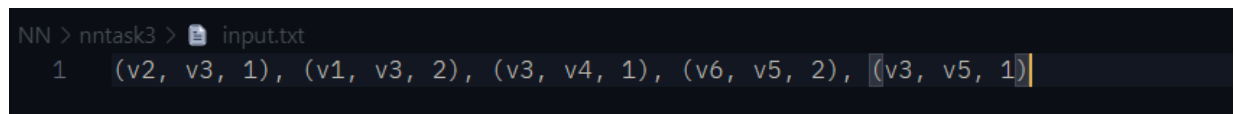
Допустимы следующие символы операций:

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 С. Протестируем программу nntask3.py, на следующих входных значениях (См. Рис. 11, 12):



```
NN > nntask3 > input.txt
1 (v2, v3, 1), (v1, v3, 2), (v3, v4, 1), (v6, v5, 2), (v3, v5, 1)
```

Рисунок 11 – Содержимое первого входящего файла input.txt

```
NN > nntask3 > input3.txt
1 (v2, v3, 1)
```

Рисунок 12 – Содержимое второго входящего файла input3.txt

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

```
NN > nntask3 > op.txt
1 v1 : 23
2 v2 : 2
3 v3 : +
4 v4 : exp
5 v5 : *
6 v6 : 2
```

Рисунок 13 – Содержимое файла операций op.txt

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

```
PS D:\Projects\Course_5\NN\nntask3> & "C:/Python Projects/venv/Scripts/python.exe" d:/Projects/Course_5/NN/nntask3/nntask3.py -i input.txt input3.txt
Файл 1: input.txt
Вершины: ['v1', 'v2', 'v3', 'v4', 'v5', 'v6']
Граф: [('v2', 'v3', 1), ('v1', 'v3', 2), ('v3', 'v4', 1), ('v6', 'v5', 2), ('v3', 'v5', 1)]
Стоки графа: v4 v5
Операции: {'v1': 23, 'v2': 2, 'v3': '+', 'v4': 'exp', 'v5': '*', 'v6': 2}
Результат для v4(v3(v2, v1)): exp(+(2, 23) = 25) = 72004899337.38588
Результат для v5(v3(v2, v1), v6): *(+(2, 23) = 25, 2) = 50
None
Файл 2: input3.txt
Вершины: ['v2', 'v3']
Граф: [('v2', 'v3', 1)]
Стоки графа: v3
Операции: {'v1': 23, 'v2': 2, 'v3': '+', 'v4': 'exp', 'v5': '*', 'v6': 2}
Результат для v3(v2): +(2) = 2
None
PS D:\Projects\Course_5\NN\nntask3>
```

Рисунок 14 – Информация отображающееся в терминале при выполнении программы

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

```
NN > nntask3 > output_1.txt
1 Результат для v4(v3(v2, v1)): exp(+(2, 23) = 25) = 72004899337.38588
2 Результат для v5(v3(v2, v1), v6): *(+(2, 23) = 25, 2) = 50
3
```

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

```

NN > nntask3 > output_2.txt
1  Результат для v3(v2): +(2) = 2
2

```

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:

```

NN > nntask3 > errors.txt
1  Ошибка при обработке файла input.txt: v3 должно быть функцией, но является константой.
2  Ошибка при обработке файла input.txt: v3 должно быть функцией, но является константой.
3  Ошибка при обработке файла input3.txt: v3 должно быть функцией, но является константой.
4

