УДК 519.6, 004.42

**И.Д. Камкин**1**, И.А. Седых**2

1Липецкий государственный технический университет, Липецк, Россия;

2Липецкий государственный технический университет, Липецк, Россия

**реализация и сравнение методов градиентного спуска и ньютона на python**

В работе рассмотрены численные методы многомерной оптимизации: градиентный спуск на основе импульса и метод Ньютона. Описаны достоинства, недостатки, а также особенности данных методов и сферы их применения. Приведены алгоритмы методов градиентного спуска и Ньютона, описаны ограничения на условия их применимости. На языке программирования Python с использованием математической библиотеки numpy выполнена программная реализация методов отображения графика функции и движения точки промежуточного минимума по данному графику в процессе работы рассматриваемых методов оптимизации, а также реализация методов градиентного спуска и Ньютона. Numpy использовался для матричных операций. Для визуализации графиков использовалась библиотека matplotlib. Произведено сравнение рассматриваемых методов на примере нахождения минимума непрерывной функции нескольких переменных, а также в задаче минимизации среднеквадратической ошибки в контролируемой классификации. В задаче классификации используется база данных skilern.dataset рукописных цифр, находящаяся в открытом доступе. База данных использовалась в виде пакетного модуля для Python. Для функции двух переменных сравнение методов минимизации с различными значениями параметров осуществлялось с отображением движения точки промежуточного минимума по поверхности исследуемой функции. В задаче минимизации среднеквадратической ошибки в контролируемой классификации на графике отображается величина ошибки в процессе работы методов. Для градиентного спуска импульс оставался фиксированным при различных значениях скорости обучения. Изменяемыми параметрами, которые оказывали влияние на найденный промежуточный минимум, являлись скорость обучения и импульс для метода градиентного спуска и скорость обучения для метода Ньютона. В ходе анализа выяснилось, что минимальные изменения параметров приводят к значительному изменению значения промежуточного минимума в задаче минимизации. Параметры методов оптимизации не являются универсальными и для каждой функции их оптимальные значения будут различны.

**Ключевые слова:** многомерная оптимизация, метод градиентного спуска на основе импульса, метод Ньютона, минимизация, среднеквадратическая ошибка, Python.

**I.D. Kamkin**1**, I.A. Sedykh**2

1Lipetsk State Technical University, Lipetsk, Russia;

2Lipetsk State Technical University, Lipetsk, Russia

**implementation and comparison of gradient descent and Newton methods on python**

The paper considers numerical methods of multidimensional optimization: gradient descent based on momentum and the Newton method. The advantages, disadvantages, as well as the features of these methods and the scope of their application are described. Algorithms of gradient descent and Newton methods are given, restrictions on the conditions of their applicability are described. In the Python programming language, using the numpy mathematical library, the software implementation of methods for displaying the graph of the function and the movement of the intermediate minimum point on this graph in the course of the optimization methods under consideration, as well as the implementation of gradient descent and Newton methods, has been performed. Numpy was used for matrix operations. The matplotlib library was used to visualize the graphs. The comparison of the methods under consideration is made on the example of finding the minimum of a continuous function of several variables, as well as in the problem of minimizing the root-mean-square error in a controlled classification. The classification task uses the skilern.dataset database of handwritten digits, which is publicly available. The database was used as a batch module for Python. For a function of two variables, the comparison of minimization methods with different parameter values was carried out with the display of the movement of the intermediate minimum point along the surface of the function under study. In the task of minimizing the root-mean-square error in the controlled classification, the graph shows the magnitude of the error during the operation of the methods. For gradient descent, the momentum remained fixed at different values of the learning rate. The variable parameters that influenced the intermediate minimum found were the learning rate and momentum for the gradient descent method and the learning rate for the Newton method. During the analysis, it turned out that minimal changes in the parameters lead to a significant change in the value of the intermediate minimum in the minimization problem. The parameters of optimization methods are not universal and their optimal values will be different for each function.

**Keywords**: multidimensional optimization, momentum-based gradient descent method, Newton's method, minimization, root-mean-square error, Python.

## Введение

Одними из самых распространённых методов многомерной оптимизации являются метод градиентного спуска и метод Ньютона [1]. Оба этих метода являются итерационными, то есть значение на следующем шаге рассчитывается исходя из значения на предыдущем. Главными рассмотренными характеристиками обоих методов являются количество сделанных шагов, а также различие между найденным локальным и глобальным минимумами. Существует множество модификаций градиентного спуска, в данной статье будет рассмотрен градиентный спуск на основе импульса, поскольку он является простой модификацией исходного алгоритма, лишённой главных недостатков, а именно попадания в ловушку локального минимума и осцилляции. Градиентный спуск на основе импульса относится к методам первого порядка, в то время как метод Ньютона к методам второго порядка.

Данные методы применяются для поиска экстремумов функции нескольких переменных, в частности для минимизации ошибки в процессе обучения нейронных сетей [2]. На данный момент нейросети активно используются как в анализе данных, так и в прикладных задачах [3]. В работе рассмотрено сравнение методов градиентного спуска и Ньютона при изменении их параметров на примере нахождения минимума непрерывной функции и минимизации среднеквадратической ошибки в контролируемой классификации.

## 1. Методы безусловной оптимизации

## 1.1 Градиентный спуск на основе импульса

Градиентный спуск [4] — простой метод оптимизации, позволяющий найти минимум целевой функции. Это жадный алгоритм, который делает шаг в сторону максимальной скорости убывания функции. Рассмотрим функцию нескольких переменных f(w), где . Чтобы найти , при котором функция достигает своего локального минимума, градиентный спуск делает следующие шаги:

1. Выбирается случайное значение .
2. *i* = 0.
3. Выбирается максимальное количество итераций T.
4. Выбирается значение скорости обучения (шаг сходимости) .
5. Повторять шаги a, b, c пока или число итераций не превысит T.

a. . (1)

b. .

c. *i* = *i* +1.

