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 \_\_\_\_\_\_\_\_\_\_\_\_\_ И.И. Слеповичев

подпись, дата

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), ..., (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.

**Способ проверки результата:**

1. 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>

1. Сообщение об ошибке с указанием номера строки с ошибкой во входном файле.

**Примеры**

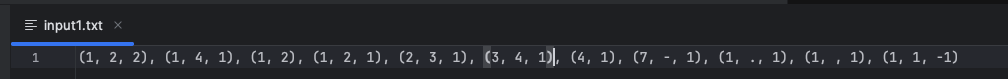
****

Рис. 1 – Входные данные программы

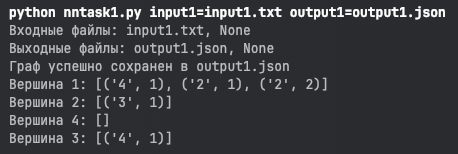


Рис. 2 – Запуск программы

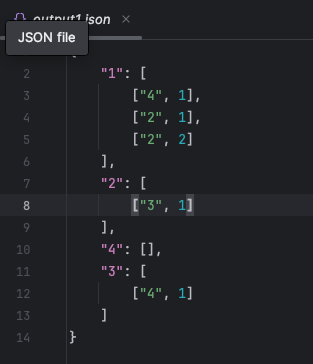


Рис. 3 – Результат работы программы

**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}")

1. **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:

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

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

**Примеры**

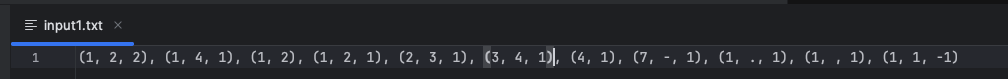
****

Рис. 4 – Входные данные программы

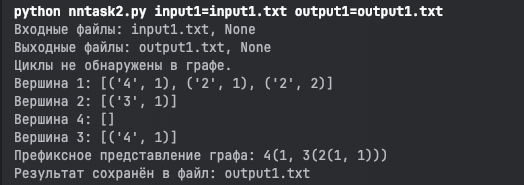
****

Рис. 5 – Запуск программы

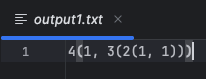


Рис. 6 – Результат работы программы

**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  
 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}")  
  
 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}")

1. **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:

**+** – cумма 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**

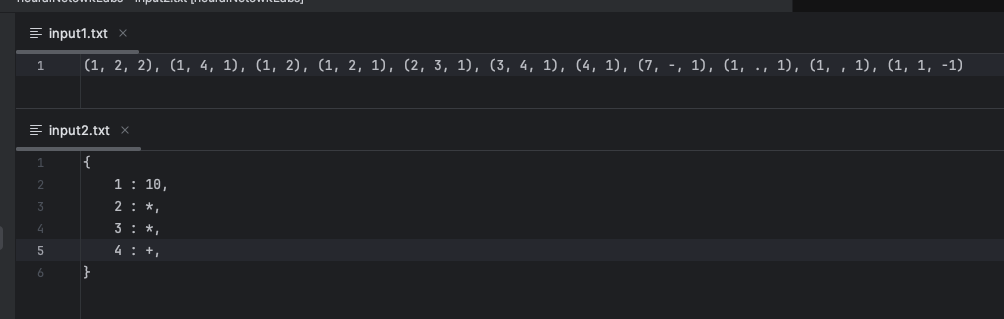
****

Рис. 7 – Входные данные программы

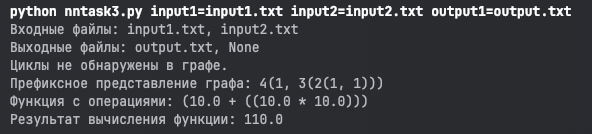
****

Рис. 8 – Запуск программы

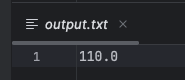


Рис. 9 – Результат работы программы

**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)

1. **Построение многослойной нейронной сети**

**На входе:**

а) 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]]

б) Текстовый файл с входным вектором в формате:

x1, x2, …, xn.

**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:

y1, y2, …, yn.

