МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Инженерная школа информационных технологий и робототехники
Направление подготовки 09.04.01 Информатика и вычислительная техника
Отделение Информационных технологий

Отчет по индивидуальному заданию
по дисциплине «Нейроэволюционные вычисления»

Тема работы
Реализация алгоритма SANE

Студент

Группа	**ФИО**	**Подпись**	**Дата**
8ВМ22	Ямкин Н.Н.		

Руководитель

Должность	**ФИО**	**Ученая степень, звание**	**Подпись**	**Дата**
Старший преподаватель ОИТ	Григорьев Д.С.			

Томск – 2023

Оглавление

Цель работы: реализовать нейроэволюционный алгоритм SANE.

1 Реализация алгоритма

Алгоритм SANE (Symbiotic Adaptive NeuroEvolution) является вариантом коэволюционного алгоритма для эволюции весов и структуры нейронной сети. В алгоритме используется нейронная сеть с одним скрытым слоем. В хромосоме кодируется список связей нейрона и веса связей. Алгоритм вводит понятие комбинации нейронной сети – это набор нейронов, представляющих одну нейронную сеть. В алгоритме используются две популяции: популяция нейронов и популяция комбинаций нейронов. Данные популяции эволюционируют независимо друг от друга, т. к. представляют разные сущности.

Ниже приведены основные шаги алгоритма в рамках одного поколения.

1. Сброс приспособленностей нейронов в популяции комбинаций.

2. Для каждой комбинации:

- Выбор соответствующих нейронов.
- Формирование ИНС.
- Оценка ИНС.
- Присваивание полученного значения приспособленности текущей комбинации.

3. Обновить приспособленность нейрона = средняя приспособленность 5 лучших ИНС, включающих этот нейрон.

4. Сортировка популяции нейронов по приспособленности.

5. Скрещивание.

- Скрещиваются только 25% лучших особей.
- После скрещивания остается только 1 случайный потомок.
- Вместо второго потомка используется один из родителей.
- Два потомка замещают худших особей.

6. Мутация.

- Проводится для всех нейронов.
- Вероятность мутации 0,1% па каждый бит для всех особей.

7. Скрещивание комбинаций.

- 1- точечный кроссинговер.
- Точки разрыва только межу указателями.
- Все остальное как в скрещивании нейронов.

8. Мутация комбинаций.

- Только для комбинаций-потомков.
- 1% вероятность перенаправить указатель на другой нейрон.
- 50% вероятность перенаправить указатель на нейрон-потомок.

В данной работе используется модифицированный алгоритм SANE: вес кодируется помощью вещественного кодирования.

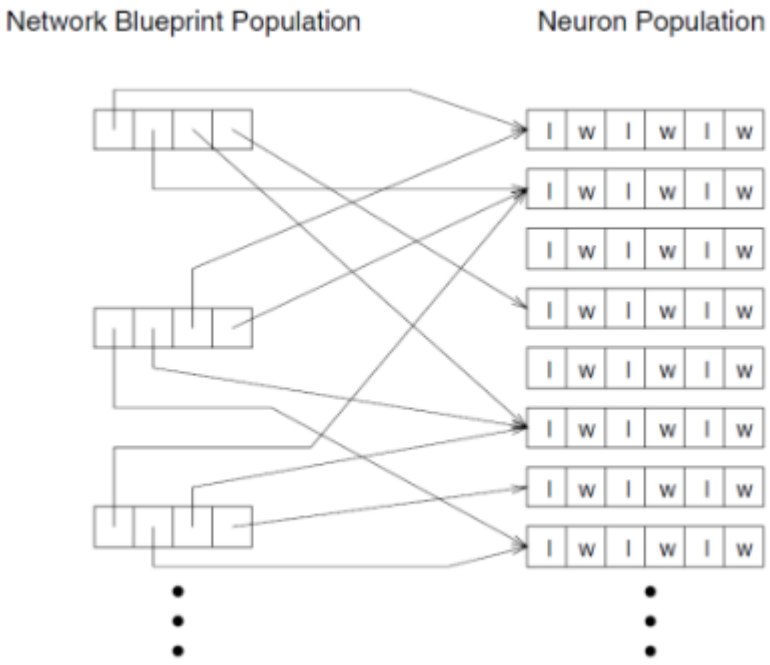

Рисунок 1 – Кодирование информации об ИНС в алгоритме SANE

Рассмотрим основные классы, участвующие в реализации алгоритма.

1.1 Класс RealGene

Класс RealGene реализует логику работы с одним геном, использующим вещественное кодирование. Имеются следующие поля:

1. value – вещественное значение гена;

В классе имеются следующие методы:

1. __init__ – конструктор с параметрами:
   - value – значение гена;

2. init – инициализация векторов входных и выходных весов случайными значениями;
   - min_value – минимальное случайно сгенерированное число;
   - max_value – максимальное случайно сгенерированное число;

3. mutation – мутация векторов весов. Используется нормальное распределение в окрестности значения гена.

4. crossover – скрещивание двух нейронов, используется скрещивание смешением (Blend crossover). В качестве результата возвращается два новых нейрона. Параметры:
   - parent1 – первый родитель;
   - parent2 – второй родитель;

Для скрещивания используется дополнительная функция crossover_real, принимающая в качестве параметров два вещественных значения родительских генов и возвращающая два вещественных значения дочерних генов.

1.2 Класс Gene

Класс Gene реализует логику работы с одним геном, хранящим вес и структуру нейрона.

Имеются следующие поля:

1. label – целочисленное значение, хранящее направление связи (входная или выходная) и индекс нейрона, с которым установлена связь;

2. weight – вес связи, объект класса RealGene.

В классе имеются следующие методы:

1. __init__ – конструктор, происходит инициализация полей нулевыми значениями;

2. get_connection_type – метод возвращает направление соединения, при этом, если поле label > 127, то нейрон выходной, иначе входной;

3. get_index – метод возвращает индекс нейрона. Индекс вычисляется по следующей формуле label mod N, где N – общее количество нейронов в скрытом слое;

4. get_weight – метод возвращает вес связи;

5. init – инициализация полей случайными значениями. Параметры:

- min_value – минимальное случайно сгенерированное число;
- max_value – максимальное случайно сгенерированное число;

6. mutation – мутация весов и меток; для метки используется инвертирование бита; для веса вызывается метод mutation класса RealGene;

7. crossover – скрещивание двух генов; для метки используется одноточечный кроссинговер; для веса вызывается метод crossover класса RealGene. Параметры:

- parent1 – первый родитель;
- parent2 – второй родитель;

Исходный код классов RealGene и Gene представлены в листинге 1.

