TOMSK POLYTECHNIC UNIVERSITY ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский Томский политехнический университет» (ТПУ)

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

Индивидуальное задание по дисциплине
«Нейроэволюционные вычисления»

Тема работы
Реализация алгоритма ESP

Студент

Группа	ФИО	Подпись	Дата
8ВМ22	Ткачева С.В.		

Руководитель

Должность	ФИО	Подпись	Дата
Старший преподаватель ОИТ	Григорьев Д.С.		

Томск – 2023

Содержание

1 Реализация алгоритма

1.1 Класс Neuron

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

1. input_count - количество входов
2. output_count - количество выходов
3. id - уникальный идентификатор нейрона
4. input_weights - вектор входных весов
5. output_weights - вектор выходных весов
6. cumulative_fitness - кумулятивная приспособленность нейрона
7. avg_fitness - средняя приспособленность нейрона, равная отношению кумулятивной приспособленности к количеству попыток, в которых участвовал данный нейрон
8. trials - количество попыток, в которых участвовал данный нейрон

В классе имеются следующие методы:

1. __init__ - конструктор; параметры:
   a) input_count - количество входов
   b) output_count - количество выходов
   c) neuron_id - уникальный идентификатор нейрона
2. init - инициализация векторов входных и выходных весов случайными значениями; параметры:
   a) min_value - минимальное случайно сгенерированное число
   b) max_value - максимальное случайно сгенерированное число
3. fit_avg_fitness - корректировка средней приспособленности
4. mutation - мутация векторов весов, используется распределение Коши
5. crossover - скрещивание двух нейронов, используется одноточечный кроссинговер; в качестве результата возвращаются два новых нейрона; параметры:
   a) parent1 - первый родитель

b) parent2 - второй родитель

Исходный код класса представлен в листинге 1.

Листинг 1. Исходный код класса Neuron.

```
import numpy as np
import random


def crossover_weights(parent1: np.array, parent2: np.array):
    weights_count = parent1.shape[0]
    crossover_point = random.randrange(weights_count)
    child1, child2 = np.zeros(weights_count), np.zeros(weights_count)
    child1[:crossover_point] = parent1[:crossover_point]
    child1[crossover_point:] = parent2[crossover_point:]
    child2[:crossover_point] = parent2[:crossover_point]
    child2[crossover_point:] = parent1[crossover_point:]
    return child1, child2


class Neuron(object):
    def __init__(self,
                 input_count: int,
                 output_count: int,
                 neuron_id: int):
        self.input_count = input_count
        self.output_count = output_count
        self.id = neuron_id
        self.input_weights = None
        self.output_weights = None
        self.cumulative_fitness = 0.0
        self.avg_fitness = 0.0
        self.trials = 0

    def init(self, min_value: float, max_value: float):
        self.input_weights = np.random.uniform(
            low=min_value,
            high=max_value,
```

```
            size=self.input_count)
        self.output_weights = np.random.uniform(
            low=min_value,
            high=max_value,
            size=self.output_count)

    def fit_avg_fitness(self):
        self.avg_fitness = self.cumulative_fitness / self.trials

    def mutation(self):
        self.input_weights += np.random.standard_cauchy(self.input_count) *
0.05
        self.output_weights += np.random.standard_cauchy(self.output_count) *
0.05

    @staticmethod
    def crossover(parent1, parent2):
        input_count = parent1.input_count
        output_count = parent1.output_count
        child1 = Neuron(
            input_count=input_count,
            output_count=output_count,
            neuron_id=parent1.id)
        child2 = Neuron(
            input_count=input_count,
            output_count=output_count,
            neuron_id=parent2.id)
        child1.input_weights, child2.input_weights = crossover_weights(
            parent1=parent1.input_weights,
            parent2=parent2.input_weights)
        child1.output_weights, child2.output_weights = crossover_weights(
            parent1=parent1.output_weights,
            parent2=parent2.output_weights)
        return child1, child2
```

1.2 Класс NeuronSubPopulation

Класс NeuronSubPopulation реализует логику работы с подпопуляцией. Имеются следующие поля:

1. population - массив, хранящий нейроны (объекты класса Neuron)
2. last_generations_count - количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация
3. trials_per_neuron - минимальное количество попыток, в которых должен учавствовать нейрон
4. id - уникальный идентификатор подпопуляции
5. generation - счётчик, указывающий на количество поколений текущей подпопуляции
6. best_neurons - словарь, хранящий список приспособленностей лучших нейронов. Максимальный размер списка равен last_generations_count

В классе имеются следующие методы:

1. __init__ - конструктор. В конструкторе происходит создание подпопуляции нейронов. Параметры:
   a) population_size - количество особей (нейронов) в подпопуляции
   b) input_count - количество входов для нейрона (необходим для создания нейрона)
   c) output_count - количество выходов для нейрона (необходим для создания нейрона)
   d) last_generations_count - количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация
   e) trials_per_neuron - минимальное количество попыток, в которых должен учавствовать нейрон
   f) subpopulation_id - уникальный идентификатор подпопуляции

2. init - инициализация подпопуляции нейронов. Для каждого нейрона вызывается метод init, в который передаются параметры метода. Параметры:

a) min_value - минимальное случайно сгенерированное число

b) max_value - максимальное случайно сгенерированное число

3. get_neuron - получение случайного нейрона из подпопуляции

4. is_trials_completed - метод проверяет, что каждый из нейронов в подпопуляции поучаствовал в оценке работы нейронной сети заданное минимальное количество раз (trials_per_neuron)

5. reset_trials - сброс количества попыток для каждого нейрона в подпопуляции

6. fit_avg_fitness - корректировка средней приспособленности для каждого нейрона в подпопуляции

7. crossover - скрещивание нейронов в подпопуляции. Скрещиваются 1/4 часть лучших особей в подпопуляции, потомки добавляются в конец подпопуляции. Для скрещивающихся нейронов вызывается метод crossover класса Neuron

8. mutation - мутация нижней половины подпопуляции. Вызывается метод mutation класса Neuron

9. get_best_neuron - получение лучшего нейрона в подпопуляции

10. check_degeneration - проверка вырождения подпопуляции. Выбирается лучший нейрон, и его средняя приспособленность добавляется в список приспособленностей лучших нейронов

11. burst_mutation - взрывная мутация. Происходит создание новой подпопуляции нейронов вокруг входного нейрона. Для инициализации весов нейронов используется распределение Коши в точке, соответствующей весу нейрона. Параметры:

a) neuron - нейрон, вокруг которого генерируется новая подпопуляция

Исходный код класса представлен в листинге 2.

