

|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |

**Институт** Информационных Технологий

**Кафедра** Вычислительной Техники

**ПРАКТИЧЕСКАЯ РАБОТА**

**по дисциплине**

**«Разработка систем поддержки принятий решений»**

Студент группы: ИКБО-14-20 Васютин И.А. *(Фамилия студента)*

Руководитель самостоятельной работы Холмогоров В.В.

*(Фамилия преподавателя)*

|  |  |  |
| --- | --- | --- |
|  |  |  |

Москва 2023

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc137862736)

[1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 4](#_Toc137862737)

[2 ПРАКТИЧЕСКИЙ РАЗДЕЛ 6](#_Toc137862738)

[ЗАКЛЮЧЕНИЕ 9](#_Toc137862739)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 10](#_Toc137862740)

[ПРИЛОЖЕНИЯ 11](#_Toc137862741)

# ВВЕДЕНИЕ

В данной практической работе будут разработаны гибридные системы для решения задачи исключающего ИЛИ. В качестве взаимодействующих компонентов предполагается использовать многослойную полносвязную нейронную сеть и различные эволюционные алгоритмы для её обучения на обучающей выборке.

Функция исключающего ИЛИ считается линейно неразделимой, для графического представления данной функции необходимо более одной гиперплоскости. Данное ограничение делает невозможным персептронное представление функции. Однако, при каскадной связи нескольких однослойных нейронных сетей — это ограничение становится преодолимым.

В целом, обучение многослойной нейронной сети сводится к задаче минимизации её ошибки, а это значит, что обучения подойдёт любой такой алгоритм, для которого характерно решение задачи оптимизации невыпуклой функции. В качестве таких алгоритмов для обучения можно рассмотреть генетический алгоритм, алгоритм, основанный на число эволюционной стратегии, роевые алгоритмы оптимизации функции многих переменных.

# 1 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

**1.1 Описание системы**

В качестве алгоритмов обучения рассматриваются

1. Генетический алгоритм
2. Чисто эволюционный

В качестве признакового пространства будут использоваться веса нейронной сети. Каждый ген будет представлять из себя бинарную последовательность, где каждые пять бит будут отведены под какой-то конкретный вес: 1 бит под знак веса, 4 бита под интервалы абсолютного значения веса.

Генетический алгоритм будет иметь операторы мутации и селекции. Селекция будет подразумевать турнирный отбор с применением стратегии элитаризма. Такой подход имеет определенные недостатки. Одним из таких недостатков является потеря полезных аллелей вместе с вымершими в результате турнирного отбора особями. Для такого случая в алгоритме предусмотрена вторая попытка на турнире. В следующей реализации турнирного отбора с определённой высокой вероятностью выживает альфа-особь и с определённой низкой вероятностью выживает случайная особь из числа участников турнира.

Следующим этапом алгоритма определено одноточечное скрещивание. Оно также имеет свою вероятность: либо происходит скрещивание между выжившими особями, либо эти две выбранные для скрещивания особи, не оставляя потомства, переходят в новую популяцию.

С не очень высокой вероятностью в генетическом алгоритме происходит мутация. Если особь выбирается для мутации, то с определённой вероятностью некоторые её бинарные гены отражаются.

Применённая эволюционная стратегия имеет ряд отличий. Одно из самых заметных – наличие фиксированной популяции. Не смотря на наличие оператора скрещивания, размеры популяции остаются неизменными, новые особи занимают место старых.

Для эволюционного алгоритма основной движущей силой является оператор мутации. Для выбора особи используем фиксированную вероятность, а для мутации генов формулу (1).

Гены лучших особей будут мутировать реже.

Если текущая мутация дала значение фитнесса меньшее, чем исходное состояние особи, то с фиксированной вероятностью деградации возвращаем исходное значение гена.

Нейронная сеть имеет на внутреннем слое ступенчатые функции активации, а на единственном нейроне внешнего логистическую функцию активации.

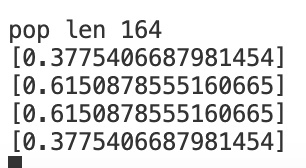
Объект нейронной сети также предоставляет алгоритмам обучения функцию фитнесса, основанную на обратном значении логарифмической функции потери (2) .