Листинг 1. Исходный код классов RealGene и Gene.

```
from enum import Enum
import random
class ConnectionType(Enum):
    INPUT = 1
    OUTPUT = 2
    @classmethod
    def from_label(cls, label: int):
        return cls(cls.OUTPUT if label > (2**7 - 1) else cls.INPUT)

def crossover_integer(parent1: int, parent2: int, precision: int):
    crossover_point = random.randrange(precision)
    mask1 = ((2 ** precision - 1) << crossover_point) & (2 ** precision -
1)
    mask2 = ((2 ** precision - 1) >> (precision - crossover_point)) & (2
** precision - 1)
    child1 = (parent1 & mask1) | (parent2 & mask2)
    child2 = (parent2 & mask1) | (parent1 & mask2)
    return child1, child2

# СКРЕЩИВАНИЕ ДВУХ ГЕНОВ
# принимает два вещественных значения родительских генов и возвращает два
# вещественных значения дочерних генов

def crossover_real(parent1: float, parent2: float, blend=0.1):
    child1 = parent1 - blend * (parent2 - parent1)
    child2 = parent2 + blend * (parent2 - parent1)
    return child1, child2

def invert_bit(value: int, bit: int, precision: int):
    mask = (1 << bit) & (2 ** precision - 1)
    return value ^ mask

# Класс RealGene реализует логику работы с одним геном, использующим
# вещественное кодирование
class RealGene(object):
    def __init__(self, value: float, min_value: float, max_value: float):
        self.value = value          # вещественное значение гена
        self.min_value = min_value  # минимальное случайно сген. число
        self.max_value = max_value  # максимальное случайно сген. число

# инициализация векторов входных и выходных весов случайными значениями
    def init(self):
# случайное значение, полученное из равномерного распределения в диапазоне
от min_value до max_value
        self.value = random.uniform(self.min_value, self.max_value)

# мутация векторов весов, используется распределение Гаусса в окрестности
значения гена
```

```
    def mutation(self):
        self.value = random.gauss(mu=self.value, sigma=0.1)   # случайное
значение, полученное из нормального распределения с заданным средним и
стандартным отклонением

# СКРЕЩИВАНИЕ ДВУХ НЕЙРОНОВ, используется скрещивание смешением (blend
crossover); в качестве результата возвращаются два новых нейрона

# метод не имеет доступа к экземпляру класса или к атрибутам экземпляра

    @staticmethod
    def crossover(parent1, parent2):
# parent1 и parent2 - экзепляры класса RealGene
        min_value = parent1.min_value
        max_value = parent1.max_value
        child1_gene, child2_gene = crossover_real(    # скрещивание двух
генов
            parent1=parent1.value,
            parent2=parent2.value)
        child1 = RealGene(          # первый нейрон потомок
            value=child1_gene,
            min_value=min_value,
            max_value=max_value)
        child2 = RealGene(          # второй нейрон потомок
            value=child2_gene,
            min_value=min_value,
            max_value=max_value)
        return child1, child2


# Класс Gene реализует логику работы с одним геном, хранящим вес и
структуру нейрона
class Gene(object):
    def __init__(self, min_value: float, max_value: float):
        self.label = 0    # целочисленное значение, хранящее направление
связи (входная или выходная) и индекс нейрона, с которым установлена связь
        self.weight = RealGene(   # вес связи, объект класса RealGene
            value=0.0,
            min_value=min_value,
            max_value=max_value)

    def get_connection_type(self) -> ConnectionType:    # метод,
возвращает направление соединения (label > 127 - нейрон выходной, иначе
входной)
        return ConnectionType.from_label(self.label)

# метод возвращает индекс нейрона
    def get_index(self, neurons_count) -> int:
        return self.label % neurons_count
```

```
# метод возвращает вес связи
    def get_weight(self) -> float:
        return self.weight.value

    def init(self):
        self.label = int(random.random() * (2**8 - 1))  # инициализация
метки случайным числом от 0 до 255
        self.weight.init()     # инициализация веса связи

# мутация весов и меток; для метки используется инвертирование бита; для
веса вызывается метод mutation класса RealGene
    def mutation(self):
      # мутация произойдет с вероятностью 0.01 (1 %)
        if random.random() <= 0.01:
# генерация случайного числа от 0 до 8. Получаем номер бита, который будем
инвертировать
            mutation_bit = random.randrange(8)
# инвертируем полученный ранее номер бита метки
            self.label = invert_bit(self.label, mutation_bit, 8)
            self.weight.mutation()    # мутация веса связи

# скрещивание двух генов; для метки используется одноточечный
кроссинговер; для веса вызывается метод crossover класса RealGene
    @staticmethod
    def crossover(parent1, parent2):   # parent1 и parent2 - экзепляры
класса Gene
        min_value = parent1.weight.min_value
        max_value = parent1.weight.max_value
        child1 = Gene(                    # потомок 1
            min_value=min_value,
            max_value=max_value)
        child2 = Gene(                    # потомок 2
            min_value=min_value,
            max_value=max_value)
        # скрещивание метки
        child1.label, child2.label = crossover_integer(
            parent1=parent1.label,
            parent2=parent2.label,
            precision=8)
        # скрещивание весов
        child1.weight, child2.weight = RealGene.crossover(
            parent1=parent1.weight,
            parent2=parent2.weight)
        return child1, child2
```

1.3 Класс Neuron

Класс Neuron реализует логику работы с нейроном. Имеются следующие поля:

1. genes – массив, хранящий гены (объекты класса Gene);
2. connections_count – количество соединений в одном нейроне (в данном алгоритме подразумевается, что сеть может быть не полносвязной);
3. fitness – приспособленность нейрона.

В классе имеются следующие методы:

1. __init__ – конструктор. Параметры:
   - connections_count – количество соединений в одном нейроне;

2. init – инициализация нейрона. Инициализация происходит в цикле, в котором создаются гены со случайными значениями полей и в конце итерации проверяется, что гены имеют разные направления соединений. Это необходимо для того, чтобы не получился нейрон, имеющий связи одного направления, поскольку такая конфигурация нейрона не будет иметь смысла. Параметры:
   - min_value – минимальное случайно сгенерированное число;
   - max_value – максимальное случайно сгенерированное число;

3. get_weight – получение вектора весов из генов соответствующего направления. Параметры:
   - neurons_count – количество нейронов в скрытом слое (необходимо для вычисления индекса нейрона входного, либо выходного слоев);
   - connection – направление соединения;

4. get_input_weights – получение вектора весов входных соединений скрытого слоя. Параметры:
   - neurons_count – количество нейронов в скрытом слое

5. get_output_weights - получение вектора весов выходных соединений скрытого слоя. Параметры:

- neurons_count – количество нейронов в скрытом слое;

6. mutation – мутация генов. Вызывается метод mutation класса Gene для каждого гена.

7. crossover - скрещивание двух нейронов. Вызывается метод crossover класса Gene для каждого гена. Параметры

- parent1 – первый родитель;
- parent2 – второй родитель.

Исходный код класса представлен в листинге 2.

