

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ
компьютерной безопасности и
криптографии

НЕЙРОННЫЕ СЕТИ

ОТЧЕТ ПО ПРАКТИЧЕСКОМУ КУРСУ

студента 5 курса 531 группы

факультета компьютерных наук и информационных технологий

Ежовой Елены Дмитриевны

фамилия, имя, отчество

Научный руководитель

Ст. преподаватель

подпись, дата

И.И. Слеповичев

Саратов 2024

ПРАКТИЧЕСКАЯ РАБОТА

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:

- а) Ориентированный граф с именованными вершинами и линейно упорядоченными дугами (в соответствии с порядком из текстового файла).
- б) Сообщение об ошибке в формате файла, если ошибка присутствует.

Способ проверки результата:

- а) Сериализованная структура графа в формате XML или JSON.

Пример:

```
<graph>
  <vertex>v1</vertex>
  <vertex>v2</vertex>
  <vertex>v3</vertex>
  <arc>
    <from>v1</from>
    <to>v3</to>
    <order>1</order>
  </arc>
  <arc>
    <from>v2</from>
    <to>v3</to>
    <order>2</order>
  </arc>
</graph>
```

- б) Сообщение об ошибке с указанием номера строки с ошибкой во входном файле.

Примеры

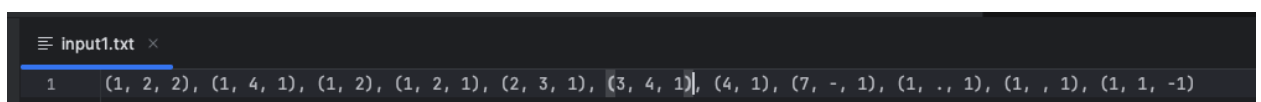


Рис. 1 – Входные данные программы

```
python nntask1.py input1=input1.txt output1=output1.json
Входные файлы: input1.txt, None
Выходные файлы: output1.json, None
Граф успешно сохранен в output1.json
Вершина 1: [('4', 1), ('2', 1), ('2', 2)]
Вершина 2: [('3', 1)]
Вершина 4: []
Вершина 3: [('4', 1)]
```

Рис. 2 – Запуск программы



```
output1.json x
JSON file
2      "1": [
3          ["4", 1],
4          ["2", 1],
5          ["2", 2]
6      ],
7      "2": [
8          ["3", 1]
9      ],
10     "4": [],
11     "3": [
12         ["4", 1]
13     ]
14 }
```

Рис. 3 – Результат работы программы

Листинг кода

```
class DirectedGraph:
    def __init__(self, file_path):
        self.file_path = file_path
        self.graph = {}
        self.error_file = "error_" + file_path
        self.first_error_log = True # Флаг для перезаписи файла при
        первом вызове
        self.load_graph()

    def log_error(self, message):
        # Перезаписываем файл при первом вызове, затем добавляем новые
        записи
        mode = 'w' if self.first_error_log else 'a'
        with open(self.error_file, mode, encoding='utf-8') as
        error_file:
            error_file.write(message + "\n")
            self.first_error_log = False # Меняем режим на добавление
            после первого вызова
```

```

def load_graph(self):
    try:
        with open(self.file_path, 'r', encoding='utf-8') as file:
            data = file.read().strip() # Считываем весь файл
одной строкой
            data = data[1:-1] # Убираем первую и последнюю скобки
            edges = data.split('), (') # Разбиваем на отдельные
рёбра

            for line_number, edge in enumerate(edges, start=1):
                parts = [part.strip() for part in edge.split(',') ]
# Разбиваем каждое ребро на части
                if len(parts) != 3 or not all(parts):
                    error_message = f"Строка {line_number}:
'({edge})'"
                    self.log_error(error_message + ' - не хватает
данных')
                    continue # Переходим к следующей строке

                a, b, n = parts[0], parts[1], parts[2]

                # Проверяем, что a и b являются числами, а n также
                # должен быть числом, если присутствует
                if not (a.isdigit() and b.isdigit() and
(n.isdigit() or n == '-')):
                    error_message = f"Строка {line_number}:
'({edge})'"
                    self.log_error(error_message + ' -
некорректные данные')
                    continue

                n = int(n) if n.isdigit() else None # Преобразуем
n в число, если возможно

                # Проверяем существование вершин и добавляем в
граф
                if a not in self.graph:
                    self.graph[a] = []
                if b not in self.graph and n is not None:
                    self.graph[b] = []

                # Добавляем дугу, если n не None
                if n is not None:
                    self.graph[a].append((b, n))

            # Сортировка исходящих дуг по порядковому номеру для
каждой вершины
            for node, edges in self.graph.items():
                self.graph[node] = sorted(edges, key=lambda x:
x[1])

    except FileNotFoundError:
        self.log_error(f"Файл {self.file_path} не найден.")
    except Exception as e:
        self.log_error(f"Произошла непредвиденная ошибка: {e}")

def save_graph_as_json(self, output_file):
    # Преобразуем граф в словарь для записи в JSON