# 2 ПРАКТИЧЕСКИЙ РАЗДЕЛ

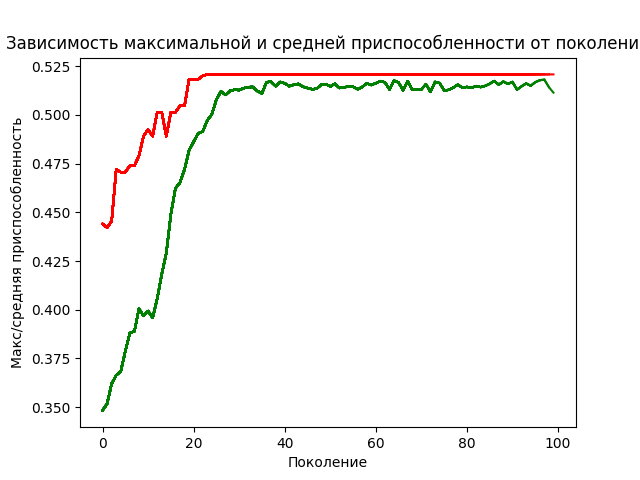
Программное решение было выполнено на языке Python. Использовались библиотеки:

* Numpy для работы со случайными числами
* Matplotlib для отрисовки графиков
* Random для генерации случайных значений

Реализуем следующие классы: neuron – класс нейрона, network – класс нейронной сети, Individual – класс особи для генетического и эволюционного алгоритмов, Population – класс популяции для генетического и эволюционного алгоритмов. Отдельно реализуем функции toWeight и toValue для конвертации кодов Грея значения весов нейронной сети.



**Рисунок 2.1 – Результат ответов системы на тестовую выборку**



**Рисунок 2.2 – История уменьшения фитнеса генетического алгоритма**

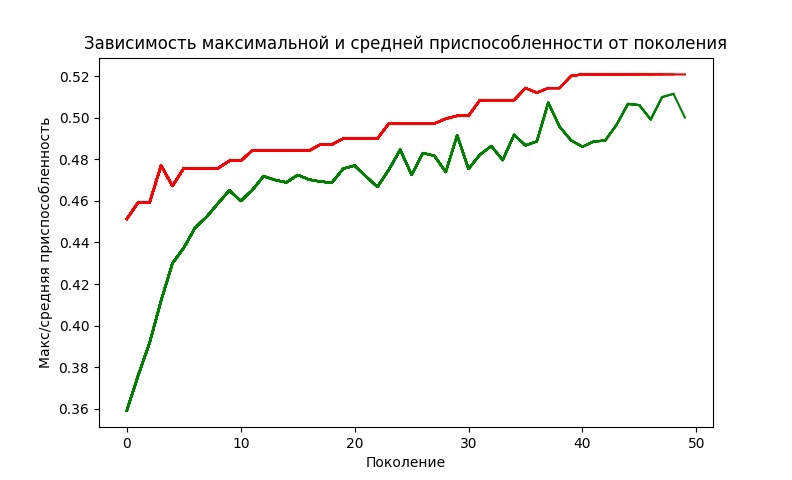


Рисунок 2.3 –История изменения фитнеса функции эволюционного алгоритма

# ЗАКЛЮЧЕНИЕ

Рассмотренные способы обучения нейронной сети справились со своей задачей. Однако, из графиков (Рисунки 2.2-3) видно, что текущая имплементация генетического алгоритма находит удовлетворительное решение немного раньше чисто эволюционной стратегии.

Нейронная сеть со скрытым слоем показала свою способность представлять линейно неразделимые функции, что видно из результатов её обучения (Рисунок 2.1)

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. «Учебник по машинному обучению» от Яндекса [Электронный ресурс]. Глава 3.3. Подбор гиперпараметров. – URL: https://academy.yandex.ru/handbook/ml/article/podbor-giperparametrov (Дата обращения: 11.06.2023)
2. Статья на сайте loginom [Электронный ресурс]: Генетические алгоритмы – математический аппарат — URL: https://loginom.ru/blog/ga-math (Дата обращения: 10.06.2023)
3. Статья на сайте habr [Электронный ресурс]: Генетический алгоритм. Просто о сложном — URL: https://habr.com/ru/articles/128704/
4. Max Jaderberg, Valentin Dalibard, Simon Osindero и др. Population Based Training of Neural Networks – DeepMind, London, UK, 2017 – URL: https://arxiv.org/pdf/1711.09846.pdf (Дата обращения: 11.06.2023)