Листинг 2. Исходный код класса Neuron.

```
import numpy as np

# класс Neuron реализует логику работы с нейроном
class Neuron(object):
    def __init__(self,
                 connections_count: int):
        self.genes = [] # список, хранящий гены (объекты класса Gene)
        self.connections_count = connections_count    # количество
соединиений в одном нейроне (подразумевается, что сеть может быть не
полносвязной)
        self.fitness = 0.0 # приспособленность нейрона


    # инициализация нейрона. Инициализация происходит в цикле, в котором
создаются гены со случайными значениями полей и в конце итерации
проверяется, что гены имеют разные направления соединений.
    # Это необходимо для того, чтобы не получился нейрон, имеющий связи
одного направления, поскольку такая конфигурация нейрона не будет иметь
смысла
    def init(self,
             min_value: float,
             max_value: float):
        while True:
            for i in range(self.connections_count):    # количество генов
в хромосоме также равно connections_count
                self.genes.append(Gene(    # добавляем гены в хромосому
                    min_value=min_value,
                    max_value=max_value))
            for i in range(self.connections_count):
                self.genes[i].init()    # инициализируем метку и вес связи
            connection_types = [connection.get_connection_type().value for
connection in self.genes]
            if len(set(connection_types)) > 1:
                break
            self.genes.clear()


    # получение вектора весов из генов соответствующего направления
    def get_weights(self,
# количество нейронов в скрытом слое (необходимо для вычисления индекса
нейрона входного либо выходного слоев)
                    neurons_count: int,
# напрадение соединения
                    connection: ConnectionType) -> np.array:
        result = np.zeros(neurons_count)
        for gene in self.genes:
            if gene.get_connection_type() == connection:
                result[gene.get_index(neurons_count)] = gene.get_weight()
        return result
```

```
    # получение вектора весов входных соединений скрытого слоя
    def get_input_weights(self,
# количество нейронов в скрытом слое
                          neurons_count: int) -> np.array:
        return self.get_weights(
            neurons_count=neurons_count,
            connection=ConnectionType.INPUT)

    # получение вектора весов выходных соединений скрытого слоя
    def get_output_weights(self,
# количество нейронов в скрытом слое
                           neurons_count: int) -> np.array:
        return self.get_weights(
            neurons_count=neurons_count,
            connection=ConnectionType.OUTPUT)

# мутация генов. Вызывается метод mutation класса Gene для каждого гена
    def mutation(self):
        for i in range(len(self.genes)):
            self.genes[i].mutation()

# скрещивание двух нейронов. Вызывается метод mutation класса Gene для
каждого гена
    @staticmethod
    def crossover(parent1, parent2):
        genes_count = len(parent1.genes)
        connections_count = parent1.connections_count
        child1 = Neuron(
            connections_count=connections_count)
        child2 = Neuron(
            connections_count=connections_count)
        for i in range(genes_count):
            child1_gene, child2_gene = Gene.crossover(
                parent1=parent1.genes[i],
                parent2=parent2.genes[i])
            child1.genes.append(child1_gene)
            child2.genes.append(child2_gene)
        return child1, child2
```

1.4 Класс NeuronPopulation

Класс NeuronPopulation - реализует логику работы с популяцией нейронов, предоставляя высокоуровневый интерфейс над массивом нейронов.

Имеются следующие поля:

1. neurons – массив нейронов;

В классе имеются следующие методы:

1. __init__ – конструктор. В конструкторе происходит создание нейронов. Параметры:
   - min_value – минимальное случайно сгенерированное число;
   - max_value – максимальное случайно сгенерированное число;
2. init – инициализация популяции нейронов. Для каждого нейрона вызывается метод init, в который передаются параметры метода. Параметры:
   - min_value – минимальное случайно сгенерированное число;
   - max_value – максимальное случайно сгенерированное число;
3. crossover – скрещивание верхней четверти лучших нейронов;
4. mutation - мутация всех нейронов;
5. __ getitem__ – встроенный метод языка Python, позволяющий обращаться к данному объекту как к коллекции. При этом возвращается соответствующий индексу нейрон.
6. __len__ – встроенный метод языка Python, позволяющий получить размер коллекции. При этом возвращается размер массива нейронов.

Исходный код класса представлен в листинге 3.

Листинг 3. Исходный код класса NeuronPopulation.

```
# Класс NeuronPopulation реализует логику работы с популяцией нейронов,
предоставляя высокоуровневый интерфейс над массивом нейронов
class NeuronPopulation(object):
    def __init__(self,
                 population_size: int,
                 connections_count: int):
        self.neurons = []                    # Массив нейронов
        for i in range(population_size):     # Создание популяции нейронов
            self.neurons.append(Neuron(
                connections_count=connections_count))

    def init(self,
             min_value: float,
             max_value: float):
        for neuron in self.neurons:   # инициализация нейронов в
популяции. Для каждого нейрона вызывается метод init, в который передаются
параметры входа
            neuron.init(
                min_value=min_value,
                max_value=max_value)

    # скрещивание верхней четверти лучших нейронов. Потомки при этом
заменяют худшие особи
    def crossover(self):
# сортировка массива нейронов по функции приспособленности
        self.neurons.sort(key=lambda x: x.fitness)
# выбираем из массива нейронов верхнюю четверть нейронов с лучшими
значениями функции приспособленности
        selected_neuron_count = int(len(self.neurons) / 4)
# оставляем четное количество нейронов в списке
        selected_neuron_count -= selected_neuron_count % 2
        for i in range(0, selected_neuron_count, 2):
            parent1 = self.neurons[i]
            parent2 = self.neurons[i + 1]
            child1, child2 = Neuron.crossover(
                parent1=parent1,
                parent2=parent2)
            selected1 = parent1 if random.randrange(2) == 0 else
parent2    # после скрещивания остается только 1 случайный потомок
            selected2 = child1 if random.randrange(2) == 0 else
child2      # вместо второго потомка используется один из родителей
            self.neurons[-selected_neuron_count + i] =
selected1              # обновляем нейроны в списке
            self.neurons[-selected_neuron_count + i + 1] =
selected2         # обновляем нейроны в списке
```

```
    def mutation(self):
        for i in range(len(self.neurons)):
            self.neurons[i].mutation()        # мутация нейронов

    def __getitem__(self, key: int) -> Neuron: # встроенный метод языка
Python, позволяющий обращаться к данному объекту как к коллекции.
        return self.neurons[key]        # при этом возвращается
соответствующий индексу нейрон

    def __len__(self):                # встроенный метод языка Python,
позволяющий получить размер коллекции.
        return len(self.neurons)      # при этом возвращается размер
массива нейронов
```

1.5 Класс Blueprint

Класс Blueprint - реализует логику работы с комбинацией нейронов. Комбинация рассматривается как отдельная особь. Массив индексов нейронов является хромосомой. Имеются следующие поля:

1. neurons – массив индексов нейронов;

2. neuron_population – популяция нейронов (указатель на популяцию нейронов, поскольку данный объект не должен владеть популяцией);

3. fitness – значение приспособленности данной комбинации.

В классе имеются следующие методы:

1. _init_ – конструктор. В конструкторе происходит создание нейронов. Параметры:

- neurons – массив индексов нейронов;
- neuron_population – популяция нейронов.

2. mutation – мутация комбинации. В результате мутации происходит замена выбранного нейрона в текущей комбинации на случайно выбранный нейрон из популяции нейронов.

3. crossover - скрещивание. Скрещиваются массивы индексов нейронов. Применяется одноточечный кроссинговер.

Исходный код класса представлен в листинге 4.