```

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):

```

Файл 1: input.txt
Вершины: ['v1', 'v2', 'v3', 'v4', 'v5', 'v6']
Граф: [('v2', 'v3', 1), ('v1', 'v3', 2), ('v3', 'v4', 1), ('v6', 'v5', 2), ('v3', 'v5', 1)]
Стоки гарфа: v4 v5
Операции: {'v1': 23, 'v2': 2, 'v3': '-', 'v4': 'exp', 'v5': '*', 'v6': 2}
Ошибка при обработке файла input.txt: Неизвестная операция: -
Ошибка при вычислении для v4(v3(v2, v1)): Неизвестная операция: -
Ошибка при обработке файла input.txt: Неизвестная операция: -
Ошибка при вычислении для v5(v3(v2, v1), v6): Неизвестная операция: -
None
Файл 2: input3.txt
Вершины: ['v2', 'v3']
Граф: [('v2', 'v3', 1)]
Стоки гарфа: v3
Операции: {'v1': 23, 'v2': 2, 'v3': '-', 'v4': 'exp', 'v5': '*', 'v6': 2}
Ошибка при обработке файла input3.txt: Неизвестная операция: -
Ошибка при вычислении для v3(v2): Неизвестная операция: -
None

```

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:

$M1 : [M1[1,1], M1[1,2], \dots, M1[1,n]], \dots, [M1[m,1], M1[m,2], \dots, M1[m,n]]$

$M2 : [M2[1,1], M2[1,2], \dots, M2[1,n]], \dots, [M2[m,1], M2[m,2], \dots, M2[m,n]]$

...

$Mp : [Mp[1,1], Mp[1,2], \dots, Mp[1,n]], \dots, [Mp[m,1], Mp[m,2], \dots, Mp[m,n]]$

2. Текстовый файл с входным вектором

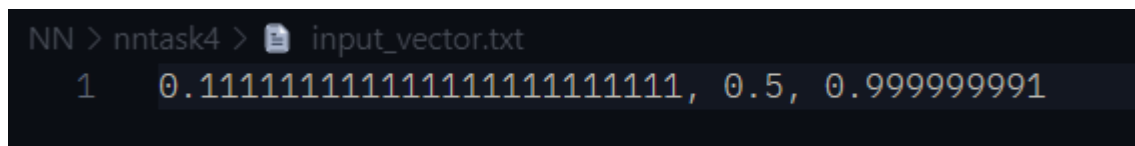
На выходе:

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 сохраняться 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):



```
NN > nntask4 > input_vector.txt
1 0.11111111111111111111111111111111, 0.5, 0.999999991
```

Рисунок 19 – Текстовый файл с входным вектором размерности 3

```

NN > nntask4 > input.txt
1  [[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]]
2  [[0.9, 0.895089257243, 12], [0.7, 0.6, 2]]
3  [[0.3545675, 0.2], [0.6, 0.5]]
4  [[0.3, 0.2]]

```

Рисунок 20 - Текстовый файл с набором матриц весов межнейронных связей

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

```

NN > nntask4 > output.txt
1  0.5837009956277854

```

Рисунок 21 – Выходной файл после выполнения программы

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

```

1  {
2    "layers": [
3      [
4        [
5          0.1,
6          0.2,
7          0.3
8        ],
9        [
10         0.4,
11         0.5,
12         0.6
13       ],
14       [
15         0.7,
16         0.8,
17         0.9
18       ]
19     ],
20     [
21       [
22         0.9,
23         0.895089257243,
24         12
25       ],
26       [
27         0.7,
28         0.6,
29         2
30       ]
31     ],
32     [
33       [
34         0.3545675,
35         0.2
36       ],
37       [
38         0.6,
39         0.5

```

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:

$[x_{11}, x_{12}, \dots, x_{1n}] \rightarrow [y_{11}, y_{12}, \dots, y_{1m}]$

...

