Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 3

по курсу «Теория искусственных нейронных сетей»

«Методы многомерного поиска»

Студент группы ИУ9-72Б Шевченко К.

Преподаватель Каганов Ю. Т.

Москва, 2023

Цель работы

1. Изучить алгоритмы многомерного поиска 1-го и 2-го порядка.
2. Разработать программы реализации алгоритмов многомерного поиска 1-го и 2-го порядка.
3. Вычислить экстремум функции.

Постановка задачи

Требуется найти минимум тестовой функции Розенброка:

$$f(x) = \sum_{i=1}^{n-1} [a(x_i^2 - x_{i+1})^2 + b(x_i - 1)^2] + f_0.$$

1. Методами сопряжённых градиентов (методом Флетчера-Ривза и методом Полака-Рибьера).
2. Квазиньютоновским методом (Девидона-Флетчера-Пауэлла).
3. Методом Левенберга-Марквардта.

   В качестве методов одномерного поиска использовать любой из известных методов одномерного поиска.

   Реализовать также генетический алгоритм, используя функцию Розенброка в качестве целевой функции.

Индивидуальный вариант

Параметры для функции Розенброка:
$a = 10, b = 250, f_0 = 45, n = 3$.

Реализация

Исходный код программы приведён в листингах 1–9.

Листинг 1: Импорт необходимых зависимостей

```
1 from typing import Callable
2 import numpy as np
3 import pandas as pd
```

Листинг 2: Функция Розенброка

```python
def rosenbrock_func(a: float, b: float, f0: float, n: int) -> Callable:
    def rosenbrock_func_wrapper(x: np.ndarray) -> float:
        return sum(a * (x[i] ** 2 - x[i + 1]) ** 2 + b * (x[i] - 1) ** 2 for i in range(n - 1)) + f0
    return rosenbrock_func_wrapper
```

Листинг 3: Градиент функции Розенброка

```python
def rosenbrock_func_grad(a: float, b: float, f0: float, n: int) -> Callable:
    def rosenbrock_func_grad_wrapper(x: np.ndarray) -> np.ndarray:
        grad = []
        df_dx0 = 4 * a * x[0] * (x[0] ** 2 - x[1]) + 2 * b * (x[0] - 1)
        grad.append(df_dx0)
        for i in range(1, n - 1):
            df_dxi = - 2 * a * (x[i - 1] ** 2 - x[i]) + 4 * a * x[i] * (x[i] ** 2 - x[i + 1]) + 2 * b * (x[i] - 1)
            grad.append(df_dxi)
        df_dxn_add_1 = - 2 * a * (x[n - 2] ** 2 - x[n - 1])
        grad.append(df_dxn_add_1)
        return np.array(grad)
    return rosenbrock_func_grad_wrapper
```

Листинг 4: Гессиан функции Розенброка

```python
def rosenbrock_func_hessian(a: float, b: float, f0: float, n: int) -> Callable:
    def rosenbrock_func_hessian_wrapper(x: np.ndarray) -> np.ndarray:
        H = []
        df_dx0_dx0 = 4 * a * (x[0] ** 2 - x[1]) + 8 * a * x[0] ** 2 + 2 * b
        df_dx0_dx1 = - 4 * a * x[0]
        H.append([df_dx0_dx0, df_dx0_dx1] + [0 for _ in range(n - 2)])
        for i in range(1, n - 1):
            df_dxi_dxi_min1 = - 4 * a * x[i - 1] - 4 * a * x[i]
            df_dxi_dxi = 2 * a + 12 * a * x[i] ** 2 + 2 * b
            df_dxi_dxi_add_1 = - 4 * a * x[i]
            H.append([0 for _ in range(i - 2)] + [df_dxi_dxi_min1, df_dxi_dxi, df_dxi_dxi_add_1] + [0 for _ in range(i + 2, n)])
        H.append([0 for _ in range(n - 2)] + [-4 * a * x[n - 2], 2 * a])
        return np.array(H)
    return rosenbrock_func_hessian_wrapper
```

Листинг 5: Равномерная норма вектора

```python
def vector_norm(vec: np.ndarray) -> float:
    return max(map(abs, vec))
```

Листинг 6: Метод одномерного поиска (золотое сечение)

```python
def find_min(func: Callable, a: float, b: float, xk: np.ndarray, dk:
↪  np.ndarray, eps: float=1e-4):
    phi = 1.618
    a1 = b - (b - a) / phi
    b1 = a + (b - a) / phi

    while abs(b1 - a1) > eps:
        if func(xk + a1 * dk) < func(xk + b1 * dk):
            b = b1
        else:
            a = a1

        a1 = b - (b - a) / phi
        b1 = a + (b - a) / phi

    return (a + b) / 2
```