```

```

graph_dict = {node: edges for node, edges in
self.graph.items()}

try:
    # Создаем компактный JSON без лишних отступов для
    # вложенных списков
    compact_json = json.dumps(graph_dict, ensure_ascii=False)

    # Добавляем отступы на верхнем уровне вручную
    formatted_json = (
        compact_json
        .replace(']', [' ', ']', '\n          [') # Отступ между
элементами верхнего уровня
        .replace(']]', "]", '\n          ]', '\n          "') # Перенос
строки после завершения списка для ключа
        .replace('": [' , '": [\n          ') # Начало верхнего
уровня с отступом
        .replace('}}', '\n}') # Закрывающая скобка JSON
        .replace('{', '{\n          ') # Начало JSON с отступом
    )

    with open(output_file, 'w', encoding='utf-8') as file:
        file.write(formatted_json)

    print(f"Граф успешно сохранен в {output_file}")
except Exception as e:
    self.log_error(f"Ошибка при сохранении графа в файл
{output_file}: {e}")

```

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:

а) выгрузка в текстовый файл результата преобразования графа в имя функции.

б) сообщение о наличии циклов в графе, если они присутствуют.

Примеры

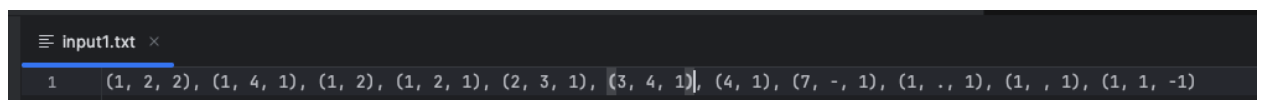


Рис. 4 – Входные данные программы

```
python nntask2.py input1=input1.txt output1=output1.txt
Входные файлы: input1.txt, None
Выходные файлы: output1.txt, None
Циклы не обнаружены в графе.
Вершина 1: [('4', 1), ('2', 1), ('2', 2)]
Вершина 2: [('3', 1)]
Вершина 4: []
Вершина 3: [('4', 1)]
Префиксное представление графа: 4(1, 3(2(1, 1)))
Результат сохранён в файл: output1.txt
```

Рис. 5 – Запуск программы

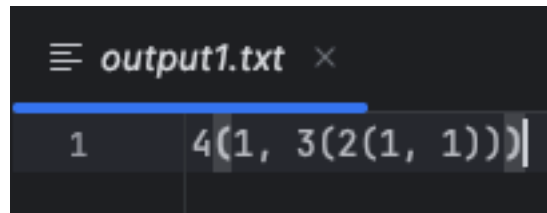


Рис. 6 – Результат работы программы

Листинг кода

```
import json
import re

class DirectedGraph:
    def __init__(self, file_path):
        self.file_path = file_path
        self.graph = {}
        self.error_file = "error_" + file_path
        self.first_error_log = True # Флаг для перезаписи файла при
        первом вызове
        self.load_graph()

    def log_error(self, message):
        # Перезаписываем файл при первом вызове, затем добавляем новые
        записи
        mode = 'w' if self.first_error_log else 'a'
        with open(self.error_file, mode, encoding='utf-8') as
        error_file:
            error_file.write(message + "\n")
            self.first_error_log = False # Меняем режим на добавление
            после первого вызова

    def load_graph(self):
        try:
            with open(self.file_path, 'r', encoding='utf-8') as file:
                data = file.read().strip() # Считываем весь файл
                одной строкой
                data = data[1:-1] # Убираем первую и последнюю скобки
                рёбра
                edges = data.split('), (') # Разбиваем на отдельные

            for line_number, edge in enumerate(edges, start=1):
                parts = [part.strip() for part in edge.split(',')]
```

```

        error_message = f"Строка {line_number}:
'({edge})'"
        self.log_error(error_message + ' - не хватает
данных')
        continue # Переходим к следующей строке

        a, b, n = parts[0], parts[1], parts[2]

        # Проверяем, что a и b являются числами, а n также
должен быть числом, если присутствует
        if not (a.isdigit() and b.isdigit() and
(n.isdigit() or n == '-')):
            error_message = f"Строка {line_number}:
'({edge})'"
            self.log_error(error_message + ' -
некорректные данные')
            continue

        n = int(n) if n.isdigit() else None # Преобразуем
n в число, если возможно

        # Проверяем существование вершин и добавляем в
граф
        if a not in self.graph:
            self.graph[a] = []
        if b not in self.graph and n is not None:
            self.graph[b] = []

        # Добавляем дугу, если n не None
        if n is not None:
            self.graph[a].append((b, n))

        # Сортировка исходящих дуг по порядковому номеру для
каждой вершины
        for node, edges in self.graph.items():
            self.graph[node] = sorted(edges, key=lambda x:
x[1])

    except FileNotFoundError:
        self.log_error(f"Файл {self.file_path} не найден.")
    except Exception as e:
        self.log_error(f"Произошла непредвиденная ошибка: {e}")

    def save_graph_as_json(self, output_file):
        # Преобразуем граф в словарь для записи в JSON
        graph_dict = {node: edges for node, edges in
self.graph.items()}

        try:
            # Создаем компактный JSON без лишних отступов для
вложенных списков
            compact_json = json.dumps(graph_dict, ensure_ascii=False)

            # Добавляем отступы на верхнем уровне вручную
            formatted_json = (
                compact_json
                .replace('[', [' ', ''], \n          [') # Отступ между
элементами верхнего уровня

