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

**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 3](#_Toc136540518)

[1.1 Класс Neuron 3](#_Toc136540519)

[1.2 Класс NeuronSubPopulation 6](#_Toc136540520)

[1.3 Класс NeuronPopulation 11](#_Toc136540521)

[1.4 Классы функций активации 14](#_Toc136540522)

[1.5 Класс Layer 14](#_Toc136540523)

[1.6 Класс NeuralNetwork 15](#_Toc136540524)

[1.7 Класс ESPAlgorithm 17](#_Toc136540525)

[2 Результаты работы программы 22](#_Toc136540526)

[3 Вывод 23](#_Toc136540527)

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:
   * 1. input\_count - количество входов
     2. output\_count - количество выходов
     3. neuron\_id - уникальный идентификатор нейрона
2. init - инициализация векторов входных и выходных весов случайными значениями; параметры:
   1. min\_value - минимальное случайно сгенерированное число
   2. 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:
   1. parent1 - первый родитель
   2. 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:
   1. population\_size - количество особей (нейронов) в подпопуляции
   2. input\_count - количество входов для нейрона (необходим для создания нейрона)
   3. output\_count - количество выходов для нейрона (необходим для создания нейрона)
   4. last\_generations\_count - количество последних поколений, для которых приспособленность лучшей особи не менялась, запускается взрывная мутация
   5. trials\_per\_neuron - минимальное количество попыток, в которых должен учавствовать нейрон
   6. subpopulation\_id - уникальный идентификатор подпопуляции
2. init - инициализация подпопуляции нейронов. Для каждого нейрона вызывается метод init, в который передаются параметры метода. Параметры:
3. min\_value - минимальное случайно сгенерированное число
4. max\_value - максимальное случайно сгенерированное число
5. get\_neuron - получение случайного нейрона из подпопуляции
6. is\_trials\_completed - метод проверяет, что каждый из нейронов в подпопуляции поучаствовал в оценке работы нейронной сети заданное минимальное количество раз (trials\_per\_neuron)
7. reset\_trials - сброс количества попыток для каждого нейрона в подпопуляции
8. fit\_avg\_fitness - корректировка средней приспособленности для каждого нейрона в подпопуляции
9. 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
10. mutation - мутация нижней половины подпопуляции. Вызывается метод mutation класса Neuron
11. get\_best\_neuron - получение лучшего нейрона в подпопуляции
12. check\_degeneration - проверка вырождения подпопуляции. Выбирается лучший нейрон, и его средняя приспособленность добавляется в список приспособленностей лучших нейронов
13. 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:
    1. 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:
   1. weights - матрица весов
   2. 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:
   1. 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:
   1. 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:
   1. 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:
   1. hidden\_layer\_size - количество нейронов в скрытом слое
   2. population\_size - размер каждой из 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 популяции подпопуляций, параметры:
   1. min\_value - минимальное случайно сгенерированное число
   2. max\_value - максимальное случайно сгенерированное число
3. train - тренировка нейронных сетей, параметры:
   1. generations\_count - количество поколений
   2. x\_train - массив входных данных
   3. y\_train - массив выходных данных
4. update\_best\_nn - обновление лучшей нейронной сети
5. check\_fitness - проверка приспособленностей нейронных сетей, параметры:
   1. x\_train - массив входных данных
   2. 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.