Здесь - градиент.

Градиент можно рассматривать как вектор, указывающий направление, в котором функция возрастает быстрее всего. Следование отрицательному направлению градиента приведёт к точкам, в которых значение функции уменьшается с максимальной скоростью. Скорость обучения, также называемая размером шага, определяет скорость движения в направлении от градиента.

При использовании градиентного спуска возникают следующие проблемы:

1. Попадание в ловушку локального минимума, что является прямым следствием жадности этого алгоритма.
2. Отсутствие глобального оптимума — это прямой результат слишком быстрого движения по направлению градиента.
3. Осцилляция [5] - это явление, которое возникает, когда значение функции не изменяется существенно независимо от направления, в котором оно движется (так называемое плато).

Чтобы решить эти проблемы, к выражению (1) добавляется импульс α. Данный метод называется градиентным спуском на основе импульса:

. (2)

## 1.2 Метод Ньютона

Метод Ньютона - итерационный алгоритм, который в одномерном случае выглядит следующим образом:

. (3)

В многомерном случае в формуле (3) первая производная заменяется на градиент, а вторая — на матрицу Гессе:

,

,

,

.

Достаточным условием сходимости в методе Ньютона является положительная определённость матрицы Гессе [6].

## 2. Программная реализация

Далее рассмотрена программаная реализация интструментов для анализа работы рассмотренных выше методов оптимизации на языке программирования Python.

## 2.1 Методы графического отображения

Для работы с матрицами используется библиотека numpy [7], для графического представления - библиотека matplotlib [8].

Перед тем, как реализовывать методы оптимизации, создадим базовый класс *Graph* с основными методами. Необходимо хранить несколько переменных:

* Поле *board* — значения исследуемой функции на фиксированном промежутке аргументов.
* Параметры *params* — дополнительные значения, которые понадобятся позже.
* Случайный сид *random\_seed* — число, необходимое для получения псевдослучайных чисел, которые будут всегда одинаковы, при одинаковом значении *random\_seed*.

Ниже представлена реализация:

Далее реализуем получение случайных стартовых значений координат, с учётом *random\_seed*, метод *get\_random\_cords* принимает левую и правую границы *left*, *right* выдаваемых значений, а также количество этих значений *quantity:*

class Graph:

def \_\_init\_\_(self, board: tuple[np.ndarray, np.ndarray], params=None) -> None:

self.board = board

self.params = params if params is not None else tuple()

self.random\_seed = np.random.RandomState(19)

Далее в методе *draw\_board* реализуем визуализацию графика *board*. Отображается график в цветовой схеме «RdBu\_r», а после рисуется стрелка с подписью, указывающая на глобальный минимум:

def get\_random\_cords(self, left, right, quantity) -> np.ndarray:

return self.random\_seed.uniform(left, right, quantity)

В методе визуализации поля использовался метод *draw\_arrow* — этот метод отображает стрелку заданного цвета с надписью, указывающую на некоторую координату:

def draw\_board(self) -> None:

pts, f\_vals = self.board

f\_plot = plt.scatter(

pts[:, 0], pts[:, 1],

c=f\_vals, vmin=min(f\_vals), vmax=max(f\_vals),

cmap='RdBu\_r')

plt.colorbar(f\_plot)

self.draw\_arrow(

min(zip(\*self.board), key=lambda x: x[1])[0],

'yellow', (-5, -10), 'глобальный минимум'

)

Далее реализуем отображение движения точки промежуточного минимума по поверхности во время работы метода оптимизации:

def draw\_arrow(self, cord\_point, color, cord\_text, text):

plt.plot(xy=cord\_point, marker='P', markersize=10, c=color)

plt.annotate(text, xy=cord\_point, xytext=cord\_text,

arrowprops={

"arrowstyle": "->",

"color": color,

"connectionstyle": 'arc3'

})

def draw\_graph\_on\_board(self, points) -> None:

self.draw\_board()

plt.plot(points[:, 0], points[:, 1], marker='o', c='magenta')

self.draw\_arrow(points[-1], 'green', (-1, 7), 'минимум')

for i, w in enumerate(points[:-1]):

plt.annotate(

"",

xy=w, xycoords='data', xytext=points[i+1, :], textcoords='data',

arrowprops={

"arrowstyle": '<-',

"connectionstyle": 'angle3'

})

Чтобы отображать только изменение значения на каждом шаге, реализуем метод *draw\_graph\_on\_ax*:

Далее для универсальности класса нужно задать несколько методов, которые будут «пустыми» в данном классе, однако далее они переопределяются для каждой исследуемой функции:

def draw\_graph\_on\_ax(self, ax, points, color, label) -> None:

\_, y = points

ax.plot(np.arange(y.size), y, color=color, label=label)

* вычисление значения функции,
* вычисление первой производной,
* вычисление матрицы Гессе.

Ниже представлена программная реализация:

def function(self, cords: np.ndarray):

raise NotImplementedError("Обязательно к переопределению.")

def function\_derivative(self, cords: np.ndarray):

raise NotImplementedError("Обязательно к переопределению.")

def function\_hesse(self, cords: np.ndarray):

raise NotImplementedError("Обязательно к переопределению.")