Листинг 7: Метод Флетчера-Ривза

```python
def fletcher_reeves(func: Callable, func_grad: np.ndarray, x0: np.ndarray, eps:
↪  float=1e-5) -> tuple[np.ndarray, float, int]:
    dk = -func_grad(x0)
    xk = x0.copy()
    k = 0
    while True:
        if vector_norm(func_grad(xk)) < eps:
            break
        # используем метод одномерной оптимизации для поиска минимума функции
        alpha = find_min(func, -1000, 1000, xk, dk, 1e-4)
        if alpha > 1e-4:
            xk1 = xk + alpha * dk
        wk = (vector_norm(func_grad(xk1)) ** 2) / (vector_norm(func_grad(xk))
        ↪  ** 2)
        dk = -func_grad(xk1) + wk * dk
        xk = xk1.copy()
        k += 1
    return xk, func(xk), k
```

Листинг 8: Метод Полака-Рибьера

```python
def polack_ribeir(func: Callable, func_grad: np.ndarray, x0: np.ndarray, eps:
↪  float=1e-5) -> tuple[np.ndarray, float, int]:
    n = len(x0)
    dk = -func_grad(x0)
    xk = x0.copy()
    k = 0
    while True:
        if vector_norm(func_grad(xk)) < eps:
            break
        # используем метод одномерной оптимизации для поиска минимума функции
        alpha = find_min(func, -1000, 1000, xk, dk, 1e-4)
        if alpha > 1e-4:
            xk1 = xk + alpha * dk
        if k % n == 0:
```

```
            wk = np.dot(func_grad(xk1), (func_grad(xk1) - func_grad(xk))) /
            ↪  (vector_norm(func_grad(xk)) ** 2)
        else:
            wk = 0
        dk = -func_grad(xk1) + wk * dk
        xk = xk1.copy()
        k += 1
    return xk, func(xk), k
```

Листинг 9: Метод Девидона-Флетчера-Пауэлла

```
def davidon_fletcher_powell(func: Callable, func_grad: np.ndarray, x0:
↪  np.ndarray, eps=1e-5) -> tuple[np.ndarray, float, int]:
    Gk = np.eye(len(x0))
    xk = x0.copy()
    k = 0
    while True:
        if vector_norm(func_grad(xk)) < eps:
            break
        dk = np.dot(-Gk, func_grad(xk))
        # используем метод одномерной оптимизации для поиска минимума функции
        alpha = find_min(func, -1000, 1000, xk, dk, 1e-4)
        if alpha > 1e-4:
            xk1 = xk + alpha * dk
        delta_xk = xk1 - xk
        delta_gk = func_grad(xk1) - func_grad(xk)
        delta_Gk = np.outer(delta_xk, delta_xk) /
        ↪  (np.dot(np.transpose(delta_xk), delta_gk)) \
                - np.outer(np.dot(Gk, delta_gk), np.dot(Gk, delta_gk))  \
                / (np.dot(np.transpose(np.dot(Gk, delta_gk)), delta_gk))
        Gk = Gk + delta_Gk
        xk = xk1.copy()
        k += 1
    return xk, func(xk), k
```

Листинг 10: Метод Левенберга-Марквардта

```
def levenberg_marquardt(func: Callable, func_grad: np.ndarray, H: np.ndarray,
↪  x0: np.ndarray, eps=1e-5) -> tuple[np.ndarray, float, int]:
    xk = x0.copy()
    muk = 1e1
    k = 0
    while True:
        if vector_norm(func_grad(xk)) < eps:
            break
        Hxk = H(xk)
        xk1 = xk - np.dot(np.linalg.inv(Hxk + muk * np.eye(Hxk.shape[0],
        ↪  Hxk.shape[1])), func_grad(xk))
        if func(xk1) < func(xk):
            k += 1
            muk = muk / 2
            xk = xk1.copy()
        else:
            muk = 2 * muk
    return xk, func(xk), k
```

Листинг 11: Генетический алгоритм

```python
def fitness_func(x: np.ndarray):
    return 1 / func(x)

def genetic_algorithm(func: Callable, n: int) -> tuple[np.ndarray, float]:
    t, Np, k, Mp = 0, 10, 1, 60
    x0 = np.zeros(n)
    # формируем исходную популяцию с количеством особей Mp
    population = np.array([np.random.uniform(0, 1, n) for _ in range(Mp)])
    fitness_values_individuals = np.array(list(map(fitness_func, population)))
    fitness_value_population = np.sum(fitness_values_individuals)

    while t != Np:
        k = 1
        while k < Mp:
            # этап селекции (отбора особей) для последующего их скрещивания
            q = fitness_values_individuals / fitness_value_population
            new_population = [population[np.random.choice(len(population),
            ↪  p=q)] for _ in range(Mp)]

            # этап скрещивания
            Pc = 0.4
            parent_indices = []
            for i in range(int(Pc * Mp)):
                r = np.random.random()
                if r < Pc:
                    parent_indices.append(i)

            for parent1_index, parent2_index in zip(parent_indices[0::2],
            ↪  parent_indices[1::2]):
                c = np.random.random()
                parent1, parent2 = new_population[parent1_index],
                ↪  new_population[parent2_index]
                parent1 = c * parent1 + (1 - c) * parent2
                parent2 = (1 - c) * parent1 + c * parent2

            # этап мутации
            Pm = 0.1
            for i in range(int(Pm * Mp)):
                r = np.random.random()
                if r < Pm:
                    p = np.random.randint(0, n, 1)
                    new_population[i][p] = np.random.uniform(0, 1)

            population[:] = new_population
            fitness_values_individuals = np.array(list(map(fitness_func,
            ↪  population)))
            fitness_value_population = np.sum(fitness_values_individuals)
            k += 1
        t += 1
    best_individual = population[np.argmax(fitness_values_individuals)]
    return best_individual, func(best_individual)
```

Тестирование

Исходный код для тестирования разработанной программы представлен в листинге 12.