Листинг 4. Исходный код класса Blueprint

```
from typing import List
# класс Blueprint - реализует логику работы с комбинацией нейронов.
Комбинация рассматривается как отдельная особь
# хромосома - массив индексов нейронов
class Blueprint(object):
    def __init__(self,
                 neurons: List[int],  # массив индексов нейронов
(комбинация нейронов)
                 neuron_population: NeuronPopulation):
        self.neurons = neurons              # массив индексов нейронов
        # популяция нейронов (указатель на популяцию нейронов)
        self.neuron_population = neuron_population
        # значение приспособленности данной комбинации
        self.fitness = 0.0

 # мутация комбинации. В результате мутации происходит замена случайно
выбранного нейрона в текущей комбинации на случайно выбранный нейрон из
популяции нейронов
    def mutation(self):
        if random.random() <= 0.01:
            new_neuron_index =
random.randrange(len(self.neuron_population))  # случайно выбранный нейрон
в популяции нейронов
            neuron_index = random.randrange(len(self.neurons))     #
случайно выбранный нейрон в текущей комбинации
            self.neurons[neuron_index] = new_neuron_index          # замена
нейронов

    # скрещивание. Скрещиваются массивы индексов нейронов. Применяется
одноточечный кроссинговер
    @staticmethod
    def crossover(parent1, parent2):
        neurons_count = len(parent1.neurons)              # длина
комбинации нейронов
        crossover_point = random.randrange(neurons_count) # случайная
точка для скрещивания комбинации
        child1_neurons = parent1.neurons[:crossover_point] +
parent2.neurons[crossover_point:]  # скрещивание комбинаций нейронов
        child2_neurons = parent2.neurons[:crossover_point] +
parent1.neurons[crossover_point:]
        return Blueprint(child1_neurons, None), Blueprint(child2_neurons,
None)
```

1.6 Класс BlueprintPopulation

Класс BlueprintPopulation – реализует логику работы с популяцией комбинаций нейронов, предоставляя высокоуровневый интерфейс над массивом комбинаций нейронов. Имеются следующие поля:

1. population_size – размер популяции (количество комбинаций в популяции);

2. blueprint_size – размер комбинации нейронов (количество нейронов в комбинации). Должен быть равен количеству нейронов в скрытом слое.

3. neuron_population – популяция нейронов;

4. blueprints – массив комбинаций нейронов.

В классе имеются следующие методы:

1. _init_ – конструктор. Параметры:

- population_size – размер популяции;
- blueprint_size – размер комбинации нейронов.

2. init – инициализация популяции комбинаций нейронов. Происходит создание комбинаций нейронов. Параметры:

- neuron_population – популяция нейронов;

3. select_neurons – внутренний метод, возвращающий список индексов случайно выбранных нейронов из популяции нейронов. При этом индексы являются уникальными.

4. crossover – скрещивание верхней четверти лучших комбинации нейронов. Для скрещивания вызывается метод crossover класса Blueprint. Потомки при этом заменяют худшие особи.

5. mutation – мутация всех комбинаций нейронов.

6. __getitem__ – встроенный метод языка Python, позволяющий обращаться к данному объекту как к коллекции. При этом возвращается соответствующий индексу нейрон.

7. __len__ – встроенный метод языка Python, позволяющий получить размер коллекции. При этом возвращается размер массива нейронов

8. __iter__ – встроенный метод языка Python, позволяющий итерироваться по объекту. Итерация происходит по массиву комбинаций нейронов.

Исходный код класса представлен в листинге 5.

Листинг 5. Исходный код класса BlueprintPopulation

```
# Класс BlueprintPopulation - реализует логику работы с популяцией
комбинаций нейронов, предоставляя высокоуровневый интерфейс над массивом
комбинаций нейронов.
class BlueprintPopulation(object):
    def __init__(self,
                 population_size:
                 int, blueprint_size: int):
        self.population_size = population_size  # размер популяции
(количество комбинаций в популяции)
        self.blueprint_size = blueprint_size    # размер комбинации
нейронов (количество нейронов в комбинации). Должен быть равен количеству
нейронов в скрытом слое
        self.neuron_population = None           # популяция нейронов
        self.blueprints = []                    # массив комбинаций
нейронов

    # инициализация популяции комбинаций нейронов. Происходит создание
комбинаций нейронов
    def init(self,
             neuron_population: NeuronPopulation):
        self.neuron_population = neuron_population   # создание объекта
популяции нейронов
        for _ in range(self.population_size):
            selected_neurons = self.select_neurons() # отбор случайных
нейронов из популяции нейронов
            self.blueprints.append(Blueprint(        # создание массива
популяции комбинаций нейронов. Элементы массива объекты класса Blueprint
                neurons=selected_neurons,            # массив индексов
выбранных нейронов
                neuron_population=self.neuron_population))  # популяция
нейронов

    # внутренний метод, возвращающий список индексов случайно выбранных
нейронов из популяции нейронов. При этом индексы являются уникальными
    def select_neurons(self) -> List[int]:
        result = []
        while True:                                  # формирование списка
индексов
            for _ in range(self.blueprint_size):     # формирование одной
комбинации нейронов
                result.append(random.randrange(len(self.neuron_population)
))
            if len(set(result)) == self.blueprint_size:
                break
            result.clear()
        return result
```

```
    # мутация всех комбинаций нейронов
    def mutation(self):
        for blueprint in self.blueprints:
            blueprint.mutation()

    # скрещивание верхней четверти лучших комбинаций нейронов. Для
скрещивания вызывается метод crossover класса Blueprint. Потомки при этом
заменяют худшие особи
    def crossover(self):
# сортировка массива комбинаций нейронов по функции приспособленности
        self.blueprints.sort(key=lambda x: x.fitness)
# выбираем из массива нейронов верхнюю четверть нейронов с лучшими
значениями функции приспособленности
        selected_blueprint_count = int(len(self.blueprints) / 4)
# оставляем четное количество нейронов в списке
        selected_blueprint_count -= selected_blueprint_count % 2
        for i in range(0, selected_blueprint_count, 2):
            parent1 = self.blueprints[i]
            parent2 = self.blueprints[i + 1]
            child1, child2 = Blueprint.crossover(
                parent1=parent1,
                parent2=parent2)
            child1.neuron_population = self.neuron_population
            child2.neuron_population = self.neuron_population
            selected1 = parent1 if random.randrange(2) == 0 else
parent2  # после скрещивания остается только 1 случайный потомок
            selected2 = child1 if random.randrange(2) == 0 else
child2    # вместо второго потомка используется один из родителей
            self.blueprints[-selected_blueprint_count + i] =
selected1    # обновляем комбинации в списке
            self.blueprints[-selected_blueprint_count + i + 1] = selected2
# обновляем комбинации в списке
# встроенный метод языка Python, позволяющий обращаться к данному объекту
как к коллекции. При этом возвращается соответсвующий индексу нейрон
    def __getitem__(self, key: int) -> Blueprint:
        return self.blueprints[key]
    # встроенный метод языка Python, позволяющий получить размер
коллекции. При это возвращается размер массива нейронов
    def __len__(self):
        return len(self.blueprints)
    # встроенный метод языка Python, позволяющий итерироваться по объекту.
Итерация происходит по массиву комбинаций нейронов
    def __iter__(self):
        return iter(self.blueprints)
```

1.7 Классы функций активации

Базовым классом для всех функций активации является класс AbstractActivationFunction, представляющий интерфейс для работы с функциями активации. В данном классе имеется метод forward, который принимает на вход numpy-массив (вектор), применяет к каждому элементу функцию активации и возвращает numpy-массив. Дочерние классы должны реализовать данный метод.

Имеется реализация нескольких функций активации. Исходный код классов представлен в листинге 6.