Последнее, что необходимо реализовать — это метод вычисления :

@staticmethod

def calculate\_diff(pred, pred\_pred):

return np.absolute(pred - pred\_pred)

## **2.**2 Реализация градиентного спуска

Используя приведённые ранее инструменты, опишем реализацию метода градиентного спуска на основе импульса. Метод должен возвращать все координаты точек и значения функции в данных точках, обозначим их соответственно *w\_history* и *f\_history*. На вход метода *gradient\_descent* подаются следующие значения:

* *learning\_rate*: float — скорость обучения (шаг сходимости),
* *momentum*: float — импульс,
* *max\_iterations*: int — максимальное количество итераций,
* *threshold*: float — величина значения calculate\_diff, при котором произойдёт остановка,
* *cords\_copy* — координата, с которой начинается алгоритм оптимизации.

Ниже представлена реализация метода градиентного спуска на основе импульса:

def gradient\_descent(self,

learning\_rate: float,

momentum: float,

max\_iterations: int,

threshold: float,

cords\_copy: np.ndarray

) -> tuple[np.ndarray, np.ndarray]:

w\_history = cords\_copy.copy()

f\_history = self.function(cords\_copy.copy())

delta\_w = np.zeros(cords\_copy.shape)

i, diff = 0, 1e10

while i < max\_iterations and diff > threshold:

delta\_w = -learning\_rate \* self.function\_derivative(

cords\_copy

) + momentum \* delta\_w

cords\_copy += delta\_w

w\_history = np.vstack((w\_history, cords\_copy))

f\_history = np.vstack((f\_history, self.function(cords\_copy)))

i += 1

diff = self.calculate\_diff(f\_history[-1], f\_history[-2])

return w\_history, f\_history

## 2.3 Реализация метода Ньютона

Далее опишем реализацию метода Ньютона. Для сходимости алгоритма необходима положительная определённость матрицы Гессе, для этого напишем функцию проверки *is\_pos\_def*:

Именно для метода Ньютона необходим метод *function\_hesse*, приведённый ранее. Далее реализуем сам метод Ньютона, входные и выходные параметры такие же, как и у метода градиентного спуска, исключая импульс из входных параметров.

def is\_pos\_def(self, matrix\_hesse):

return np.all(np.linalg.eigvals(matrix\_hesse) > 0)

def newton(self,

learning\_rate: float,

max\_iterations: int,

threshold: float,

cords\_copy: np.ndarray

) -> tuple[np.ndarray, np.ndarray]:

w\_history = cords\_copy.copy()

f\_history = self.function(cords\_copy)

i, diff = 0, 1e10

while i < max\_iterations and diff > threshold:

hesse = self.function\_hesse(cords\_copy)

grad = self.function\_derivative(cords\_copy)

if self.is\_pos\_def(hesse) and np.linalg.det(hesse) != 0:

hesse\_inverse = np.linalg.inv(hesse)

cords\_copy -= learning\_rate \* np.dot(hesse\_inverse, grad)

else:

cords\_copy -= learning\_rate \* grad

w\_history = np.vstack((w\_history, cords\_copy))

f\_history = np.vstack((f\_history, self.function(cords\_copy)))

i += 1

diff = self.calculate\_diff(f\_history[-1], f\_history[-2])

return w\_history, f\_history

## 3. Примеры сравнения методов

## 3.1 Сравнение на примере функции нескольких переменных

Рассмотрим пример с непрерывной функцией двух переменных вида:

. (4)

Для градиентного спуска на каждой итерации обновляются по формуле (2), а для метода Ньютона по формуле (3). Чтобы сравнить два рассматриваемых метода оптимизации на примере функции (4), необходимо реализовать функцию создания поля *board.* Рассмотрим область от -10 до 10.

Необходимо объявить класс *Paraboloid,* наследуемый от класса *Graph,* для расчёта значений функции (4), градиента и матрицы Гессе.

def init\_graph():

x = np.linspace(-10.0, 10.0, 100)

y = np.linspace(-10.0, 10.0, 100)

w1, w2 = np.meshgrid(x, y)

pts = np.vstack((w1.flatten(), w2.flatten()))

pts = pts.transpose()

f\_vals = np.sum(pts \* pts, axis=1)

return pts, f\_vals

Далее реализуем класс *ViewParaboloid* для отображения поля и пути перемещения точки при различных параметрах. Импульс для градиентного спуска выберем 0.5, скорость обучения будет меняться: 0.05, 0.3, 0.7 и 0.9.

class Paraboloid(Graph):

def \_\_init\_\_(self, board: tuple[np.ndarray, np.ndarray], params=None):

super().\_\_init\_\_(board, params)

def function(self, cords: np.ndarray):

return np.sum(cords \* cords)

def function\_derivative(self, cords: np.ndarray):

return 2 \* cords

def function\_hesse(self, cords: np.ndarray):

return np.array(((2, 0), (0, 2)))

class ViewParaboloid:

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

self.graph = Paraboloid(init\_graph())

def show(self):

cords = self.graph.get\_random\_cords(-10, 10, 2)

fig, \_ = plt.subplots(nrows=4, ncols=4, figsize=(54, 54))

learning\_rates = [.05, .3, .7, .9]

ind = 1

for rate in learning\_rates:

plt.subplot(2, 4, ind)

self.graph.draw\_graph\_on\_board(self.graph.gradient\_descent(

learning\_rate=rate,

momentum=.5,

max\_iterations=100,

threshold=1e-2,

cords\_copy=cords.copy())[0])

plt.subplot(2, 4, ind+1)

self.graph.draw\_graph\_on\_board(self.graph.newton(

learning\_rate=rate,

max\_iterations=10,

threshold=1e-2,

cords\_copy=cords.copy())[0])

ind += 2

plt.text(-39, 12, f'Градиент', fontsize=13)

plt.text(-3, 12, f'Ньютон', fontsize=13)