```

```

        .replace(']]', '"', ']\n    ],\n    "') # Перенос
строки после завершения списка для ключа
        .replace('": [' , '": [\n    ') # Начало верхнего
уровня с отступом
        .replace('}}', '\n}') # Закрывающая скобка JSON
        .replace('{', '{\n    ') # Начало JSON с отступом
    )

    with open(output_file, 'w', encoding='utf-8') as file:
        file.write(formatted_json)

    print(f"Граф успешно сохранен в {output_file}")
except Exception as e:
    self.log_error(f"Ошибка при сохранении графа в файл
{output_file}: {e}")

def has_cycle(self):
    visited = set()
    rec_stack = set()

    # обход в глубину
    def dfs(v):
        visited.add(v)
        rec_stack.add(v)

        for neighbor, _ in self.graph.get(v, []):
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in rec_stack:
                print("Цикл обнаружен в графе.")
                return True

        rec_stack.remove(v)
        return False

    # Запуск DFS для каждой компоненты связности
    for vertex in self.graph:
        if vertex not in visited:
            if dfs(vertex):
                return True

    print("Циклы не обнаружены в графе.")
    return False

def find_sink(self):
    for node, edges in self.graph.items():
        if not edges:
            return node
    return None

def build_function(self, node):
    # Инициализация списка для хранения входящих рёбер
    incoming = []
    # Перебор всех рёбер графа
    for parent, edges in self.graph.items():
        # Перебор рёбер для каждой вершины
        for child, order in edges:

```



```

        # Если текущее ребро ведёт к искомой вершине
        if child == node:
            # Добавляем исходящую вершину и порядок ребра в
            # список входящих
            incoming.append((parent, order))
        # Сортировка входящих рёбер по порядку
        incoming.sort(key=lambda x: x[1])

        # Если список входящих рёбер пуст, возвращаем текущую вершину
        if not incoming:
            return node

        # Рекурсивное построение функций для каждой исходящей вершины
        children = [self.build_function(parent) for parent, _ in
incoming]
        # Формирование и возвращение строки вызова текущей вершины как
        # функции от её детей
        return f"{node}({' , '.join(children)})"

    def to_prefix_notation(self):
        sink = self.find_sink()
        if not sink:
            return "Невозможно построить функцию"
        return self.build_function(sink)

    def save_prefix_notation_to_file(self, file_path):
        with open(file_path, 'w') as file:
            notation = self.to_prefix_notation()
            file.write(notation)
            print(f"Результат сохранён в файл: {file_path}")

    def display_graph(self):
        # Вывод графа для проверки
        for node, edges in self.graph.items():
            print(f"Вершина {node}: {edges}")

```

3. Вычисление значения функции на графе

На входе:

- а) Текстовый файл с описанием графа в виде списка дуг (смотри задание 1).
- б) 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:

$+$ – сумма значений,

$*$ – произведение значений,

exp – экспонирование входного значения,

число – любая числовая константа.

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

Способ проверки результата: результат вычисления, выведенный в файл.