Листинг 6. Исходный код классов функций активации.

```
class AbstractActivationFunction(object):
    def __init__(self):
        pass

    def forward(self, input_data: np.array) -> np.array:
        raise NotImplementedError()


class Sigmoid(AbstractActivationFunction):
    def __init__(self):
        pass

    def forward(self, input_data: np.array) -> np.array:
        return 1.0 / (1.0 + np.exp(-input_data))


class ReLU(AbstractActivationFunction):
    def __init__(self):
        pass

    def forward(self, input_data: np.array) -> np.array:
        return np.maximum(0.0, input_data)


class Tanh(AbstractActivationFunction):
    def __init__(self):
        pass

    def forward(self, input_data: np.array) -> np.array:
        return np.tanh(input_data)
```

1.8 Класс Layer

Класс Layer реализует логику работы со слоем нейронной сети. Имеются следующие поля:

1. weights – матрица весов слоя;

2. activation – функция активации (объект класса AbstractActivationFunction);

В классе имеются следующие методы:

1. _init_ – конструктор. Параметры:

- weights – матрица весов;
- activation – функция активации.

2. forward – прямой проход по слою. Матрица весов умножатся на вектор-столбец входных данных, к каждому элементу полученного вектор-столбца применяется функция активации. Параметры:

- input_data – вектор входных данных.

Исходный код класса представлен в листинге 7.

Листинг 7. Исходный код класса Layer.

```
# Класс Layer реализует логику работы со слоем нейронной сети
class Layer(object):
    def __init__(self,
                 weights: np.array,
                 activation: AbstractActivationFunction):
        self.weights = weights          # вектор весов
        self.activation = activation    # функция активации

# прямой проход по слою. Вектор весов умножается на вектор-столбец входных данных, к каждому элементу полученного вектора-столбца применяется функция активации
    def forward(self,
                input_data: np.array) -> np.array: # вектор входных данных
        return self.activation.forward(
            input_data=np.dot(self.weights, input_data))
```

1.9 Класс NeuralNetwork

Класс NeuralNetwork реализует логику работы с нейронной сетью. Имеются следующие поля:

1. fitness – приспособленность нейронной сети;

2. input_weights – матрица входных весов скрытого слоя;

3. output_weights – матрица выходных весов скрытого слоя (входных весов выходного слоя);

4. layers – список, содержащий слои нейронной сети (объекты класса Layer).

В классе имеются следующие методы:

1. _init_ – конструктор. Происходит инициализация весов нейронной сети и создание скрытого и выходного слоев сети. Параметры:

- hidden_neurons – массив индексов нейронов, из которых будет построен скрытый слой;
- inputs_counts – количество входов нейронной сети;
- outputs_counts – количество выходов нейронной сети;
- neuron_population – популяция нейронов, из которой будет происходить выборка нейронов.

2. forward – прямой проход по сети. Последовательно выполняется прямой проход по слоям сети с помощью метода forward класса Layer. Параметры:

- input_data – вектор входных данных.

Исходный код класса представлен в листинге 8.

Листинг 8. Исходный код класса NeuralNetwork.

```
class NeuralNetwork(object):
  # происходит инициализация весов нейронной сети и создание скрытого и
выходного слоёв сети
    def __init__(self,
                 hidden_neurons: List[int],  # массив индексов нейронов,
из которых будет построен скрытый слой
                 inputs_count: int,    # количество входов нейронной сети
                 outputs_count: int,   # количество выходов нейронной сети
                 neuron_population: NeuronPopulation):  # популяция
нейронов, из которой будет происходить выборка нейронов
        self.fitness = 0.0      # приспособленность нейронной сети
        self.input_weights = np.zeros((len(hidden_neurons), inputs_count))
# матрица входных весов скрытого слоя
        output_weights = np.zeros((len(hidden_neurons),
outputs_count))    # матрица выходных весов скрытого слоя (входных весов
выходного слоя)
        for i in range(len(hidden_neurons)):
            self.input_weights[i] =
neuron_population[hidden_neurons[i]].get_input_weights(inputs_count)
            output_weights[i] =
neuron_population[hidden_neurons[i]].get_output_weights(outputs_count)
        self.output_weights = np.zeros((outputs_count,
len(hidden_neurons)))
        for i in range(outputs_count):
            self.output_weights[i] = output_weights[:, i]

        self.layers = []                    # список, содержащий слои
нейронной сети (объекты класса Layer)
        self.layers.append(Layer(           # создание скрытого слоя сети
            weights=self.input_weights,
            activation=Sigmoid()))
        self.layers.append(Layer(           # создание выходного слоя сети
            weights=self.output_weights,
            activation=Sigmoid()))

 # прямой проход по сети. Последовательно выполняется прямой проход по
слоям сети с помощью метода forward класса Layer
    def forward(self,
                input_data: np.array) -> np.array: # вектор входных данных
        output = input_data
        for layer in self.layers:
            output = layer.forward(output)
        return output
```

1.10 Класс SANEAlgorithm

Класс SANEAlgorithm реализует логику работы с алгоритмом SANE. Имеются следующие поля:

1. neuron_population – популяция нейронов;

2. blueprint_population – популяция комбинаций нейронов;

3. best_nn – лучшая нейронная сеть.

В классе имеются следующие молоды:

1. _init_ – конструктор. Происходит создание популяции нейронов и популяции комбинаций нейронов. Параметры:

- blueprints_population_size – размер популяции комбинаций нейронов;
- neuron_population_size – размер популяции нейронов;
- hidden_layer_size – количество нейронов в скрытом слое;
- connections_count – количество соединений для одного нейрона.

2. init – инициализация популяции нейронов и популяции комбинаций нейронов. Параметры:

- min_value – минимальное случайно сгенерированное число;
- max_value – максимальное случайно сгенерированное число.

3. train – тренировка нейронных сетей. Параметры:

- generations_count – количество поколений;
- x_train – массив входных данных;
- y_train – массив выходных данных.

4. test – тестирование сети. Возвращается массив среднеквадратичных ошибок в соответствии с записями во входном наборе данных. Параметры:

- x_train – массив входных данных;
- y_train – массив выходных данных.

5. forward – внутренний метод, предназначенный для прохода по лучшей нейронной сети с одним набором данных. Возвращает среднеквадратичную ошибку. Параметры:

- x_train – вектор входных данных;
- y_train – вектор выходных данных.

6. forward_train – внутренний метод предназначенный для прохода всех комбинаций нейронных сетей по всем входным данным. По результатам прохода берётся среднее значение среднеквадратичных ошибок, полученных в результате прохода всех данных. Полученное значение является приспособленностью для нейронной сети и для комбинации нейронной сети, Параметры:

- neural_networks – массив нейронных сетей;
- x_train – массив входных данных;
- y_train – массив выходных данных.

7. create_neural_networks – внутренний метод, создающий массив нейронных сетей из комбинаций нейронных сетей. Параметры:

- inputs_counts – количество входов нейронной сети;
- outputs_counts – количество выходов нейронной сети.