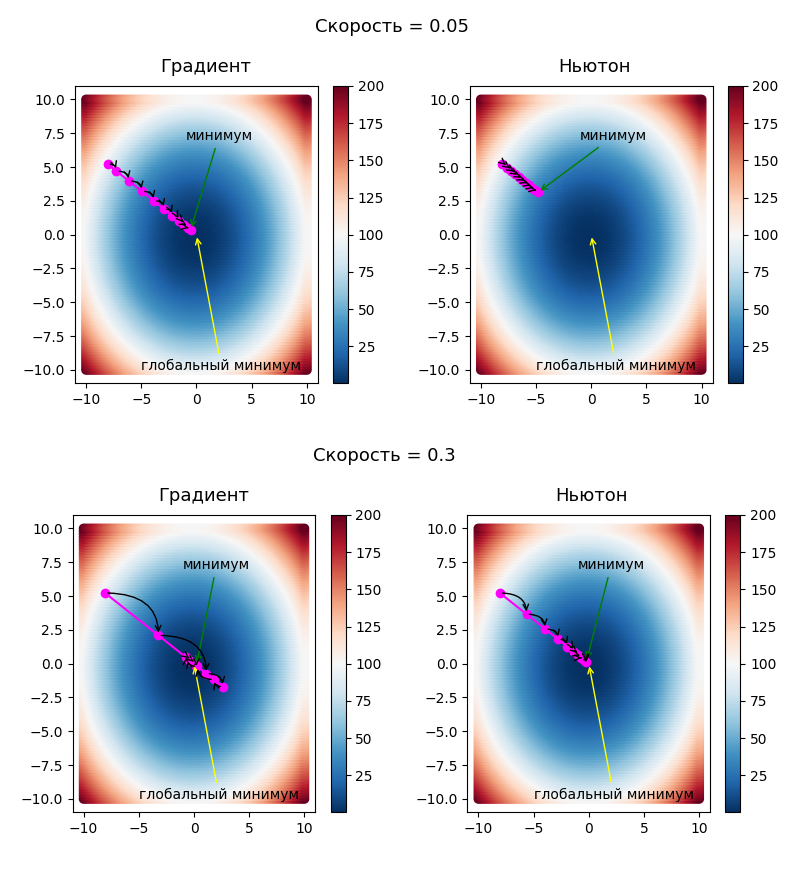
plt.text(-25, 15, f'Скорость = {rate}', fontsize=13)

fig.subplots\_adjust(hspace=.5, wspace=.3)

plt.show()

Ниже представлено создание объекта класса *ViewParaboloid* и вызов у него метода *show*:

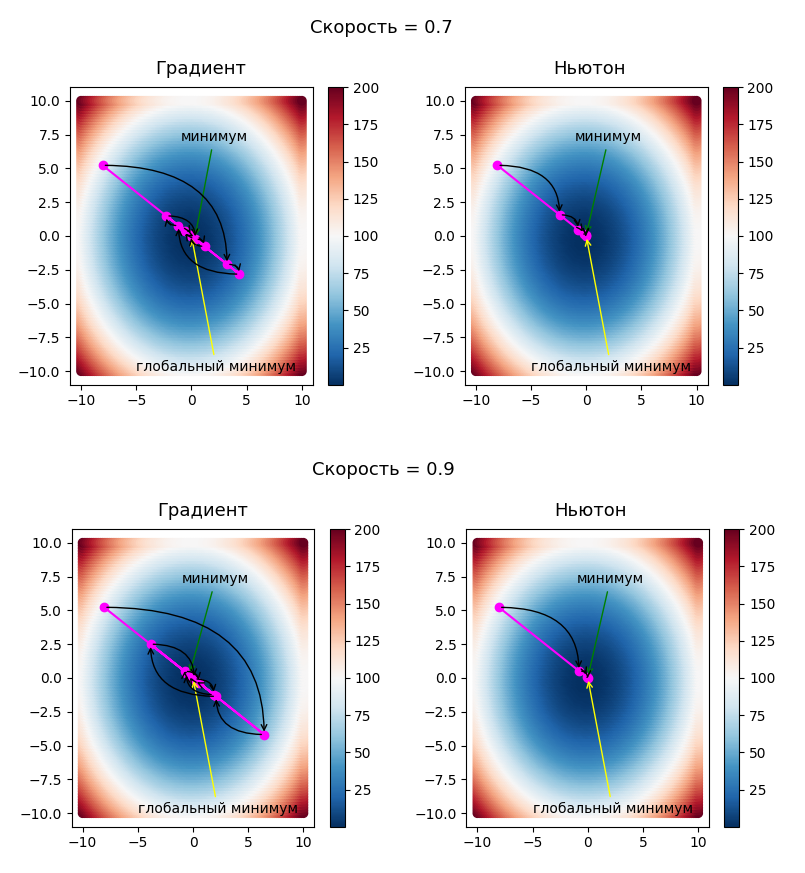
На рисунках 1 и 2 показано движение точки промежуточного минимума в процессе выполнения методов градиентного спуска и Ньютона на примере функции (4) с различными значениями скоростей.