Примеры

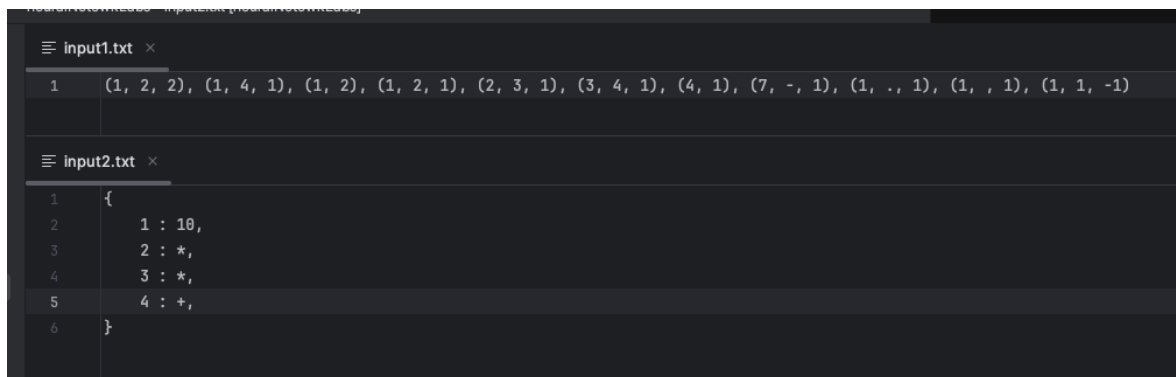


Рис. 7 – Входные данные программы

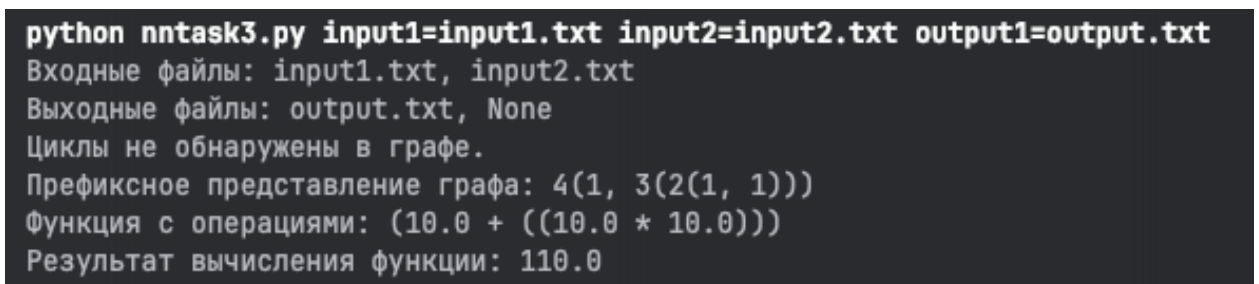


Рис. 8 – Запуск программы

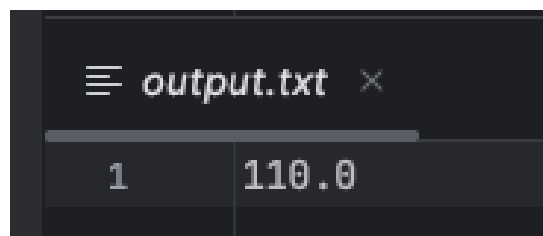


Рис. 9 – Результат работы программы

Листинг кода

```
from helpers.file_handler import parse_args
from graph import DirectedGraph
import math
```

```

import re

def load_operations_from_file(operations_file):
    operations = {}
    try:
        with open(operations_file, 'r') as file:
            for line_number, line in enumerate(file, 1):
                line = line.strip()

                # Игнорируем строки с фигурными скобками
                if line == '{' or line == '}':
                    continue

                try:
                    # Разделяем строку по ':', чтобы получить имя и
операцию
                    name, operation = line.split(':')
                    name = name.strip()
                    operation = operation.strip().rstrip(',') #
Убираем запятую в конце

                    # Если операция является числом, преобразуем её в
float
                    if re.match(r"^-\d+(\.\d+)?$", operation): #
Проверка на числовую константу
                        operation = float(operation) # Преобразуем в
число

                    operations[name] = operation
                except ValueError:
                    print(f"Ошибка в формате данных на строке
{line_number}, строка пропущена")
                except FileNotFoundError:
                    print(f"Файл не найден: {operations_file}")
    return operations

def save_result(output_file, result ):
    with open(output_file, "w") as file:
        file.write(str(result))

class DirectedGraphWithOperations(DirectedGraph):
    def __init__(self, file_path, operations_file):
        super().__init__(file_path)
        self.operations = load_operations_from_file(operations_file)

    def to_prefix_with_operations(self):
        """
        Метод преобразует граф операций в префиксную запись, где
каждая вершина представлена операцией
или значением.
        """

    def dfs_with_operations(node):
        """
        Рекурсивный обход графа для построения выражения в