Листинг 2. Исходный код класса NeuronSubPopulation.

```
import random
import numpy as np
from collections import deque
from .neuron import Neuron


class NeuronSubPopulation(object):
    def __init__(self,
                 population_size: int,
                 input_count: int,
                 output_count: int,
                 last_generations_count: int,
                 trials_per_neuron: int,
                 subpopulation_id: int):
        self.population = []
        for i in range(population_size):
            self.population.append(Neuron(
                input_count=input_count,
                output_count=output_count,
                neuron_id=i))
        self.last_generations_count = last_generations_count
        self.trials_per_neuron = trials_per_neuron
        self.id = subpopulation_id
        self.generation = 0
        self.best_neurons = {}

    def init(self, min_value: float, max_value: float):
        for neuron in self.population:
            neuron.init(
                min_value=min_value,
                max_value=max_value)

    def get_neuron(self) -> Neuron:
        return random.choice(self.population)
```

```
    def is_trials_completed(self) -> bool:
        trials = [neuron.trials for neuron in self.population]
        return min(trials) >= self.trials_per_neuron

    def reset_trials(self):
        for neuron in self.population:
            neuron.trials = 0

    def fit_avg_fitness(self):
        for neuron in self.population:
            neuron.fit_avg_fitness()

    def crossover(self):
        self.population.sort(key=lambda x: x.avg_fitness)
        selected_neurons_count = int(len(self.population) / 4)
        selected_neurons_count -= selected_neurons_count % 2
        for i in range(0, selected_neurons_count, 2):
            parent1 = self.population[i]
            parent2 = self.population[i + 1]
            child1, child2 = Neuron.crossover(
                parent1=parent1,
                parent2=parent2)
            self.population[-selected_neurons_count + i] = child1
            self.population[-selected_neurons_count + i + 1] = child2

    def mutation(self):
        bottom_half = int(len(self.population) / 2)
        for neuron in self.population[bottom_half:]:
            neuron.mutation()

    def get_best_neuron(self) -> Neuron:
        self.population.sort(key=lambda x: x.avg_fitness)
        return self.population[0]

    def check_degeneration(self):
```

```
        best_neuron = self.get_best_neuron()
        if best_neuron.id in self.best_neurons.keys():
            self.best_neurons[best_neuron.id].append(best_neuron.avg_fitness)
        else:
            self.best_neurons[best_neuron.id] =
deque(maxlen=self.last_generations_count)
        clear_best_neurons = False
        for neuron_id, fitness_list in self.best_neurons.items():
            if len(fitness_list) == fitness_list.maxlen:
                if self.population[neuron_id].avg_fitness > min(fitness_list):
                    self.burst_mutation(neuron=best_neuron)
                    clear_best_neurons = True
                    break
        if clear_best_neurons:
            self.best_neurons = {}


    def burst_mutation(self, neuron: Neuron):
        print('Взрывная мутация для подпопуляции {0:>3d}. Текущее поколение
{1:>3d}'
              .format(self.id, self.generation))
        input_count = neuron.input_count
        output_count = neuron.output_count
        new_population = []
        for i in range(len(self.population)):
            new_neuron = Neuron(
                input_count=input_count,
                output_count=output_count,
                neuron_id=i)
            new_neuron.input_weights = \
                np.random.standard_cauchy(input_count) * 0.05 +
neuron.input_weights
            new_neuron.output_weights = \
                np.random.standard_cauchy(output_count) * 0.05 +
neuron.output_weights
            new_population.append(new_neuron)
        self.population = new_population
```

```
        self.generation += 1
```

1.3 Класс NeuronPopulation

Класс NeuronPopulation - реализует логику работы с популяцией подпопуляций. Имеются следующие поля:

1. population - список подпопуляций

В классе имеются следующие методы:

1. __init__ - конструктор. В конструкторе происходит создание попдпопуляций. Параметры:
    1. population_size - количество подпопуляций (так же количество нейронов в скрытом слое)
    2. subpopulation_size - размер подпопуляции
    3. input_count - количество входов для нейрона
    4. output_count - количество выходов для нейрона
    5. last_generations_count - количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация
    6. trials_per_neuron - минимальное количество попыток, в которых должен учавствовать нейрон
2. init - инициализация подпопуляций нейронов. Для каждой подпопуляции вызывается метод init, в который передаются параметры метода. Параметры:
    1. min_value - минимальное случайно сгенерированное число
    2. max_value - максимальное случайно сгенерированное число
3. get_neurons - метод возвращает список нейронов, при этом из каждой подпопуляции берётся по одному случайному нейрону вызовом метода get_neuron класса NeuronSubPopulation
4. get_best_neurons - метод возвращает список нейронов, при этом из каждой подпопуляции берётся по одному лучшему нейрону вызовом метода get_best_neuron класса NeuronSubPopulation

5. is_trials_completed - метод проверяет, что каждый из нейронов во всех подпопуляциях поучаствовал в оценке работы нейронной сети заданное минимальное количество раз

6. reset_trials - сброс количества попыток для каждого нейрона во всех подпопуляциях

7. fit_avg_fitness - корректировка средней приспособленности для всех подпопуляций

8. crossover - скрещивание всех подпопуляций

9. mutation - мутация всех подпопуляций

10. check_degeneration - проверка вырождения всех подпопуляций

Исходный код класса представлен в листинге 3.