Рис. 1. Сравнение методов на примере функции (4) при скоростях 0.05 и 0.3

from visual import ViewParaboloid

if \_\_name\_\_ == '\_\_main\_\_':

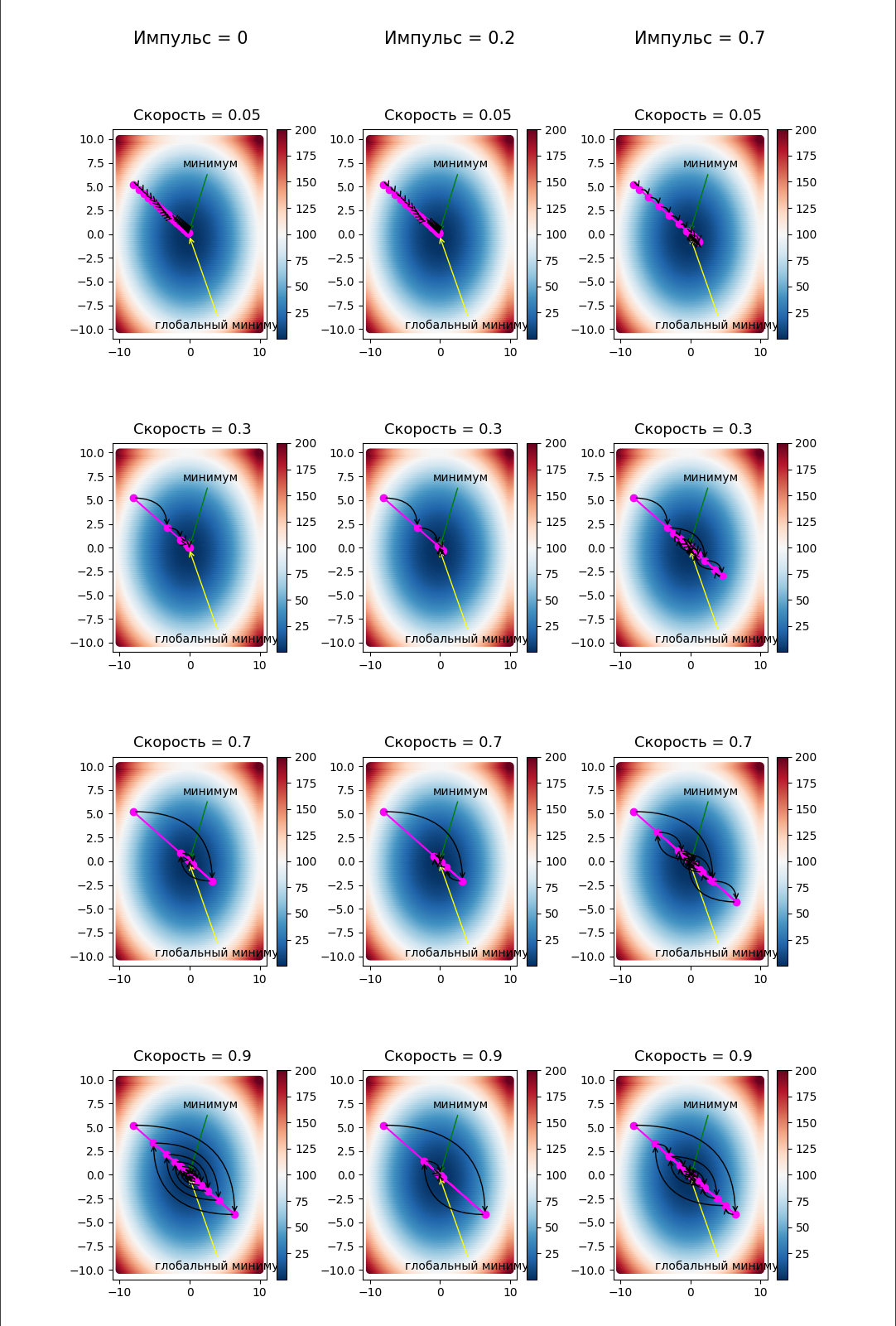
ViewParaboloid().show()

Рис. 2. Сравнение методов на примере функции (4) при скоростях 0.7 и 0.9

На рисунке 1 видно, что при небольшом значении скорости обучения 0.05 ни один из методов не достиг глобального минимума, однако градиентный спуск на основе импульса оказался ближе к оптимуму. Это связано с тем, что, помимо скорости обучения, он имеет импульс, который смещает точку к искомому значению. Начиная со значения скорости обучения в 0.3, градиентный спуск начинает «перепрыгивать» через минимум, а метод Ньютона движется исключительно к искомому значению.

На рисунке 2 видно, что метод Ньютона работает тем лучше, чем больше значение скорости обучения. Градиентный спуск на основе импульса при больших значениях скорости обучения начинает часто перескакивать через искомую точку.

Рассмотрим, как импульс и скорость обучения влияют на найденное значение минимума. Используем те же значения скорости, что и выше, а также несколько значений импульсов: 0, 0.2 и 0.7. (см. рис. 3)

Рис 3. Визуализация градиентного спуска с различными параметрами

На рисунке 3 рассмотрим график с импульсом 0 и скоростью обучения 0.05. Видно, что алгоритм не достиг глобального минимума за установленное количество шагов, однако при увеличении импульса найденный минимум находится ближе к глобальному. Также рассмотрим график с импульсом 0.7 и скоростью обучения 0.9. Видно, что алгоритм совершает колебания, что не является оптимальным вариантом. Для рассмотренного примера оптимально использовать небольшое значение скорости обучения и среднее значение импульса.

## 3.2 Сравнение на примере контролируемой классификации

Рассмотренные ранее методы часто применяют для минимизации среднеквадратической ошибки [9], например в контролируемой классификации, в задаче регрессии, в окрестностном моделировании [10] и в других задачах.

Предположим, что дано *m* обучающих примеров, где каждый пример имеет *n* признаков, то есть задана выборка , где i=1,…,m; j=1,…,n. Если соответствующие целевое и выходное значения для каждого примера равны и соответственно, то функция среднеквадратической ошибки определяется как:

. (5)

В формуле (5) выход определяется взвешенной линейной комбинацией входов, определяемой как:

.