```

префиксной записи.

```

        :param node: текущая вершина графа
        :return: строковое представление выражения для данной
вершины
        """
        # Получаем операцию или значение для текущей вершины
        operation = self.operations.get(node, node)

        # Если операция числовая, возвращаем её как строку
        if isinstance(operation, (int, float)):
            return str(operation)

        # Получаем все входящие рёбра для текущей вершины
        incoming = []
        for parent, edges in self.graph.items():
            for child, order in edges:
                if child == node:
                    incoming.append((parent, order))
        # Сортируем входящие рёбра по порядковому номеру (order)
        incoming.sort(key=lambda x: x[1])

        # Рекурсивно строим строковые выражения для всех входящих
вершин
        children_str = [dfs_with_operations(parent) for parent, _
in incoming]

        # Формируем строку в зависимости от операции
        if operation == '+':
            # Суммируем значения дочерних вершин
            return f"({' + '.join(children_str)})"
        elif operation == '*':
            # Перемножаем значения дочерних вершин
            return f"({' * '.join(children_str)})"
        elif operation == 'exp':
            # Проверяем, что операция экспоненты имеет ровно один
аргумент
            if len(children_str) != 1:
                raise ValueError("Операция 'exp' должна иметь
ровно один аргумент")
            return f"exp({children_str[0]})"
        else:
            # Если операция неизвестна, возвращаем её как строку
            return str(operation)

        # Находим вершину без исходящих рёбер (синк)
        sink = self.find_sink()
        if not sink:
            # Если синк не найден, выбрасываем исключение
            raise ValueError("Не удалось найти конечную вершину
графа.")

        # Начинаем построение выражения с найденного синка
        return dfs_with_operations(sink)

def evaluate_function(self):
    """
    Метод вычисляет значение функции, представленной графом

```

операций.

Обходит граф начиная с конечной вершины (синка) и применяет операции рекурсивно.

```
"""
def evaluate(node):
    """
    Рекурсивное вычисление значения для указанной вершины
    графа.

    :param node: текущая вершина графа
    :return: результат вычисления для данной вершины
    """
    # Получаем операцию для данной вершины
    operation = self.operations.get(node)
    # Если операция не найдена, выбрасываем исключение
    if operation is None:
        raise ValueError(f"Операция для вершины {node} не
найдена")

    # Если операция является числовой константой, возвращаем
её значение
    if isinstance(operation, (int, float)):
        return operation

    # Инициализируем список для хранения входящих рёбер
    incoming = []
    # Перебираем все рёбра графа
    for parent, edges in self.graph.items():
        # Перебираем рёбра для текущей вершины
        for child, order in edges:
            # Если ребро ведёт к искомой вершине, добавляем
его

            if child == node:
                incoming.append((parent, order))
    # Сортируем входящие рёбра по порядковому номеру (order)
    incoming.sort(key=lambda x: x[1])

    # Рекурсивно вычисляем значения для всех дочерних вершин
    children_values = [evaluate(parent) for parent, _ in
incoming]

    # Применяем операцию в зависимости от её типа
    if operation == "+":
        # Сложение всех дочерних значений
        return sum(children_values)
    elif operation == "*":
        # Умножение всех дочерних значений
        result = 1
        for value in children_values:
            result *= value
        return result
    elif operation == "exp":
        # Вычисление экспоненты (e^x) для одного аргумента
        if len(children_values) != 1:
            # Логируем ошибку и выбрасываем исключение, если
аргументов не один

            self.log_error("Операция 'exp' должна иметь ровно
```

```

один аргумент")
        raise ValueError("Операция 'exp' должна иметь
ровно один аргумент")
    return math.exp(children_values[0])
    else:
        # Если операция неизвестна, логируем ошибку и
выбрасываем исключение
        self.log_error(f"Неизвестная операция '{operation}'
для вершины {node}")
        raise ValueError(f"Неизвестная операция '{operation}'
для вершины {node}")

    # Находим вершину без исходящих рёбер (синк)
    sink = self.find_sink()
    if not sink:
        # Если конечная вершина (синк) не найдена, выбрасываем
исключение
        raise ValueError("Не удалось найти конечную вершину
графа.")

    # Начинаем вычисление с найденной конечной вершины
    return evaluate(sink)

```

4. Построение многослойной нейронной сети

На входе:

а) Текстовый файл с набором матриц весов межнейронных связей:

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

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

...

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

б) Текстовый файл с входным вектором в формате:

x_1, x_2, \dots, x_n .

На выходе:

а) Сериализованная многослойная нейронная сеть (в формате XML или JSON) с полносвязной межслойной структурой.

б) Файл с выходным вектором – результатом вычислений НС в формате:

y_1, y_2, \dots, y_n .

в) Сообщение об ошибке, если в формате входного вектора или файла описания НС допущена ошибка.

