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](https://up.tpu.ru/view/detali.html?id=22531)

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**

[1 Реализация алгоритма 3](#_Toc143200147)

[1.1 Класс RealGene 5](#_Toc143200148)

[1.2 Класс Gene 5](#_Toc143200149)

[1.3 Класс Neuron 10](#_Toc143200150)

[1.4 Класс NeuronPopulation 14](#_Toc143200151)

[1.5 Класс Blueprint 17](#_Toc143200152)

[1.6 Класс BlueprintPopulation 19](#_Toc143200153)

[1.7 Классы функций активации 23](#_Toc143200154)

[1.8 Класс Layer 24](#_Toc143200155)

[1.9 Класс NeuralNetwork 25](#_Toc143200156)

[1.10 Класс SANEAlgorithm 27](#_Toc143200157)

[1.11 Классы для загрузки данных 35](#_Toc143200158)

[2 Результаты работы программы 40](#_Toc143200159)

[3 Вывод 42](#_Toc143200160)

**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.

1. **Обновить приспособленность нейрона = средняя приспособленность 5**

**лучших ИНС, включающих этот нейрон.**

1. **Сортировка популяции нейронов по приспособленности.**
2. **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.

1. **Мутация.**

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

1. **Скрещивание комбинаций.**

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

1. **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.

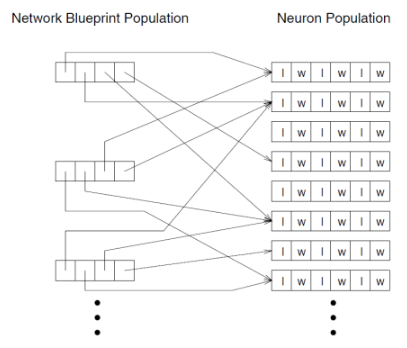


Рисунок 1 – Кодирование информации об ИНС в алгоритме SANE

Рассмотрим основные классы, участвующие в реализации алгоритма.

* 1. **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;

1. init – инициализация векторов входных и выходных весов случайными значениями;

* min\_value – минимальное случайно сгенерированное число;
* max\_value – максимальное случайно сгенерированное число;

1. mutation – мутация векторов весов. Используется нормальное распределение в окрестности значения гена.
2. 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. **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;

1. mutation – мутация весов и меток; для метки используется инвертирование бита; для веса вызывается метод mutation класса RealGene;
2. 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. **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;

1. 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;

1. 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;

1. get\_input\_weights – получение вектора весов входных соединений скрытого слоя. Параметры:

* neurons\_count – количество нейронов в скрытом слое

1. get\_output\_weights - получение вектора весов выходных соединений скрытого слоя. Параметры:

* neurons\_count – количество нейронов в скрытом слое;

1. mutation – мутация генов. Вызывается метод mutation класса Gene для каждого гена.
2. 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. **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;

1. 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;

1. crossover – скрещивание верхней четверти лучших нейронов;
2. mutation - мутация всех нейронов;
3. \_\_ getitem\_\_ – встроенный метод языка Python, позволяющий обращаться к данному объекту как к коллекции. При этом возвращается соответствующий индексу нейрон.
4. \_\_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. **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.

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

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

* neuron\_population – популяция нейронов;

1. select\_neurons – внутренний метод, возвращающий список индексов случайно выбранных нейронов из популяции нейронов. При этом индексы являются уникальными.
2. crossover – скрещивание верхней четверти лучших комбинации нейронов. Для скрещивания вызывается метод crossover класса Blueprint. Потомки при этом заменяют худшие особи.
3. mutation – мутация всех комбинаций нейронов.
4. \_\_getitem\_\_ – встроенный метод языка Python, позволяющий обращаться к данному объекту как к коллекции. При этом возвращается соответствующий индексу нейрон.
5. \_\_len\_\_ – встроенный метод языка Python, позволяющий получить размер коллекции. При этом возвращается размер массива нейронов
6. \_\_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. **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. **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.

1. 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. **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.

1. 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. **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.

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

* min\_value – минимальное случайно сгенерированное число;
* max\_value – максимальное случайно сгенерированное число.

1. train – тренировка нейронных сетей. Параметры:

* generations\_count – количество поколений;
* x\_train – массив входных данных;
* y\_train – массив выходных данных.

1. 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.

1. 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.

1. 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.

1. 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.

1. 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. **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')

1. **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.

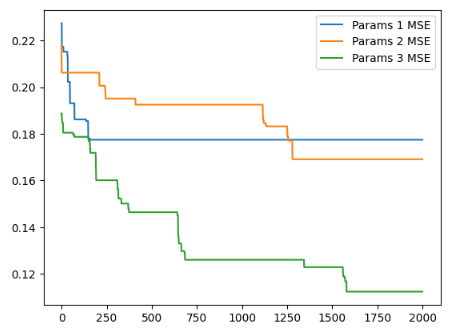


Рисунок 2 – Изменение среднеквадратичной ошибки в процессе эволюции

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.

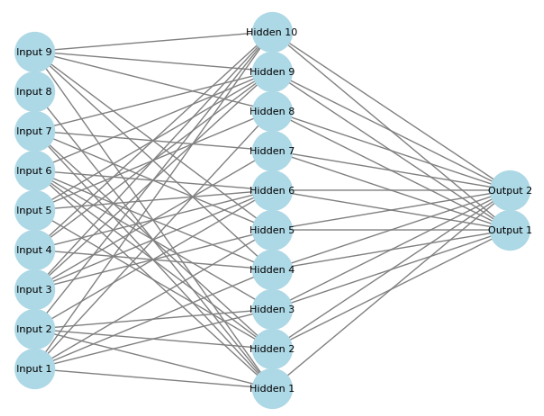


Рисунок 3 – Визуализация топологии первой нейронной сети

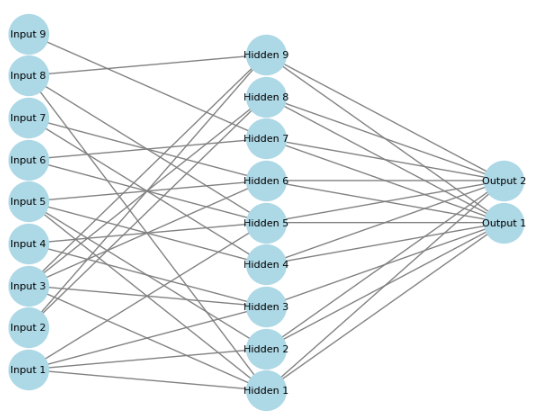


Рисунок 4 - Визуализация топологии второй нейронной сети

1. **Вывод**

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