Листинг 12: Тестирование

```
a, b, f0, n = 10, 250, 45, 3
x0 = np.random.uniform(-10, 10, n)
func = rosenbrock_func(a, b, f0, n)
func_grad = rosenbrock_func_grad(a, b, f0, n)
H = rosenbrock_func_hessian(a, b, f0, n)
data = {
    'fletcher_reeves': [],
    'polack_ribeir': [],
    'davidon_fletcher_powell': [],
    'levenberg_marquardt': []
}
for eps in (1e-2, 1e-3, 1e-4, 1e-5):
    print(f'{eps = }')
    it_nums = []
    for method in (fletcher_reeves, polack_ribeir, davidon_fletcher_powell,
    ↪  levenberg_marquardt):
        x, y, it_num = method(func, func_grad, x0, eps) if method.__name__ !=
        ↪  'levenberg_marquardt' else method(func, func_grad, H, x0, eps)
        data[f'{method.__name__}'].append(it_num)
        print(method.__name__, x, y, it_num, sep='\n', end='\n\n')

df = pd.DataFrame(data=data, index=['1e-2', '1e-3', '1e-4', '1e-5'])
df.index.name = 'eps'
styled_df = df.style.set_table_styles([{'selector': '', 'props': [('border',
↪  '2px solid black')]}])\
.set_properties(**{'border': '2px solid black'})
styled_df
```

В качестве метода одномерного поиска использовался метод золотого сечения. Для заданных параметров индивидуального варианта функция Розенброка достигает минимума в точке $x = (1, 1, 1)$, при этом $f(x) = 45$.

Сравнительный анализ методов многомерного поиска проводился путём сравнения количества итераций, необходимых для сходимости, для разных чисел ε – параметров останова итерационного процесса.

На рисунке 1 представлена сравнительная таблица методов многомерного поиска для $\varepsilon = e^{-2}, e^{-3}, e^{-4}, e^{-5}$.

eps	fletcher_reeves	polack_ribeir	davidon_fletcher_powell	levenberg_marquardt
1e-2	18	18	7	7
1e-3	22	18	8	8
1e-4	26	24	8	9
1e-5	32	24	9	10

Рисунок 1 — Сравнительная таблица методов многомерного поиска

Результаты показали, что метод Полака-Рибьера немного превосходит метод Флетчера-Ривза по скорости сходимости. Данные методы являются градиентными, и различаются лишь способом обновления коэффициента в направлении спуска. Метод Полака-Рибьера в некоторых случаях может обеспечивать более эффективное направление для поиска минимума функции. Самыми быстрыми оказались методы Девидона-Флетчера-Пауэлла и Левенберга-Марквардта. Метод Девидона-Флетчера-Пауэлла является квазиньютоновским методом, отличающимся большей адаптацией к кривизне функции, что объясняет его быструю сходимость. Метод Левенберга-Марквардта, сочетающий в себе градиентный спуск и метод Ньютона, показывает наибольшую эффективность и устойчивость среди рассмотренных методов, однако налагает ограничения на целевую функцию (ограниченность снизу, наличие непрерывных вторых частных производных).

Тестирование генетического алгоритма проводилось при различных начальных приближениях искомой точки минимума. Размер исходной популяции составлял 60 особей. Вероятность скрещивания – 0.4, вероятность мутации – 0.1. На рисунке 1 показан результат работы алгоритма.

```
[0.98824822 0.94726039 0.90163219] 45.73870851249531
```

Рисунок 2 — Результат работы генетического алгоритма

Генетический алгоритм, хотя и показал себя эффективным в нахождении глобального минимума, был значительно медленнее остальных методов. Его преимущество заключается в способности избегать локальных минимумов, однако на функции Розенброка это преимущество не было столь значительным.

Выводы

В ходе выполнения лабораторной работы были реализованы методы многомерного поиска 1-го и 2-го порядка на языке высокого уровня Python. Дополнительно реализован генетический алгоритм и проведён сравнительный анализ методов на примере функции Розенброка.

Полученные результаты позволяют сделать следующие выводы:

1. Вычисление минимума для функции Розенброка является нетривиальной задачей ввиду сложности сходимости к глобальному минимуму из-за особенностей характера функции. Поэтому использование её в качестве целевой функции для тестирования алгоритмов оптимизации является хорошей практикой.
2. Наилучшую устойчивость и эффективность в случае нелинейных функций показывает метод Левенберга-Марквардта, поскольку данный алгоритм сочетает в себе градиентный и ньютоновский подходы.
3. Генетический алгоритм показывает хорошую эффективность в нахождении глобального минимума, однако его недостатком является время работы по сравнению с другими методами. Вместе с тем для корректной работы алгоритма необходимо осуществить корректный подбор параметров.