# ПРИЛОЖЕНИЯ

Приложение А – Программная реализация

**Приложение А – Программная реализация**

Приложение А1 –Реализация нейронной сети

Приложение А2 – Реализация генетического алгоритма

Приложение А3 – Реализация эволюционного алгоритмаъ

*Листинг А1 – Реализация нейронной сети*

import numpy as np

from math import e

from random import random, randint, sample, uniform

from copy import deepcopy

from math import e, tanh, log

# from baseGA import GA

# rnd.seed(1)

# np.random.seed(1)

def sigmoid(x):

return 1/(1+e\*\*(-x))

def inputFunc(x):

return x

def linear(x):

return 1 if x>=0 else 0

# input = [[0, 0, 0], [0, 1, 0], [0, 0, 1], [0, 1, 1], [1,1,1]]

input = [[0, 0], [1, 0], [0, 1], [1, 1]]

output = [0, 1, 1, 0]

class neuron():

def \_\_init\_\_(self, activation, n) -> None:

self.net = 0

self.activation = activation

self.activity = 0

self.w = [random() for i in range(n)]

self.b = random()

pass

def netter():

pass

def \_\_str\_\_(self) -> str:

return "{0}".format(self.activity)

def reset(self):

self.net = 0

*Продолжение Листинга А1*

class network():

def \_\_init\_\_(self, inputN=2, innerN=2, outerN=1, func=sigmoid) -> None:

self.inputLayer = [neuron(inputFunc, inputN) for i in range(inputN)]

self.innerLayer = [neuron(linear, inputN)for i in range(innerN)]

self.outerLayer = [neuron(func, innerN) for i in range(outerN)]

self.number = innerN\*(inputN+1) + outerN\* (innerN+1)

def reset(self):

for i in self.innerLayer:

i.reset()

for i in self.outerLayer:

i.reset()

def compute(self, image):

self.reset()

#заполняем входной слой

for i in range(len(self.inputLayer)):

self.inputLayer[i].net = image[i]

#заполняем сумматоры скрытого слоя

for i in range(len(range(len(self.innerLayer)))):

for j in range(len(self.innerLayer[i].w)):

self.innerLayer[i].net += self.innerLayer[i].w[j]\*self.inputLayer[j].net

self.innerLayer[i].net -= self.innerLayer[i].b

#активируем внутренний слой

self.innerLayer[i].activity = self.innerLayer[i].activation(self.innerLayer[i].net)

#заполняем сумматоры выходного слоя

for i in range(len(range(len(self.outerLayer)))):

for j in range(len(self.outerLayer[i].w)):

self.outerLayer[i].net += self.outerLayer[i].w[j]\*self.innerLayer[j].activity

self.outerLayer[i].net -= self.outerLayer[i].b

#активируем выходной слой

self.outerLayer[i].activity = self.outerLayer[i].activation(self.outerLayer[i].net)

def loss(self, inp=input, perfection=output):

Loss =L= 0

for i in range(len(inp)):

self.compute(inp[i])

outp = self.outerLayer[0].activity

Loss += ((perfection[i] - outp)\*\*2)\*\*0.5

L += -perfection[i]\*log(outp)-(1-perfection[i])\*log(1-outp)

*Продолжение Листинга А1*

print(f"loss {Loss}")

return 1/(L)

def GFit(self, alg):