$[x_{k1}, x_{k2}, \dots, x_{kn}] \rightarrow [y_{k1}, y_{k2}, \dots, y_{km}]$

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 : Ошибка N

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 Е. Рассмотрим выполнение программы `nntask5.txt` на следующих входных файлах (См. Рис. 23, 24)

```
NN > nntask5 > network.txt
1  [[0.2856685645060374, 0.8176266560080384, 0.8375476963608741], [0.9593586310558598, 0.5262995615105793, 0.08364241048146914], [0.27782
2  [[0.10570812073221825, 0.041504707278552044, 0.4695171211737599], [0.8401917883676104, 0.7810230980193997, 0.7910406889258125], [0.820
3  [[0.7645825736632081, 0.7649420188717467, 0.6604898099166542], [0.24784686369099818, 0.05380651609757903, 0.439568523863609], [0.66182
4  [[0.39858707760903944, 0.3080994097294927, 0.6947976886700945], [0.48727300726546097, 0.699423203650122, 0.21741356872866846], [0.2535
5  [[0.5988835936479046, 0.18552761539329576, 0.31797132978880127], [0.9728324163258556, 0.4258713604070905, 0.5835890082972353], [0.2642
6  [[0.3258631002393534, 0.6765755664305332, 0.3251103656651071], [0.8162618771319481, 0.8242714212716294, 0.47655362811167223], [0.93471
7  [[0.686148823259598, 0.7824950966919199, 0.3192453710367599], [0.6384712356278033, 0.6694519992554633, 0.7996711031222552], [0.6030440
8  [[0.15886355475464287, 0.43498870888648067, 0.019500890676110494], [0.25073826700887847, 0.7677735117025066, 0.8982599392885481], [0.6
9  [[0.395879293900517, 0.7529638521658863, 0.6775081198462325], [0.011829710678269545, 0.43484822692190817, 0.36067511829574017], [0.894
10 [[0.9536514302030865, 0.467637687890593, 0.10414041292083343], [0.9765074980441358, 0.527280466003587, 0.09755738237326839], [0.314658
11
```

Рисунок 23 – Текстовый файл с описанием НС

```

NN > nntask5 > vectors.txt
1 [0.19962479143951806, 0.4866018190689836, 0.04618671600155655] -> [0.3479305133825352, 0.5792253434042122, 0.04806675784685266]
2 [0.5225643563161582, 0.9114754699911202, 0.7050183305446931] -> [0.26662436222031227, 0.7947244591294497, 0.2087219198442184]
3 [0.039194731451731246, 0.769618027720023, 0.8864082875904656] -> [0.9119241602218977, 0.3093692597129748, 0.40741214592549146]
4

```

Рисунок 24 - Текстовый файл с обучающей выборкой

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

```

PS D:\Projects\Course_5\NN\nntask5> & "C:/Python Projects/venv/Scripts/python.exe" d:/Projects/Course_5/NN/nntask5/nntask5.py network.txt vectors.t
xt 100
Обучение завершено. История сохранена в 'training_history.txt'.
PS D:\Projects\Course_5\NN\nntask5> 

```

Рисунок 25 – Информация в терминале о успешном выполнении программы

```

1 1 : 1.1750029380561104
2 2 : 0.9225553790469624
3 3 : 0.7635306306099958
4 4 : 0.6604469258480169
5 5 : 0.592625651400617
6 6 : 0.5487536418879584
7 7 : 0.5210765761228917
8 8 : 0.5038996363337247
9 9 : 0.4932707583376137
10 10 : 0.4866345194739824
11 11 : 0.48241718216067614
12 12 : 0.47967375570430837
13 13 : 0.47784109593745266
14 14 : 0.4765820806034131
15 15 : 0.4756923186958536
16 16 : 0.47504565037683877
17 17 : 0.474562553180033
18 18 : 0.4741917735424542
19 19 : 0.4738995388134705
20 20 : 0.47366312418906226
21 21 : 0.4734669438366355
22 22 : 0.47330012533203497
23 23 : 0.4731549710411863
24 24 : 0.47302596054853124
25 25 : 0.4729090902976608
26 26 : 0.4728014280597742
27 27 : 0.47270080721693103
28 28 : 0.47260561387720335
29 29 : 0.47251463675125843
30 30 : 0.47242696014069324
31 31 : 0.4723418869492819
32 32 : 0.47225888284759426
33 33 : 0.47217753548997615
34 34 : 0.47209752453314635
35 35 : 0.47201859946255276

```

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 сериализация
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 графа:", *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 сериализация
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):
    # Обработываем 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 графа:", *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'.")

```