8. update_neuron_fitness – внутренний метод, обновляющий приспособленность нейронов, являющуюся средним значением приспособленности 5 лучших нейронных сетей, включающих данный нейрон

Рассмотрим подробнее работу алгоритма (метод train). Сначала запускается цикл по количеству поколений. В цикле сначала создаются нейронные сети с помощью метода create_neural_networks. Затем, через созданные сети пропускается весь тестовый датасет. Для этого используется метод forward_train. После пропускания данных у нейронных сетей появится ненулевое значение приспособленности, и нейронные сети сортируются по ней по возрастанию. Таким образом в начале массива находятся лучшие нейронные сети. Далее происходит скрещивание и мутация популяции

нейронов, и затем скрещивание и мутация популяции комбинаций нейронов. Данные шаги повторяются заданное количество раз.

Исходный код класса представлен в листинге 9.

Листинг 9. Исходный код класса SANEAlgorithm.

```
import platform
import os
from copy import deepcopy

def mse(y_true: np.array, y_pred: np.array) -> float:
    return np.square(y_true - y_pred).mean()

CRLF = '\r\x1B[K' if platform.system() != 'Windows' else '\r'

# Класс SANEAlgorithm реализует логику работы с алгоритмом SANE
class SANEAlgorithm(object):
    def __init__(self,
                 blueprints_population_size: int,    # размер популяции
комбинации нейронов
                 neuron_population_size: int,        # размер популяции
нейронов
                 hidden_layer_size: int,             # количество нейронов
скрытого слоя
                 connections_count: int):            # количество
соединений для одного нейрона
        self.neuron_population = NeuronPopulation(  # популяция нейронов
            population_size=neuron_population_size,
            connections_count=connections_count)
        self.blueprint_population = BlueprintPopulation( # популяция
комбинаций нейронов
            population_size=blueprints_population_size,
            blueprint_size=hidden_layer_size)
        self.best_nn = None                          # лучшая нейронная сеть

        self.hidden_layer_size = hidden_layer_size

    # инициализация популяции нейронов и популяции комбинаций нейронов
    def init(self,
             min_value: float,
             max_value: float):
        self.neuron_population.init(  # инициализация популяции нейронов
            min_value=min_value,
            max_value=max_value)
        self.blueprint_population.init(  # инициализация популяции
комбинаций нейронов
            neuron_population=self.neuron_population)

    # тренировка нейронных сетей
    def train(self,
              generations_count: int,  # количество поколений
              x_train: np.array,       # массив входных данных
              y_train: np.array):      # массив выходных данных
        if x_train.shape[0] != y_train.shape[0]:
```

```
            raise Exception()
        result = []
        for generation in range(generations_count):
            inputs_count = x_train[0].size
            outputs_count = y_train[0].size

            neural_networks = self.create_neural_networks(    # создание
нейронных сетей
                inputs_count=inputs_count,                    # количество
входов нейронной сети
                outputs_count=outputs_count)                  # количество
выходов нейронной сети
            self.forward_train(                               # пропускаем
весь трейновый датасет через созданную НС
                neural_networks=neural_networks,
                x_train=x_train,
                y_train=y_train)

            neural_networks.sort(key=lambda x: x.fitness)     # сортировка
НС по приспособленности
            best_nn = neural_networks[0]                      # выбираем
лучшую нейронную сеть

            if self.best_nn is None:
                self.best_nn = deepcopy(best_nn)    # используется функция
deepcopy, чтобы создать независимую копию лучшей нейронной сети, а не
просто ссылку на оригинальный объект.
            if best_nn.fitness < self.best_nn.fitness:
                self.best_nn = deepcopy(best_nn)

            result.append(self.best_nn.fitness)
            self.update_neuron_fitness()                      # обновляем
фунцию приспособленности нейрона
            self.neuron_population.crossover()                # скрещивание
популяции нейронов
            self.neuron_population.mutation()                 # мутация
популяции нейронов
            self.blueprint_population.crossover()             # скрещивание
популяции комбинации нейронов
            self.blueprint_population.mutation()              # мутация
популяции комбинации нейронов
            print('{}{}/{} best fitness = {}, current fitness = {}'
                  .format(CRLF, generation, generations_count,
self.best_nn.fitness, best_nn.fitness), end='')
        print(os.linesep)
        return result

    # тестирование сети. Возвращается массив среднеквадратичесных ошибок в
соотвествии с записями во входном наборе данных
    def test(self,
```

```
                x_test: np.array,   # массив входных данных
                y_test: np.array):  # массиы выходных данных
            if x_test.shape[0] != y_test.shape[0]:
                raise Exception()
            dataset_size = x_test.shape[0]
            result = []
            for i in range(dataset_size):
                error = self.forward(x=x_test[i], y=y_test[i])
                result.append(error)
            return result

    # внутренний метод, предназначенный для прохода по лучшей нейронной
сети с одним набором данных. Возвращает среднеквадратическую ошибку
    def forward(self, x: np.array, y: np.array):
            output = self.best_nn.forward(x)
            # print(output)
            return (mse(y, output),output)     # УБРАТЬ ВЫВОД output при
показе MSE

    # внутренний метод предназначенный для прохода всех комбинаций
нейронных сетей по всем входным данным.
    # по результатам прохода берется среднее значение среднеквадратичных
ошибок, полученных в результате прохода всех данных.
    # полученное значение является приспособленностью для нейронной сети и
для комбинации нейронной сети
    def forward_train(self,
                      neural_networks: List[NeuralNetwork],  # массив
нейронных сетей
                      x_train: np.array,                     # массив
входных данных
                      y_train: np.array):                    # массив
выходных данных
            dataset_size = x_train.shape[0]
            for i in range(len(neural_networks)):    # для каждой комбинации
НС
                errors = []
                for j in range(dataset_size):
                    output = neural_networks[i].forward(input_data=x_train[j])
# прямой проход по сети. Последовательно выполняется прямой проход по
слоям сети с помощью метода forward класса Layer
                    error = mse(y_true=y_train[j], y_pred=output)  # считаем
ошибку
                    errors.append(error)  # добавляем её в список всех ошибок
                avg_error = np.array(errors).mean()  # вычисляем среднюю
ошибку нейронной сети и популяции комбинаций нейронных сетей
                neural_networks[i].fitness = avg_error # средняя ошибка
нейронной сети
                self.blueprint_population[i].fitness = avg_error # средняя
ошибка популяции комбинаций нейронных сетей
```

```
    # внутренний метод,создающий массив нейронных сетей из комбинаций
нейронных сетей
    def create_neural_networks(self,
                               inputs_count:
int,                           # количество входов нейронной сети
                               outputs_count: int) ->
List[NeuralNetwork]:  # количество выходов нейронной сети
        result = []
        for population in self.blueprint_population:  # итерация по
комбинациям нейронов в популяции
            hidden_neurons = population.neurons       # инициализация
скрытых нейронов НС
            result.append(NeuralNetwork(              # создание массива
НС. Каждый элемент массива - объект класса NeuralNetwork
                hidden_neurons=hidden_neurons,
                inputs_count=inputs_count,
                outputs_count=outputs_count,
                neuron_population=self.neuron_population))
        return result

    # внутренний метод, обновляющий приспособленность нейронов, являющуюся
средним значением приспособленности 5 лучших нейронных сетей, включающих
данный нейрон
    def update_neuron_fitness(self):
        for neuron in self.neuron_population:
            fitness_list = []
            for population in self.blueprint_population:
                if neuron in population.neurons:
                    fitness_list.append(population.fitness)
                if len(fitness_list) == 5:
                    break
            neuron.fitness = np.array(fitness_list).mean() if
len(fitness_list) > 0 else 0.0

    def plot_network(self,
                     #  best_nn: NeuralNetwork,
                      x_train: np.array,
                      y_train: np.array):
      # Создание пустого графа
      G = nx.Graph()
      # Узлы входного слоя
      input_nodes = ['Input {}'.format(i+1) for i in
range(len(x_train[0]))]
      # Узлы скрытого слоя
      hidden_nodes = ['Hidden {}'.format(i+1) for i in
range(self.hidden_layer_size)]
      # Узлы выходного слоя
      output_nodes = ['Output {}'.format(i+1) for i in
range(len(y_train[0]))]
```

```
      # Добавление узлов в граф
      G.add_nodes_from(input_nodes)
      G.add_nodes_from(hidden_nodes)
      G.add_nodes_from(output_nodes)

      # Добавление ребер (связей) между узлами
      # G.add_edges_from([(i, j) for i in input_nodes for j in
hidden_nodes])
      # G.add_edges_from([(i, j) for i in hidden_nodes for j in
output_nodes])

      input_to_hidden_edges = []
      for i, hidd_node in enumerate(self.best_nn.input_weights):
        for j, weight in enumerate(self.best_nn.input_weights[i]):
          if weight == 0:
            continue
          else:
            input_to_hidden_edges.append(('Input {}'.format(j+1),'Hidden
{}'.format(i+1)))

      hidden_output_edges = []
      for i, out_node in enumerate(self.best_nn.output_weights):
        for j, weight in enumerate(self.best_nn.output_weights[i]):
          if weight == 0:
            continue
          else:
            hidden_output_edges.append(('Hidden {}'.format(j+1),'Output
{}'.format(i+1)))

      G.add_edges_from(input_to_hidden_edges)
      G.add_edges_from(hidden_output_edges)

      # Определение позиции узлов по столбцам
      pos = {}
      for i, node in enumerate(input_nodes):
        pos[node] = (1, i+0.5)
      for i, node in enumerate(hidden_nodes):
        pos[node] = (2, i)
      for i, node in enumerate(output_nodes):
        pos[node] = (3, i + 4)

      # Рисование графа
      nx.draw(G, pos, with_labels=True, node_color='lightblue',
edge_color='gray', node_size=1000, font_size=8, arrows=True)

      # Отображение графа
      plt.show()
```