Неизвестным параметром в приведенном выше уравнении является весовой вектор .

Целевая функция в этом случае представляет собой среднеквадратическую ошибку с градиентом, координаты которого определяются выражением:

.

Создадим класс *MSE*, унаследованный от *Graph*. Далее необходимо изменить метод *calculate\_diff*, поскольку значение функции (5) является значением ошибки. Также необходимо написать методы вычисления значений функции (5), градиента и матрицы Гессе.

class MSE(Graph):

def \_\_init\_\_(self, board: tuple[np.ndarray, np.ndarray], params=None):

super().\_\_init\_\_(board, params)

@staticmethod

def calculate\_diff(pred, pred\_pred):

return pred

def function(self, cords: np.ndarray, temp\_params = None):

params = self.params if temp\_params is None else temp\_params

o = np.sum(params[0] \* cords, axis=1)

mse = np.sum((params[1] - o) \*\* 2)

return mse / params[1].size

def function\_derivative(self, cords: np.ndarray):

rows, cols = self.params[0].shape

zn = self.params[1] - np.sum(self.params[0] \* cords, axis=1)

diff = np.tile(zn.reshape((rows, 1)), (1, cols))

grad = -2 \* np.sum(diff \* self.params[0], axis=0)

return grad / self.params[1].size

def function\_hesse(self, cords: np.ndarray):

hesse = 2 \* np.sum(

[np.outer(i, i) for i in self.params[0]], axis=0)

return hesse / self.params[1].size

В качестве данных для визуализации и анализа воспользуемся набором рукописных цифр, включённых в открытую базу данных *sklearn.datasets* [11]. База данных состоит из 360 примеров цифр 0 и 1, которые были разделены на 288 обучающих и 72 тестовых. Входные данные представлены в виде одномерных векторов из 64 значений яркости пикселя величиной от 0 до 16. Выходное значение *-* цифра нуль или единица.

После предварительного анализа данных выяснилось, что на всех изображениях первый и последний столбцы имеют только нулевые элементы, которые не влияют на результат, следовательно можно их исключить. Для этого напишем функцию *alignment* и обработаем входные данные, получив 48 параметров, вместо 64.

Отобразим первые 10 примеров обучающей выборки (см. рис. 4). Массив из 48 значений яркости пикселей является изображением размера 8 на 6, которое выглядит как рукописная цифра. Ниже представлен метод отображения 10-и первых примеров.

@dataclass

class Data:

x: np.ndarray

y: np.ndarray

def get(self):

return self.x, self.y

class ViewMSE:

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

self.graph = MSE(tuple())

digits, target = dt.load\_digits(n\_class=2, return\_X\_y=True)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(

digits, target, test\_size=0.2, random\_state=10

)

x\_train = self.alignment(x\_train)

x\_test = self.alignment(x\_test)

self.train = Data(x\_train, y\_train)

self.test = Data(x\_test, y\_test)

@staticmethod

def alignment(data: np.ndarray):

del\_index = list(range(0, 64, 8)) + list(range(7, 64, 8))

data = np.array([np.delete(i, del\_index) for i in data])

return data

def show\_numbers(self, data):

\_, ax = plt.subplots(

nrows=1, ncols=10, figsize=(12, 4),

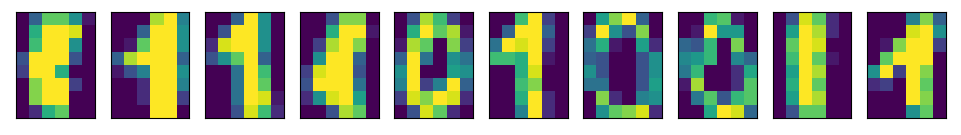
subplot\_kw={"xticks": [], "yticks": []}

)

for i in np.arange(10):

ax[i].imshow(data[i].reshape(8, 6))

Вызовем данный метод для обучающих элементов, результатом будет является последовательность рукописных цифр (см. рис. 4).

Рис 4. Рукописные цифры

Далее рассмотрим работу метода градиентного спуска и метода Ньютона для функции (5). Максимальное количество итераций равно 100, критерием остановки является значение ошибки 0.01, что составляет 1%. Стартовые значения параметров *w* для обоих методов одинаковы. Для градиентного спуска скорость изменяется от до включительно, для метода Ньютона - от до включительно. Ниже представлен метод отрисовки *show*  в классе *ViewMSE*:

def show(self):

cords\_copy = self.graph.get\_random\_cords(

-1, 1, self.train.x.shape[1]

) \* 1e-6

\_, ax = plt.subplot\_mosaic([

[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]

])

self.graph.params = self.train.get()

for ind\_grad, rate\_grad in enumerate(range(-8, -4)):

for ind\_newton, rate\_newton in enumerate(range(-5, -1)):

ax[ind\_grad \* 4 + ind\_newton].clear()

cords\_grad, f\_grad = self.graph.gradient\_descent(

learning\_rate=10 \*\* rate\_grad,

momentum=0.7,

max\_iterations=100,

threshold=0.01,

cords\_copy=cords\_copy.copy()

)

self.graph.draw\_graph\_on\_ax(

ax[ind\_grad \* 4 + ind\_newton],

(cords\_grad, f\_grad),

'green',

'Градиент'

)

cords\_newton, f\_newton = self.graph.newton(

learning\_rate=10 \*\* rate\_newton,

max\_iterations=100,

threshold=0.01,

cords\_copy=cords\_copy.copy()

)

self.graph.draw\_graph\_on\_ax(

ax[ind\_grad \* 4 + ind\_newton],

(cords\_newton, f\_newton),

'red',

'Ньютон'