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

**Примеры**

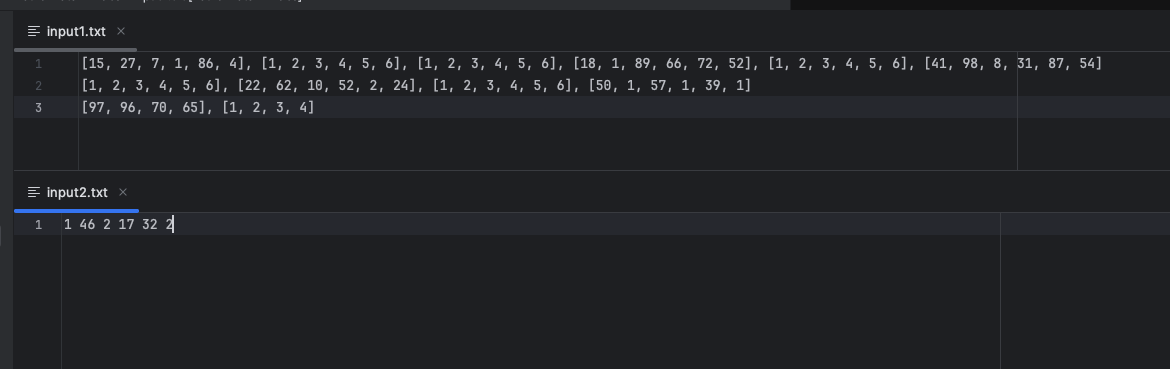
****

Рис. 10 – Входные данные программы

****

Рис. 11 – Запуск программы



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}")

1. **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

**Примеры**

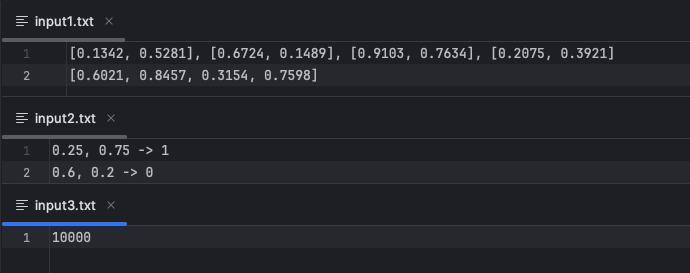
****

Рис. 13 – Входные данные программы

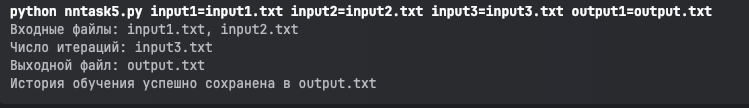
****

Рис. 14 – Запуск программы

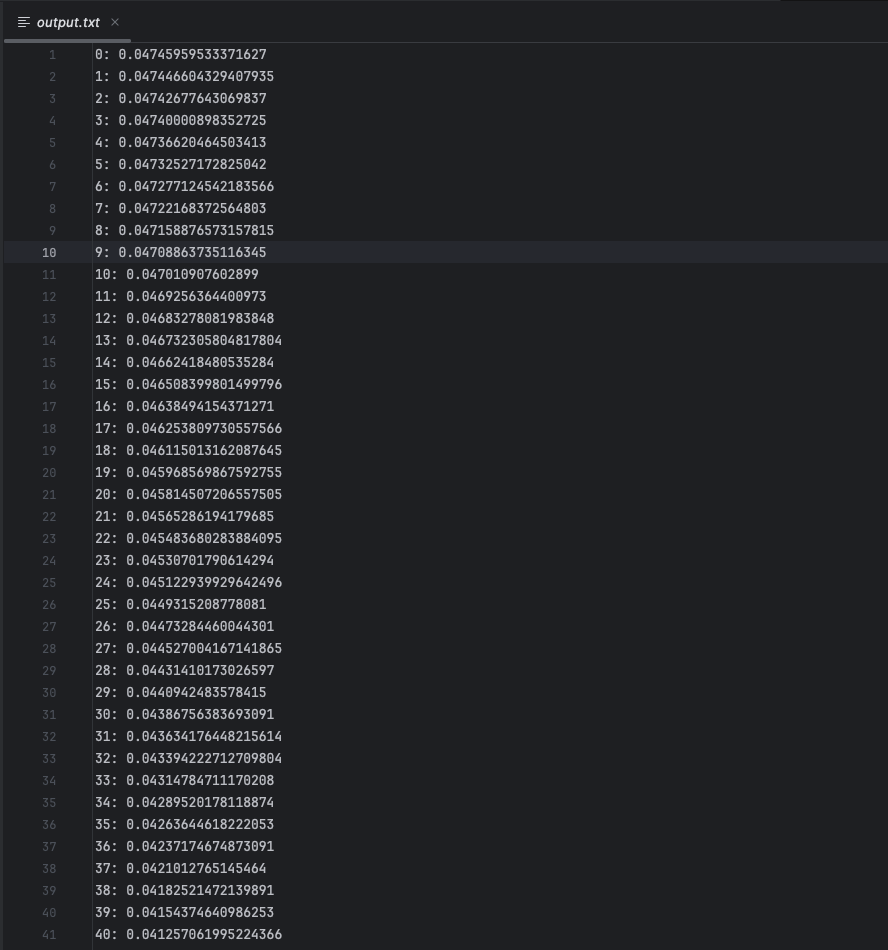


Рис. 15 – Результат работы программы

**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 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}")