Листинг 3. Исходный код класса NeuronPopulation.

```
from typing import List
from .neuron_subpopulation import NeuronSubPopulation
from .neuron import Neuron


class NeuronPopulation(object):
    def __init__(self,
                 population_size: int,
                 subpopulation_size: int,
                 input_count: int,
                 output_count: int,
                 last_generations_count: int,
                 trials_per_neuron: int):
        self.population = []
        for i in range(population_size):
            self.population.append(NeuronSubPopulation(
                population_size=subpopulation_size,
                input_count=input_count,
                output_count=output_count,
                last_generations_count=last_generations_count,
                trials_per_neuron=trials_per_neuron,
                subpopulation_id=i))
```

```
    def init(self, min_value: float, max_value: float):
        for i in range(len(self.population)):
            self.population[i].init(
                min_value=min_value,
                max_value=max_value)

    def get_neurons(self) -> List[Neuron]:
        return list(map(lambda x: x.get_neuron(), self.population))

    def get_best_neurons(self) -> List[Neuron]:
        return list(map(lambda x: x.get_best_neuron(), self.population))

    def is_trials_completed(self) -> bool:
        trials = [subpopulation.is_trials_completed() for subpopulation in
self.population]
        return not (False in trials)

    def reset_trials(self):
        for subpopulation in self.population:
            subpopulation.reset_trials()

    def fit_avg_fitness(self):
        for subpopulation in self.population:
            subpopulation.fit_avg_fitness()

    def crossover(self):
        for subpopulation in self.population:
            subpopulation.crossover()

    def mutation(self):
        for subpopulation in self.population:
            subpopulation.mutation()

    def check_degeneration(self):
```

```
        for subpopulation in self.population:
            subpopulation.check_degeneration()
```

1.4 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 сигмоидальной функции активации в классе Sigmoid. Исходный код классов представлен в листинге 4.

Листинг 4. Исходный код классов функции активации.

```
import numpy as np


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))
```

1.5 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:
   a) weights - матрица весов
   b) activation - функция активации
2. forward - прямой проход по слою. Матрица весов умножается на вектор-столбец входных данных, к каждому элементу полученного вектор-столбца применяется функция активации. Параметры:
   a) input_data - вектор входных данных

Исходный код класса представлен в листинге 5.

Листинг 5. Исходный код класса Layer.

```
import numpy as np
from .activations import AbstractActivationFunction


class Layer(object):
    def __init__(self,
                 weights: np.array,
                 activation: AbstractActivationFunction):
        self.weights = weights
        self.activation = activation

    def forward(self,
                input_data: np.array) -> np.array:
        return self.activation.forward(
            input_data=np.dot(self.weights, input_data))
```

1.6 Класс NeuralNetwork

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

1. input_weights - матрица весов скрытого слоя

2. output_weights - матрица весов выходного слоя

3. 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 слоёв сети. Параметры:

a) hidden_neurons - список нейронов, из которых будут созданы скрытый и выходной слои сети

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

a) input_data - вектор входных данных

Исходный код класса представлен в листинге 6.

Листинг 6. Исходный код класса NeuralNetwork.

```
from typing import List
import numpy as np
from .layer import Layer
from .neuron import Neuron
from .activations import Sigmoid


class NeuralNetwork(object):
    def __init__(self,
                 hidden_neurons: List[Neuron]):
        input_count = hidden_neurons[0].input_count
        output_count = hidden_neurons[0].output_count
        self.input_weights = np.zeros((len(hidden_neurons), input_count))
        output_weights = np.zeros((len(hidden_neurons), output_count))
        for i in range(len(hidden_neurons)):
            self.input_weights[i] = hidden_neurons[i].input_weights
            output_weights[i] = hidden_neurons[i].output_weights
        self.output_weights = np.zeros((output_count, len(hidden_neurons)))
        for i in range(output_count):
```

```
            self.output_weights[i] = output_weights[:, i]
        self.layers = []
        self.layers.append(Layer(
            weights=self.input_weights,
            activation=Sigmoid()))
        self.layers.append(Layer(
            weights=self.output_weights,
            activation=Sigmoid()))

    def forward(self, input_data: np.array) -> np.array:
        output = input_data
        for layer in self.layers:
            output = layer.forward(output)
        return output
```

1.7 Класс ESPAlgorithm

Класс ESPAlgorithm реализует логику работы с алгоритмом ESP. Имеются следующие поля:

1. population - популяция подпопуляций
2. best_nn - лучшая нейронная сеть
3. best_nn_fitness - приспособленность лучшей нейронной сети

В классе имеются следующие методы:

1. __init__ - конструктор. Происходит создание популяции подпопуляций. Параметры:
    a) hidden_layer_size - количество нейронов в скрытом слое
    b) population_size - размер каждой из подпопуляции
    c) input_count - количество входов для нейрона
    d) output_count - количество выходов для нейрона
    e) last_generations_count - количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация

f) trials_per_neuron - минимальное количество попыток, в которых должен учавствовать нейрон

2. init - инициализация популяции подпопуляций, параметры:
   a) min_value - минимальное случайно сгенерированное число
   b) max_value - максимальное случайно сгенерированное число
3. train - тренировка нейронных сетей, параметры:
   a) generations_count - количество поколений
   b) x_train - массив входных данных
   c) y_train - массив выходных данных
4. update_best_nn - обновление лучшей нейронной сети
5. check_fitness - проверка приспособленностей нейронных сетей, параметры:
   a) x_train - массив входных данных
   b) y_train - массив выходных данных

Рассмотрим подробнее работу алгоритма (метод train). Запускается цикл согласно количеству поколений (generations_count). В цикле вызывается метод check_fitness. В методе check_fitness запускается цикл, который выполняется до тех пор, пока каждый нейрон в каждой из подпопуляций не поучаствует в работе нейронной сети. Для данной проверки используется метод is_trials_completed класса NeuronPopulation. В цикле происходит выборка нейронов - случайно, по одному из каждой подпопуляции. Выбранные нейроны будут участвовать в работе нейронной сети, поэтому для них увеличивается счётчик количества попыток. Из выбранных нейронов создаётся нейронная сеть (объект NeuralNetwork). Для данной нейронной сети делается прямой проход по каждому набору входных данных (вызов вспомогательного метода forward_train). Для каждого набора данных вычисляется среднеквадратичная ошибка и возвращается среднее значение ошибок. Данное среднее значение будет являться кумулятивной приспособленностью выбранных нейронов. После того, как каждый нейрон поучавствует в работе нейронной сети, метод check_fitness завершает свою

работу, возвращая количество попыток, которое потребовалось для того, чтобы каждый нейрон поучаствовал в работе сети. Получив значение кумулятивной приспособленности, происходит обновление средней приспособленности вызовом метода fit_avg_fitness класса NeuronPopulation. Затем делается проверка на вырождение подпопуляций вызовом метода check_degeneration класса NeuronPopulation. Далее, для каждой из подпопуляций делается скрещивание вызовом метода crossover класса NeuronPopulation. Далее, для каждой из подпопуляций делается мутация вызовом метода mutation класса NeuronPopulation. В конце итерации происходит сброс счётчика попыток у каждого из нейронов. Данные шаги выполняются заданное количество раз.

Исходный код класса представлен в листинге 7.

Листинг 7. Исходный код класса ESPAlgorithm.

```
from typing import List
import numpy as np
from statistics import mean
import time
from .neuron_population import NeuronPopulation
from .neural_network import NeuralNetwork
from .neuron import Neuron
from .utils import mse


def forward_train(neural_network: NeuralNetwork,
                  x_train: np.array,
                  y_train: np.array) -> float:
    dataset_size = x_train.shape[0]
    errors = []
    for i in range(dataset_size):
        output = neural_network.forward(input_data=x_train[i])
        error = mse(y_true=y_train[i], y_pred=output)
        errors.append(error)
    return np.array(errors).mean()
```

```
def increment_trials(neurons: List[Neuron]):
    for neuron in neurons:
        neuron.trials += 1


class ESPAlgorithm(object):
    def __init__(self,
                 hidden_layer_size: int,
                 population_size: int,
                 input_count: int,
                 output_count: int,
                 last_generations_count: int,
                 trials_per_neuron: int):
        self.population = NeuronPopulation(
            population_size=hidden_layer_size,
            subpopulation_size=population_size,
            input_count=input_count,
            output_count=output_count,
            last_generations_count=last_generations_count,
            trials_per_neuron=trials_per_neuron)
        self.best_nn = None
        self.best_nn_fitness = 0.0

    def init(self, min_value: float, max_value: float):
        self.population.init(
            min_value=min_value,
            max_value=max_value)

    def train(self,
              generations_count: int,
              x_train: np.array,
              y_train: np.array):
        result = []
        for generation in range(generations_count):
            start_time = time.time()
```

```
            trials_count = self.check_fitness(
                x_train=x_train,
                y_train=y_train)
            self.population.fit_avg_fitness()
            if self.update_best_nn():
                result.append((generation, self.best_nn_fitness))
            self.population.check_degeneration()
            self.population.crossover()
            self.population.mutation()
            full_time = time.time() - start_time
            print('Поколение {0:>5d}, Количество попыток {1:>3d}, '
                  'Приспособленность лучшего нейрона {2:2.6f}, Время выполнения
{3:3.3f} s'
                  .format(generation, trials_count, self.best_nn_fitness,
full_time))
            self.population.reset_trials()
        return result

    def update_best_nn(self):
        best_neurons = self.population.get_best_neurons()
        best_nn = NeuralNetwork(
            hidden_neurons=best_neurons)
        best_nn_fitness = mean([neuron.avg_fitness for neuron in best_neurons])
        if self.best_nn is None or best_nn_fitness < self.best_nn_fitness:
            self.best_nn = best_nn
            self.best_nn_fitness = best_nn_fitness
            return True
        return False

    def check_fitness(self,
                      x_train: np.array,
                      y_train: np.array) -> int:
        trials_count = 0
        while not self.population.is_trials_completed():
            selected_neurons = self.population.get_neurons()
            increment_trials(selected_neurons)
```

```
            neural_network = NeuralNetwork(
                hidden_neurons=selected_neurons)
            error = forward_train(
                neural_network=neural_network,
                x_train=x_train,
                y_train=y_train)
            for neuron in selected_neurons:
                neuron.cumulative_fitness += error
            trials_count += 1
        return trials_count
```