# Loss = self.loss(fitInput, fitOutput)

# startPoint = [i for i in ]

self.g = alg(self)

pass

def toValue(data):

grey = {"0,0,0,0" : 0 ,

"0,0,0,1" : 1 ,

"0,0,1,1" : 2 ,

"0,0,1,0" : 3 ,

"0,1,1,0" : 4 ,

"0,1,1,1" : 5 ,

"0,1,0,1" : 6 ,

"0,1,0,0" : 7 ,

"1,1,0,0" : 8 ,

"1,1,0,1" : 9 ,

"1,1,1,1" : 10,

"1,1,1,0" : 11,

"1,0,1,0" : 12,

"1,0,1,1" : 13,

"1,0,0,1" : 14,

"1,0,0,0" : 15}

value = 0

for i in range(len(data)):

value+=data[i]\*2\*\*(len(data)-i)//2

value = ((1/(2\*\*len(data)))\*value + 1/(2\*\*len(data))\*(value+1))/2

return value

*Листинг А2 – Реализация генетического алгоритма*

import numpy as np

import math

import random as rnd

from PIL import Image

from matplotlib import cm

individual = np.zeros(240\*240,3)

tmp = np.random.randint(420, size = 240\*240)

tmp2 = np.random.randint(840, size=240\*240)

my\_array = np.zeros((420,840))

for i in range(240):

for j in range(240):

my\_array[j+100][i+100] = 0.1

*Продолжение Листинга А2*

for i in individual:

pass

np.random.seed(123)

the\_one = np.random.rand(420,840)

my\_array+=the\_one

# print(the\_one)

im2 = Image.fromarray(np.uint8(cm.gist\_earth(the\_one)\*255))

# im2.show()

im = Image.fromarray(np.uint8(cm.gist\_earth(my\_array)\*255))

im.show()

*Листинг А3 – Реализация эволюционного алгорима*

from nn import network, toValue, linear, sigmoid

import random

import matplotlib.pyplot as plt

from copy import deepcopy

RANDOM\_SEED = 42

# random.seed(RANDOM\_SEED)

class FitnessMax():

def \_\_init\_\_(self):

self.values = [0]

class Individual(list):

def \_\_init\_\_(self, \*args):

super().\_\_init\_\_(\*args)

self.fitness = FitnessMax()

class EA():

def \_\_init\_\_(self, nn: network = None, n = 4) -> None:

self.nn = nn

self.genLen = (n+1)\*nn.number

# self.genLen = 100

# константы генетического алгоритма

self.POPULATION\_SIZE = 100 # количество индивидуумов в популяции

self.P\_CROSSOVER = 0.2 # вероятность скрещивания

self.P\_MUTATION = 0.7 # вероятность мутации индивидуума

self.P\_CHAD = 0.9

self.P\_BASTARD = 0.1

self.P\_DEGENERACY = 0.8

self.MAX\_GENERATIONS = 50

self.population = self.populationCreator(n=self.POPULATION\_SIZE)

*Продолжение Листинга 3*

self.Counter = 0

self.fitnessValues = list(map(self.fitnessFunc, self.population))

for ind, fitVal in zip(self.population, self.fitnessValues):

ind.fitness.values = fitVal

print(fitVal)

self.maxFit = []

self.meanFit = []

self.lifespan()

def individualCreator(self):

return Individual([random.randint(0,1) for i in range(self.genLen)])

def populationCreator(self, n):

return list([self.individualCreator() for i in range(n)])

def clone(self, value):

ind = Individual(value[:])

ind.fitness.values[0] = value.fitness.values[0]

return ind

def selTournament(self):

offspring = []

for n in range(len(self.population)):

i1 = i2 = i3 = 0

while i1==i2 or i1==i3 or i2==i3:

i1, i2, i3 = random.randint(0, len(self.population)-1), random.randint(0,len(self.population)-1), random.randint(0, len(self.population)-1)

offspring.append(max([self.population[i1], self.population[i2], self.population[i3]], key=lambda ind: ind.fitness.values[0]))

return offspring

def cxOnePoint(self, child1, child2):

s = random.randint(2,len(child1)-3)

child1[s:], child2[s:] = child2[s:], child1[s:]

def mutFlipBit(self, mutant, indpb=0.01):

mutator = self.clone(mutant)

for i in range (len(mutant)):

*Продолжение Листинга А3*

if random.random()<indpb:

mutant[i] = 0 if mutant[i] == 1 else 1

mutant.fitness.value = self.fitnessFunc(mutant)

if mutant.fitness.values[0]<mutator.fitness.values[0]:

if random.random()<self.P\_DEGENERACY:

mutant = mutator

def lifespan(self):

fitnessValues = [individual.fitness.values[0] for individual in self.population]

print(f"fitness values {self.population[0].fitness.values}")

best\_index = fitnessValues.index(max(fitnessValues))

globalElite = deepcopy(best\_index)

self.toWeights(self.population[best\_index])

while max(fitnessValues)<100 and self.Counter<self.MAX\_GENERATIONS:

self.Counter+=1

for mutant in self.population:

if random.random()>self.P\_MUTATION:

# if random.random()>0:

self.mutFlipBit(mutant , indpb=self.P\_MUTATION\*(mutant.fitness.values[0])/10)

offspring = self.selTournament()

# print(f"offspring{self.Counter, offspring}")

offspring = list(map(self.clone, offspring))

print(f"offspring{len(offspring)}")

# print(f"pop {self.population}")

for child1, child2 in zip(offspring[::2], offspring[1::2]):

if random.random()<self.P\_CROSSOVER:

self.cxOnePoint(child1, child2)

freshFitnessValues = list(map(self.fitnessFunc, offspring))

for ind, fitVal in zip(offspring, freshFitnessValues):

ind.fitness.values = fitVal

self.population[:]=offspring

fitnessValues = [ind.fitness.values[0] for ind in self.population]

*Продолжение Листинга А3*

maxFitVal = max(fitnessValues)

meanFitVal = sum(fitnessValues)/len(self.population)

self.maxFit.append(maxFitVal)

self.meanFit.append(meanFitVal)

print(f"Поколение {self.Counter}: Макс приспособ. = {maxFitVal}, Средняя приспособ.= {meanFitVal}")

best\_index = fitnessValues.index(max(fitnessValues))

if self.population[globalElite].fitness.values[0]<=self.population[best\_index].fitness.values[0]:

globalElite = deepcopy(best\_index)

self.toWeights(self.population[globalElite])

print("Лучший индивидуум = ", \*self.population[best\_index], "\n")

print(f"pop len {len(self.population)}")

plt.plot(self.maxFit, color='red')

plt.plot(self.meanFit, color='green')

def fitnessFunc(self, individual):

self.toWeights(individual)

return (self.nn.loss()),

def toWeights(self, instance):

print(instance)

weight = []

for i in range(0,len(instance),5):

weight.append(toValue(instance[i+1:i+5]))

if instance[i]==1:

weight[-1]\*=-1

print(f"Веееессс{weight}")

#входной на скрытый и их смещения

for k in range(len(self.nn.innerLayer)):

for i in range(k\*3,k\*3+3,3):

# print(k,i)

self.nn.innerLayer[k].w = weight[i:i+2]

# print(self.nn.innerLayer[k].w)

self.nn.innerLayer[k].b = weight[i+2:i+3][0]

#скрытый на выходной и их смещения

for k in range(len(self.nn.outerLayer)):

for i in range(k\*3+len(self.nn.innerLayer)\*3,k\*3+3 +len(self.nn.innerLayer)\*3,3):

print(k,i)

self.nn.outerLayer[k].w = weight[i:i+2]

self.nn.outerLayer[k].b = weight[i+2:i+3][0]

def OneMaxFit(self, ind):

return sum(ind),

class FitnessMax():

*Продолжение Листинга А2*

def \_\_init\_\_(self):

self.values = [0]

class Individual(list):

def \_\_init\_\_(self, \*args):

super().\_\_init\_\_(\*args)

self.fitness = FitnessMax()

# plt.plot(maxFitnessValues, color='red')

# plt.plot(meanFitnessValues, color='green')

# plt.xlabel('Поколение')

# plt.ylabel('Макс/средняя приспособленность')

# plt.title('Зависимость максимальной и средней приспособленности от поколения')

# plt.show()

n = network(inputN = 2, innerN = 2, outerN=1, func=sigmoid)

n.GFit(EA)

for i in [[0, 0], [1, 0], [0, 1], [1, 1]]:

# for i in [[0, 0, 0], [0, 1, 0], [0, 0, 1], [0, 1, 1], [1,1,1]]:

n.compute(i)

print([j.activity for j in n.outerLayer])

# g = GA()

plt.xlabel('Поколение')

plt.ylabel('Макс/средняя приспособленность')

plt.title('Зависимость максимальной и средней приспособленности от поколения')

plt.show()