1.11 Классы для загрузки данных

Представленные датасеты имеют схожую структуру. В начале файла имеется заголовок, в котором указано количество входных булевых и вещественных данных, и таких же выходных данных. Далее следуют записи о количестве записей для тренировки, валидации и тестирования. После заголовка следуют данные, значения которых разделены пробелами. Таким образом можно создать универсальный загрузчик данных в виде одного базового класса, задача которого – это непосредственно чтение данных, и дочерних классов, представляющих отдельные датасеты.

Исходный код классов представлен в листинге 10.

Листинг 10. Исходный код классов-загрузчиков данных.

```
def transform_data(dataset, inputs, outputs):
    input_data = np.zeros((len(dataset), inputs))
    output_data = np.zeros((len(dataset), outputs))
    for i in range(len(dataset)):
        data = [float(value) for value in dataset[i].split()]
        input_data[i] = data[:inputs]
        output_data[i] = data[inputs:]
    return input_data, output_data


def load(path):
    with open(path, 'r') as f:
        lines = f.readlines()

        bool_in = int(lines[0].split('=')[1])
        real_in = int(lines[1].split('=')[1])
        bool_out = int(lines[2].split('=')[1])
        real_out = int(lines[3].split('=')[1])
        training_examples_count = int(lines[4].split('=')[1])
        validation_examples_count = int(lines[5].split('=')[1])
        test_examples_count = int(lines[6].split('=')[1])

        inputs = bool_in + real_in
        outputs = bool_out + real_out

        current_line = 7
        train_x, train_y = transform_data(lines[current_line:current_line
+ training_examples_count], inputs, outputs)
        current_line += training_examples_count
```

```
        validation_x, validation_y =
transform_data(lines[current_line:current_line +
validation_examples_count], inputs, outputs)
        current_line += validation_examples_count
        test_x, test_y = transform_data(lines[current_line:current_line +
test_examples_count], inputs, outputs)

        return train_x, train_y, validation_x, validation_y, test_x,
test_y


class AbstractDataset(object):
    def __init__(self, path):
        self.train_x, self.train_y, self.validation_x, self.validation_y,
self.test_x, self.test_y = load(path)

    def get_train_data(self):
        return self.train_x, self.train_y

    def get_validation_data(self):
        return self.validation_x, self.validation_y

    def get_test_data(self):
        return self.test_x, self.test_y


class Cancer1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__( os.path.dirname(__file__) + '/data/cancer1.dt')


class Cancer2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/cancer2.dt')


class Cancer3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/cancer3.dt')


class Diabetes1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/diabetes1.dt')


class Diabetes2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/diabetes2.dt')
```

```
class Diabetes3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/diabetes3.dt')


class Glass1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/glass1.dt')


class Glass2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/glass2.dt')


class Glass3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/glass3.dt')


class Card1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/card1.dt')


class Card2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/card2.dt')


class Card3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/card3.dt')


class Flare1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/flare1.dt')


class Flare2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/flare2.dt')


class Flare3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/flare3.dt')


class Gene1Dataset(AbstractDataset):
    def __init__(self):
```

```
        super().__init__(os.path.dirname(__file__) + '/data/gene1.dt')



class Gene2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/gene2.dt')


class Gene3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/gene3.dt')


class Heart1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/heart1.dt')


class Heart2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/heart2.dt')


class Heart3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/heart3.dt')


class Horse1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/horse1.dt')


class Horse2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/horse2.dt')


class Horse3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/horse3.dt')


class Mushroom1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/mushroom1.dt')


class Mushroom2Dataset(AbstractDataset):
    def __init__(self):
```

```
        super().__init__(os.path.dirname(__file__) + '/data/mushroom2.dt')



class Mushroom3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/mushroom3.dt')


class Soybean1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/soybean1.dt')


class Soybean2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/soybean2.dt')


class Soybean3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/soybean3.dt')


class Thyroid1Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/thyroid1.dt')


class Thyroid2Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/thyroid2.dt')


class Thyroid3Dataset(AbstractDataset):
    def __init__(self):
        super().__init__(os.path.dirname(__file__) + '/data/thyroid3.dt')
```

2 Результаты работы программы

Протестируем работу алгоритма на датасете cancer1. Запустим три нейросети с различными параметрами и посмотрим, как обучается алгоритм SANE на 2000 поколениях.

Алгоритм инициализируется значениями от -1 до 1.

Таблица 1 – Параметры алгоритма

№	Размер популяции комбинаций нейронов	Размер популяции нейронов	Количество нейронов на скрытом слое	Количество связей для одного нейрона
1	10	500	10	11
2	50	1000	9	8
3	500	2000	20	8

Выведем изменение среднеквадратичной ошибки в процессе эволюции.