Примеры

input1.txt	
1	[15, 27, 7, 1, 86, 4], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [18, 1, 89, 66, 72, 52], [1, 2, 3, 4, 5, 6], [41, 98, 8, 31, 87, 54]
2	[1, 2, 3, 4, 5, 6], [22, 62, 10, 52, 2, 24], [1, 2, 3, 4, 5, 6], [50, 1, 57, 1, 39, 1]
3	[97, 96, 70, 65], [1, 2, 3, 4]
input2.txt	
1	1 46 2 17 32 4

Рис. 10 – Входные данные программы

```
python nntask4.py input1=input1.txt input2=input2.txt output1=output1.json output2=output2.txt
Входные файлы: input1.txt, input2.txt
Выходные файлы: output1.json, output2.txt
Структура сети успешно сохранена в output1.json
Результат успешно сохранен в output2.txt
```

Рис. 11 – Запуск программы

output1.json	
1	{
2	"Layer 1": [
3	[15, 27, 7, 1, 86, 4],
4	[1, 2, 3, 4, 5, 6],
5	[1, 2, 3, 4, 5, 6],
6	[18, 1, 89, 66, 72, 52],
7	[1, 2, 3, 4, 5, 6],
8	[41, 98, 8, 31, 87, 54]
9],
10	"Layer 2": [
11	[1, 2, 3, 4, 5, 6],
12	[22, 62, 10, 52, 2, 24],
13	[1, 2, 3, 4, 5, 6],
14	[50, 1, 57, 1, 39, 1]
15],
16	"Layer 3": [
17	[97, 96, 70, 65],
18	[1, 2, 3, 4]
19]
20	}
Layer 1	
output2.txt	
1	[
2	1.0,
3	0.9999546021311599
4]

Рис. 12 – Результат работы программы

Листинг кода

```
import json
import numpy as np
from helpers.file_handler import parse_args
import math

class NeuralNetwork:
    def __init__(self, weights_file, input_vector_file):
        self.weights_file = weights_file
        self.input_vector_file = input_vector_file
        self.error_file = f"error_{weights_file}"
        self.first_error_log = True
        self.layers = []
        self.input_vector = []
        self.network_structure = {}
```

```

        self.load_weights()
        self.load_input_vector()

    def log_error(self, message):
        mode = 'w' if self.first_error_log else 'a'
        with open(self.error_file, mode, encoding='utf-8') as
error_file:
            error_file.write(message + "\n")
            self.first_error_log = False

    def load_weights(self):
        try:
            with open(self.weights_file, 'r', encoding='utf-8') as
file:
                for layer_number, line in enumerate(file, start=1):
                    line = line.strip()
                    if not line:
                        continue
                    try:
                        # Интерпретируем строку как список нейронов
(каждый нейрон — список весов)
                        layer = json.loads(f"[{line}]")
                        self.layers.append(layer)
                        self.network_structure[f"Layer
{layer_number}"] = layer
                    except Exception:
                        self.log_error(f"Строка {layer_number}:
некорректный формат матрицы весов '{line}'")
                    except FileNotFoundError:
                        self.log_error(f"Файл {self.weights_file} не найден.")
                    except Exception as e:
                        self.log_error(f"Произошла ошибка при загрузке весов:
{e}")

        def load_input_vector(self):
            try:
                with open(self.input_vector_file, 'r', encoding='utf-8')
as file:
                    line = file.readline().strip()
                    if not line:
                        raise ValueError("Пустой входной файл.")
                    try:
                        self.input_vector = [float(x) for x in
line.split()]
                    except Exception:
                        self.log_error(f"Некорректный формат входного
вектора: '{line}'")
                    except FileNotFoundError:
                        self.log_error(f"Файл {self.input_vector_file} не
найден.")
                    except Exception as e:
                        self.log_error(f"Произошла ошибка при загрузке входного
вектора: {e}")

        def activation_function(self, x):
            # Функция активации
            return 1 / (1 + math.exp(-x))

```



```

def forward_pass(self):
    if not self.layers or not self.input_vector:
        self.log_error("Отсутствуют веса или входной вектор.")
        return None

    vector = self.input_vector
    for layer_number, layer in enumerate(self.layers, start=1):
        new_vector = []
        for neuron_number, neuron_weights in enumerate(layer,
start=1):
            if len(neuron_weights) != len(vector):
                self.log_error(
                    f"Ошибка на слое {layer_number}, нейрон
{neuron_number}: "
                    f"несоответствие размерностей (вход:
{len(vector)}, веса: {len(neuron_weights)})."
                )
                return None
            weighted_sum = sum(w * x for w, x in
zip(neuron_weights, vector))

        new_vector.append(self.activation_function(weighted_sum))
        vector = new_vector
    return vector

def save_network_structure(self, output_file):
    try:
        with open(output_file, 'w', encoding='utf-8') as file:
            json.dump(self.network_structure, file,
ensure_ascii=False, indent=4)
            print(f"Структура сети успешно сохранена в {output_file}")
    except Exception as e:
        self.log_error(f"Ошибка при сохранении структуры сети:
{e}")

def save_result(self, output_file):
    result = self.forward_pass()
    if result is None:
        self.log_error("Ошибка: результат не был рассчитан.")
        return
    try:
        with open(output_file, 'w', encoding='utf-8') as file:
            json.dump(result, file, ensure_ascii=False, indent=4)
            print(f"Результат успешно сохранен в {output_file}")
    except Exception as e:
        self.log_error(f"Ошибка при сохранении результата: {e}")

```

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:

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

...