2 Результаты работы программы

Для тестирования работы алгоритма применяется датасет cancer1. Параметры алгоритма следующие:

1. количество нейронов в скрытом слое: 10
2. количество нейронов в подпопуляции: 15
3. количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация: 2
4. минимальное количество попыток, в которых должен участвовать нейрон: 2
5. количество поколений: 5000

Алгоритм инициализируется значениями от -1 до 1.

Результат представлен на рисунке 1.

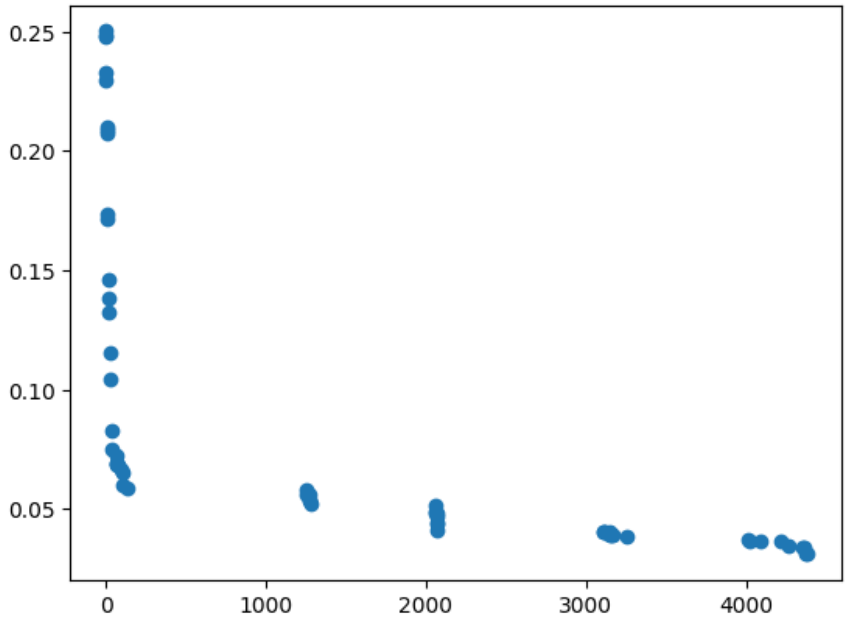

Рисунок 1. Результаты работы программы.

На данном рисунке представлено изменение среднеквадратичной ошибки лучшей нейронной сети в процессе работы сети. За 5000 поколений среднеквадратичная ошибка уменьшилась до значения 0.031031. Таким образом алгоритм обучается, однако имеет низкую производительность. Просчёт одного поколения занимает около двух секунд.

3 Вывод

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