)

ax[ind\_grad \* 4 + ind\_newton].set\_title(

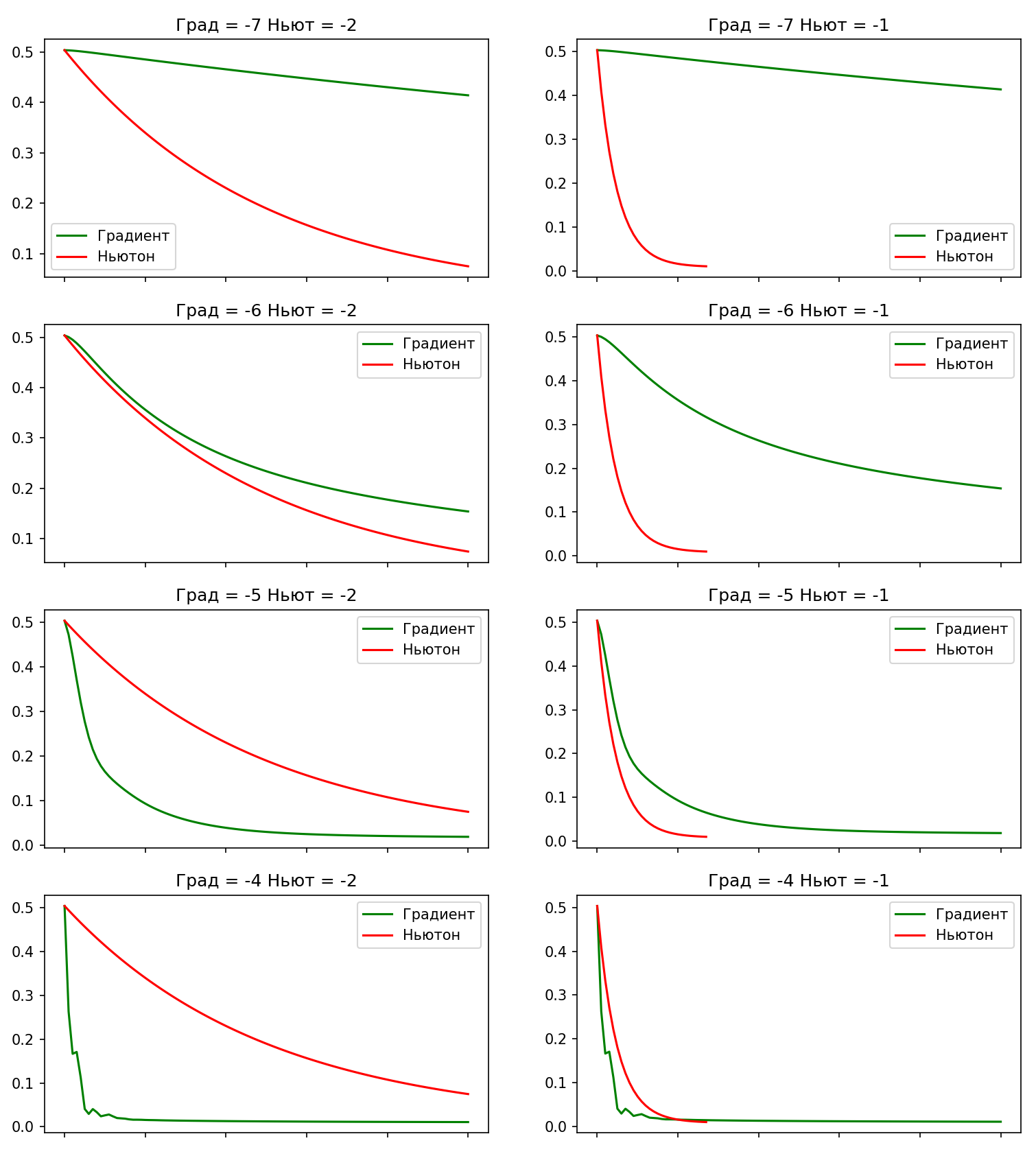
f'Град = {rate\_grad} Ньют = {rate\_newton}')

ax[ind\_grad \* 4 + ind\_newton].legend()

ax[ind\_grad \* 4 + ind\_newton].set\_xticklabels([])

plt.show()

Рассмотрим графики изменения значения ошибки в процессе работы алгоритмов оптимизации (см. рис. 5). На рисунке 5 введены следующие обозначения: Град = *a* соответствует скорости обучения .

Рис. 5. Сравнение методов на примере контролируемой классификации

В приведенном примере оптимальное значение скорости обучения для градиентного спуска является , для метода Ньютона - . При уменьшении скорости для метода Ньютона до ошибка увеличивается, при увеличении скорости для градиентного спуска до ошибка также увеличивается.

## Заключение

Таким образом, в работе были рассмотрены методы градиентного спуска на основе импульса и Ньютона в задачах нахождения минимума непрерывной функции и минимизации среднеквадратической ошибки в контролируемой классификации. Данные методы реализованы с использованием языка программирования Python. В работе было проведено сравнение этих алгоритмов, а также проанализирована зависимость результатов от параметров методов оптимизации. Анализируемыми параметрами являются скорость обучения и импульс. В ходе анализа выяснилось, что незначительное изменение параметров метода оптимизации значительно влияет на результат. Оптимальные значения скорости обучения и импульса зависят от вида минимизируемой функции. Для метода градиентного спуска небольшое значение скорости обучения и среднее значение импульса даёт лучший результат. Для метода Ньютона скорость обучения значительно выше, чем для градиентного спуска, что и было подтверждено в ходе анализа функции (4), а также в минимизации среднеквадратической ошибки (5) в задаче классификации. Результаты работы можно использовать при решении задач регрессионного анализа, в окрестностном моделировании, при обучении нейронных сетей. Разработанная программа позволяет с помощью варьирования параметров выбрать наиболее эффективный метод оптимизации в каждой конкретной задаче. С программной реализацией можно ознакомиться по ссылке [12].