$[xk1, xk2, \dots, xkn] \rightarrow [yk1, yk2, \dots, ykm]$

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:

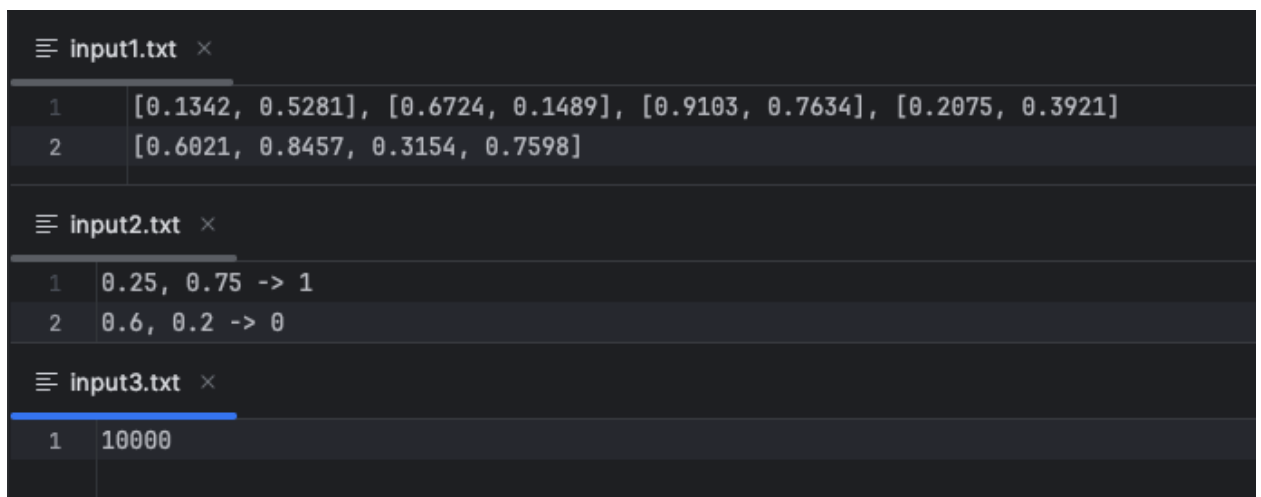
1 : Ошибка1

2 : Ошибка2

...

N : Ошибка N

Примеры



≡ input1.txt ×	
1	[0.1342, 0.5281], [0.6724, 0.1489], [0.9103, 0.7634], [0.2075, 0.3921]
2	[0.6021, 0.8457, 0.3154, 0.7598]

≡ input2.txt ×	
1	0.25, 0.75 -> 1
2	0.6, 0.2 -> 0

≡ input3.txt ×	
1	10000

Рис. 13 – Входные данные программы

```
python nntask5.py input1=input1.txt input2=input2.txt input3=input3.txt output1=output.txt
Входные файлы: input1.txt, input2.txt
Число итераций: input3.txt
Выходной файл: output.txt
История обучения успешно сохранена в output.txt
```

Рис. 14 – Запуск программы

```
output.txt x
1 0: 0.04745959533371627
2 1: 0.047446604329407935
3 2: 0.04742677643069837
4 3: 0.04740000898352725
5 4: 0.04736620464503413
6 5: 0.04732527172825042
7 6: 0.047277124542183566
8 7: 0.04722168372564803
9 8: 0.047158876573157815
10 9: 0.04708863735116345
11 10: 0.047010907602899
12 11: 0.0469256364400973
13 12: 0.04683278081983848
14 13: 0.046732305804817804
15 14: 0.04662418480535284
16 15: 0.046508399801499796
17 16: 0.04638494154371271
18 17: 0.046253809730557566
19 18: 0.046115013162087645
20 19: 0.045968569867592755
21 20: 0.045814507206557505
22 21: 0.04565286194179685
23 22: 0.045483680283884095
24 23: 0.04530701790614294
25 24: 0.045122939929642496
26 25: 0.0449315208778081
27 26: 0.04473284460044301
28 27: 0.044527004167141865
29 28: 0.04431410173026597
30 29: 0.0440942483578415
31 30: 0.04386756383693091
32 31: 0.043634176448215614
33 32: 0.043394222712709804
34 33: 0.04314784711170208
35 34: 0.04289520178118874
36 35: 0.04263644618222053
37 36: 0.04237174674873091
38 37: 0.0421012765145464
39 38: 0.04182521472139891
40 39: 0.04154374640986253
41 40: 0.041257061995224366
```