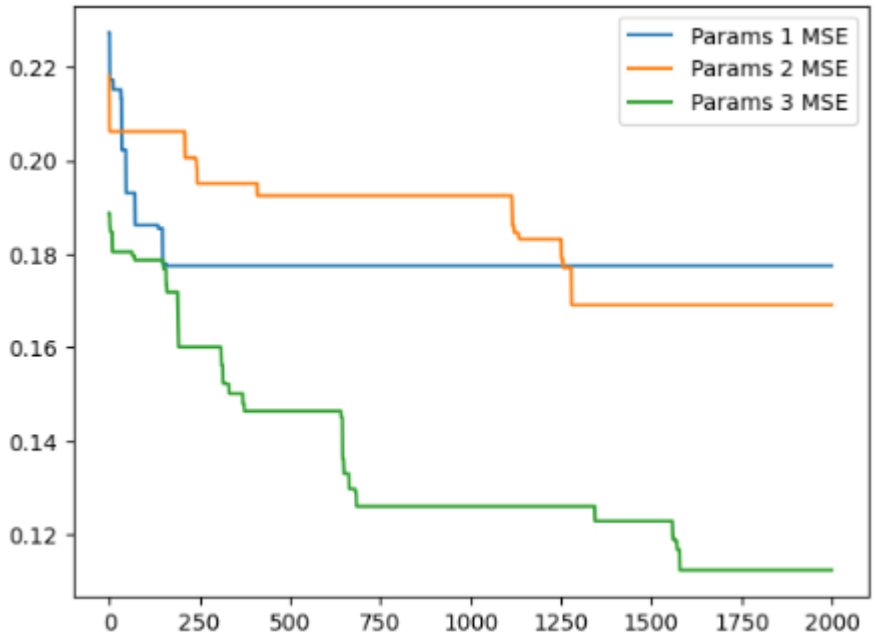

Рисунок 2 – Изменение среднеквадратичной ошибки в процессе эволюции

На рисунке 2 представлено изменение среднеквадратичной ошибки трех нейронных сетей с различными конфигурациями. За 2000 поколений ошибка у лучшей нейронной сети уменьшилась приблизительно до 0,05. Таким образом, алгоритм обучается, но имеет крайне низкую

производительность: расчёт занял почти 3,5 часа, при больших размерах популяций.

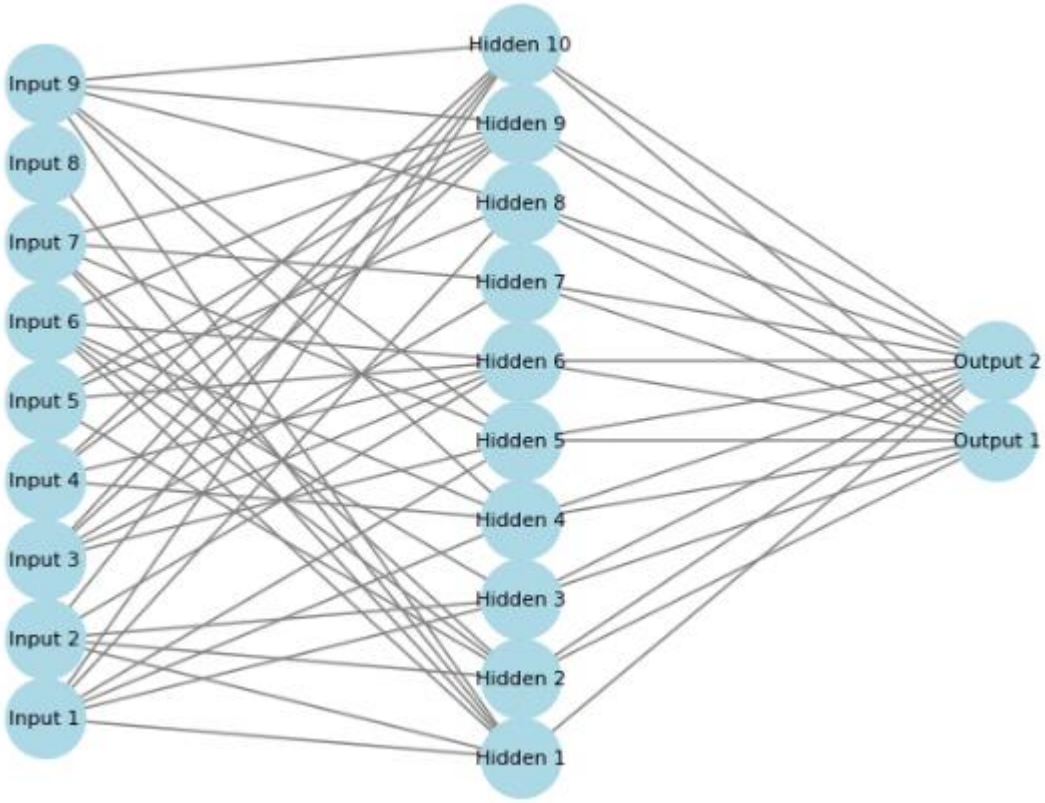

Рисунок 3 – Визуализация топологии первой нейронной сети

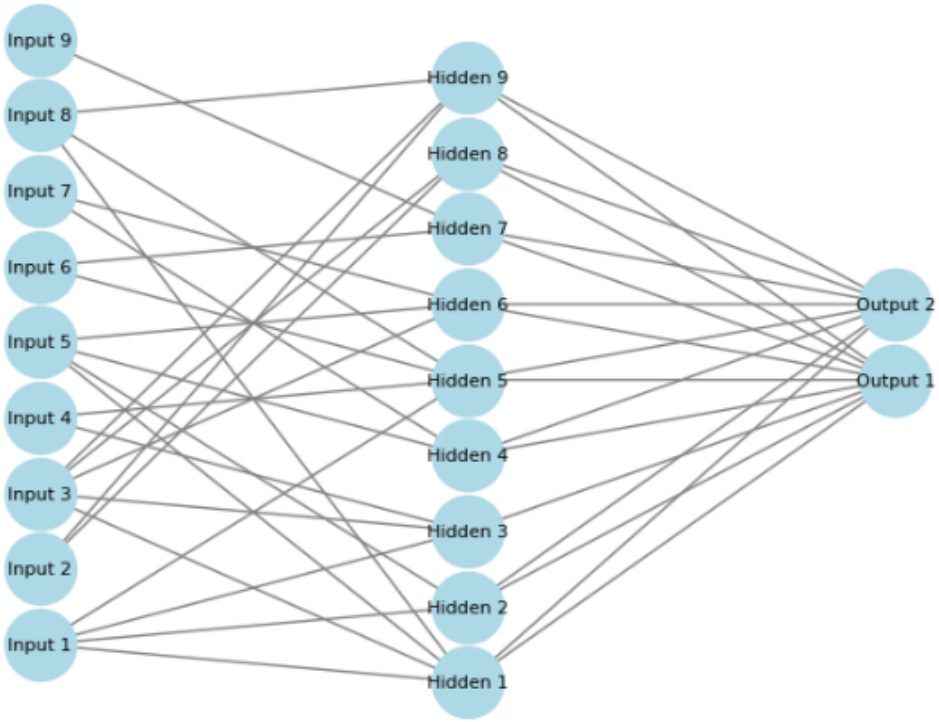

Рисунок 4 - Визуализация топологии второй нейронной сети

3 Вывод

В результате выполнения индивидуального задания был реализован нейроэволюционный алгоритм SANE. Были проанализированы результаты и показано, что алгоритм решает задачу.