## Библиографический список

1. Калиткин Н.Н., Альшина Е.А. Численные методы: численный анализ. – Москва: Академия. – 2013. – С. 304.
2. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение / пер. с анг. А. А. Слинкина. – 2-е изд., испр. – Москва: ДМК Пресс, – 2018. – С. 652.
3. Sedykh, I. A. Cold rolling neighborhood models with a fuzzy hierarchical structure // Journal of Chemical Technology and Metallurgy. – 2020. – Vol. 55. No 3. – P. 676-680.
4. Гасников А. В. Современные численные методы оптимизации: метод универсального градиентного спуска. Москва: МФТИ, – 2018. – С. 286.
5. Большая российская энциклопедия [Электронный ресурс] / Главный редактор: Кравец С. Л. Москва : БРЭ 2005–2022. URL: <https://bigenc.ru/physics/text/2696871>. (дата обращения: 10.07.22)
6. Захарченков К. В., Мрочек Т. В. Методы оптимизации: учебно-методическое издание. Могилев: Государственное учреждение высшего профессионального образования «Белорусско-Российский университет», – 2018. – С. 44.
7. Wes McKinney Python for Data Analysis. Gravenstein Highway North, Sebastopol: O'Reilly Media, – 2013. – С. 470.
8. Абдрахманов М.И. Devpractice Team. Библиотека Matplotlib. devpractice.ru, – 2019. – С. 100.
9. Кашьяп Р.Л., Рао А.Р. Построение динамических стохастических моделей по экспериментальным данным. Москва: Наука, – 1983. – С. 384.
10. Седых И.А. Идентификация линейных динамических окрестностных моделей с нечеткой иерархической структурой // Вестник Воронежского государственного технического университета. Том 15, №4, – 2019. – С. 7-13.
11. Scikit-learn Machine Learning in Python [Электронный ресурс] INRIA 2011–2022. URL: <https://scikit-learn.org/stable/>. (дата обращения: 22.05.22)
12. GitHub [Электронный ресурс] Microsoft 2008–2022. URL: https://github.com/MR-Geri/gradient\_descent/. (дата обращения: 19.07.22)

**Bibliographic list**

1. Kalitkin N.N., Alshina E.A. Numerical methods: numerical analysis. – Moscow: Academy. - 2013. – P. 304.
2. Goodfellow Ya., Benjio I., Courville A. Deep learning / trans. from ang. A. A. Slinkin. – 2nd ed., edit – Moscow: DMK Press, - 2018. – P. 652.
3. Sedykh, I. A. Cold rolling neighborhood models with a fuzzy hierarchical structure //Journal of Chemical Technology and Metallurgy. – 2020. – Vol. 55. No 3. – P. 676-680.
4. Gasnikov A.V. Modern numerical optimization methods: the method of universal gradient descent. Moscow: MIPT, - 2018. – P. 286.
5. The Great Russian Encyclopedia [Electronic resource] / Editor-in-Chief: Kravets S. L. Moscow : BRE 2005-2022. URL: https://bigenc.ru/physics/text/2696871 . (accessed: 10.07.22)
6. Zakharchenkov K. V., Mrochek T. V. Optimization methods: educational and methodical edition. Mogilev: State Institution of Higher Professional Education "Belarusian-Russian University", – 2018. – P. 44.
7. Wes McKinney Python for Data Analysis. Gravenstein Highway North, Sebastopol: O'Reilly Media, - 2013. – P. 470.
8. Abdrakhmanov M.I. Devpractice Team. The Matplotlib library. devpractice.ru , – 2019. – P. 100.
9. Kashyap R.L., Rao A.R. Construction of dynamic stochastic models based on experimental data. Moscow: Nauka, - 1983. – P. 384.
10. Sedykh I.A. Identification of linear dynamic neighborhood models with fuzzy hierarchical structure // Bulletin of the Voronezh State Technical University. Volume 15, No. 4, – 2019. – P. 7-13.
11. Scikit-learn Machine Learning in Python [Electronic resource] INRIA 2011-2022. URL: https://scikit-learn.org/stable /. (accessed: 22.05.22)
12. GitHub [Electronic resource] Microsoft 2008-2022. URL: https://github.com/MR-Geri/gradient\_descent /. (accessed: 07.19.12)

**Сведения об авторах**

**Камкин Илья Дмитриевич** (Липецк, Россия) – студент Липецкого государственного технического университета (398055, г. Липецк, ул. Московская, 30, e-mail: [ilya.kamckine@ya.ru](mailto:ilya.kamckine@ya.ru)).

**Седых Ирина Александровна** (Липецк, Россия) – доктор технических наук, доцент, профессор кафедры высшей математики Липецкого государственного технического университета (398055, г. Липецк, ул. Московская, 30, e-mail: sedykh-irina@yandex.ru).

**About the authors**

**Kamkin Ilya Dmitrievich** (Lipetsk, Russia) – Student of Lipetsk State Technical University (398055, Lipetsk, Moskovskaya. 30, e-mail: ilya.kamckine@yandex.ru ).

**Sedykh Irina Alexandorvna** (Lipetsk, Russia) – Doctor of Technical Sciences, Associate Professor, Professor of the Department of Higher Mathematics of Lipetsk State Technical University (398055, Lipetsk, Moskovskaya. 30, e-mail: sedykh-irina@yandex.ru).