Рис. 15 – Результат работы программы

Листинг кода

```
import json
import numpy as np
import math
from helpers.file_handler import parse_args

def read_from_text_file(filename):
    try:
        with open(filename, 'r') as f:
            message = f.read().strip()
            return message
    except FileNotFoundError:
        print(f"Ошибка: файл {filename} не найден.")
        return None

class NeuralNetwork:
    def __init__(self, weights_file, dataset_file, iterations,
learning_rate=0.1):
        self.weights_file = weights_file
        self.dataset_file = dataset_file
        self.iterations = iterations
        self.learning_rate = learning_rate
        self.layers = []
        self.dataset = []
```

```

self.network_structure = {}
self.error_file = f"error_{weights_file}"
self.first_error_log = True
self.load_weights()
self.load_dataset()

def log_error(self, message):
    mode = 'w' if self.first_error_log else 'a'
    with open(self.error_file, mode, encoding='utf-8') as
error_file:
        error_file.write(message + "\n")
        self.first_error_log = False

def load_weights(self):
    try:
        with open(self.weights_file, 'r', encoding='utf-8') as
file:
            for layer_number, line in enumerate(file, start=1):
                line = line.strip()
                if not line:
                    continue
                try:
                    layer = json.loads(f"[{line}]")
                    self.layers.append(np.array(layer,
dtype=np.float64))
                    self.network_structure[f"Layer
{layer_number}"] = layer
                except Exception:
                    self.log_error(f"Строка {layer_number}:
некорректный формат матрицы весов '{line}'")
            except FileNotFoundError:
                self.log_error(f"Файл {self.weights_file} не найден.")
            except Exception as e:
                self.log_error(f"Произошла ошибка при загрузке весов:
{e}")

def load_dataset(self):
    try:
        with open(self.dataset_file, 'r', encoding='utf-8') as
file:
            for line_number, line in enumerate(file, start=1):
                line = line.strip()
                if not line:
                    continue
                try:
                    inputs, outputs = line.split("->")
                    x = [float(i) for i in
inputs.strip().split(",")]
                    y = [float(o) for o in
outputs.strip().split(",")]
                    self.dataset.append((np.array(x),
np.array(y)))
                except Exception:
                    self.log_error(f"Строка {line_number}:
некорректный формат строки обучающей выборки: '{line}'")
            except FileNotFoundError:
                self.log_error(f"Файл {self.dataset_file} не найден.")
            except Exception as e:

```

```

        self.log_error(f"Произошла ошибка при загрузке обучающей
выборки: {e}")

    def activation_function(self, x):
        return 1 / (1 + np.exp(-x)) # Сигмоид

    def activation_derivative(self, x):
        return x * (1 - x) # Производная сигмоида

    def forward_pass(self, inputs):
        activations = [inputs]
        for layer in self.layers:
            inputs = self.activation_function(np.dot(inputs, layer.T))
            activations.append(inputs)
        return activations

    def backpropagate(self, activations, expected_output):
        errors = [expected_output - activations[-1]] # Ошибка
        # в выходного слоя
        deltas = [errors[-1] *
self.activation_derivative(activations[-1])]

        # Обратное распространение ошибки
        for i in range(len(self.layers) - 1, 0, -1):
            error = np.dot(deltas[-1], self.layers[i]) # Ошибка
            # предыдущего слоя
            delta = error * self.activation_derivative(activations[i])
            errors.append(error)
            deltas.append(delta)

        errors.reverse()
        deltas.reverse()

        # Обновление весов
        for i in range(len(self.layers)):
            self.layers[i] += self.learning_rate * np.outer(deltas[i],
activations[i])

        return np.mean(np.abs(errors[0]))

    def train(self, history_file):
        history = []
        for iteration in range(1, self.iterations + 1):
            total_error = 0
            for inputs, expected_output in self.dataset:
                activations = self.forward_pass(inputs)
                error = self.backpropagate(activations,
expected_output)
                total_error += error

            average_error = total_error / len(self.dataset)
            history.append(f"{iteration - 1}: {average_error}")

        try:
            with open(history_file, 'w', encoding='utf-8') as file:
                file.write("\n".join(history))
            print(f"История обучения успешно сохранена в
{history_file}")

```

```
except Exception as e:
    self.log_error(f"Ошибка при сохранении истории обучения:
{e}")
```